PROFESSOR: All right. Can everyone hear me? So I guess it's 2:35, and we might as well get started. So the calendar, I think, still says that we have a guest lecture today but sorry to disappoint you, there is no guest lecture. Instead we have a quiz for you on Thursday, which is so much more fun. Not really.

But I do realize that everybody has a lot to do, even just for this class. You guys have your final project and so on. So hopefully this will be helpful. It won't take the entire 1 and 1/2 hours, I don't think. And the remaining time will be up in office hours, answering questions about stuff.

So the first thing is I decided to run some stats that we haven't run before for this class. These are your Project 42 4.2 final submissions. And in the red are your beta submission run times. And in the blue are your corresponding final submission run times.

I thought it was just interesting to see some groups did massive improvements, while others didn't really change their programs much. And just to give you an idea, a 90% for your performance grade is indicated by that green line and below. So the majority of the class did really well.

All right. So the thing that everybody's concerned about. So Quiz 2 will be in lecture on Thursday. It'll be an 80 minute quiz, similar in format to the previous one. And you guys once again are allowed a one-page handwritten crib sheet.

And it covers everything that has been done since Quiz 1. It's not designed to be cumulative, but unfortunately due to the nature of the class, there's going to be some things from the previous part of the class that you would need to know or it would be helpful to know.

Unlike last time, we didn't post a practice quiz on Stellar, because it turns out that the material and the order of the material gets shuffled around quite a bit every year. And we have different focuses. So posting a practice quiz would actually be pretty mean, because it would cause you guys to study the wrong things instead.

Instead I try to extract the most useful things from our discussions of what the quiz is going to be this time. So hopefully all this information is helpful. Feel free to interrupt me at any time if you have questions.

So the first part, I would roughly call analyzing and possibly writing parallel programs. And to start off, is everybody familiar with what one of these graphs represents? So you have work and then each circle is a unit amount of work. And then the graph shows the dependencies of the things. So just to start off with a really simple question, what is the total amount of work represented?

AUDIENCE: [LAUGHTER]

PROFESSOR: Uh oh. I heard 17, which is not what I got when I counted this last night at 2:00.

AUDIENCE: 26.

AUDIENCE: 24.

PROFESSOR: I like 24.

AUDIENCE: [LAUGHTER]

PROFESSOR: And I'm going to assume it's 24, and not embarrass myself by trying to count it. Now what is the span?

AUDIENCE: 8.

AUDIENCE: 8.

AUDIENCE: 8.

PROFESSOR: 8 sounds right. So the longest path-- there's actually a couple different paths that

end up being 8, but I think 8's the right answer. So then parallelism?

AUDIENCE: 3.

PROFESSOR: Cool. All right. So now that we're done with that, how about a little bit of background on caches. So when you run programs in parallel, it causes some more interesting things to happen in cache that did not happen with serial execution. One of those things is true sharing-- cache misses-- which are caused when multiple processors, like multiple threads, are trying to operate on the same piece of data in memory.

So as a result, your cache would be forced to synchronize that cache line and move it across-- copy it from the freshest processor to the other ones every time that it's requested. And you end up with a lot of overhead generated by continually doing these copies. So ideally you don't want to do this. But it's a lot easier said than done.

False sharing is a little bit trickier because it's not immediately apparent from looking at your program that you're having multiple threads do the-- operate on the same data. Instead with false sharing, you have multiple processors that want to operate on different locations in memory. But due to the way that your computer's cache is organized, they just happen to be in the same cache line.

As a result, you end up with the same memory, the same overhead, that you would see with true sharing. But when you look at the code itself, you don't think that you're doing anything that involves multiple threads operating on the same data.

And so with that, here's a piece of code. And my question is this true sharing or false sharing? Let's take a vote. How many people think it's true sharing? Nobody. Well, how many people think it's false sharing? Come on. You have to vote one way or another.

Let's try this again. True sharing? False sharing? That's better. All right, so it is indeed false sharing. And can someone propose a solution to fix it? Yes?

AUDIENCE: Separate A and B, and put them in different shelves.

PROFESSOR: Would that be sufficient? Putting them in different trucks?

AUDIENCE: Then, padding the trucks.

PROFESSOR: Yeah. Maybe with a little bit of padding, depending on your system. But sure, I'll accept that. But on the quiz, you might be asked to propose solutions to problems that you find. And hopefully those will be easier.

Now on to a more fun topic-- synchronization. Yes?

AUDIENCE: I have a question on sharing. So if two programs only access data, does that still count ?

PROFESSOR: If you only access the data and nothing in the cache line is being modified, then you're safe, because every processor can independently read from their own copy of that.

AUDIENCE: So that doesn't count.

PROFESSOR: That doesn't count, correct. But the problem is if your cache line is 64 kilobytes, and any thread on any processor changes one piece of data inside that cache, then everybody has to pull new copies.

So the classic example to demonstrate synchronization correctness is the dining philosophers problem, where you have n-- I believe five in this case-- philosophers, and five chopsticks. And each philosopher has to eat. So they use mutexes to-- well, generally you need-- each philosopher's code would be pick up two chopsticks and then eat and then put down their two chopsticks.

It's somewhat of a contrived example, but it actually comes up in a lot of cases. And one of the cases where it comes up is in the system that I use to grade your unit tests. So I got impatient about running 200 or so unit test in series.

And since I had about 200 cores altogether on the cloud machines, I figured, why not do them in parallel? But the problem is I need to run-- if I have students A and

B, I need to run student A's test against student B's implementation.

So the rough way that I did that was I grabbed a lock on student A's repository, grabbed a lock on student B's repository, and then modified-- swapped in their test files, did all my testing, and then returned the files to the original conditions, and then let go of the locks. Unfortunately, when you do that, you run into much of the same problem as the dining philosophers problems.

So the naive strategy is for each philosopher, you grab a lock on your left chopstick. You grab a lock on your right one, with respect to you. And then you eat and then you unlock them in the same order. So this has a bug. Can anyone point out what it is?

AUDIENCE: You want to unlock in a certain order.

PROFESSOR: Actually, it doesn't matter which order you unlock in in in this case. So bad things happen if everybody starts at the exact same time. So everybody grabs their left chopstick. there's five people, five chopsticks, so that works fine.

Now everybody wants to grab their right one, but it's not there. Because the person to the right already grabbed it as their left. So they wait forever for the right chopstick to appear and it never does.

So the situation is called a deadlock. And it's obvious to spot, because all of your threads are going to be stuck doing absolutely nothing. Your CPU is not doing anything. And you're just sitting there.

So a very common work-around that people like for this is to use what's called a try lock pattern. So what you do is you grab a lock on one of them, like, say, your left one. And then for the right one, instead of directly going and trying to grab the lock, you ask-- you basically do-- if the thing is unlocked, grab it. Otherwise, don't grab it.

So it's a non-blocking approach to trying to get a lock. So it either succeeds and you have the lock, or it instantly fails. And if it fails, then you switch the order of which-- you put down the chopstick that you grabbed, and then you try the next one. Maybe

you wait a little bit before then.

This is more or less than what I did for your beta test, for your unit test tester. When it fails to-- if first grabs one student's directory's locks. And then it tries to grab the second one. If that fails, then it puts down both locks, and then it waits a little bit, and then it tries again. So what's the problem with this approach? Yes?

AUDIENCE: It can still hold up.

PROFESSOR: What's that?

AUDIENCE: It can keep waiting and grabbing, and sometimes it will contend.

PROFESSOR: Yeah, pretty much, the same problem happens. Only now you have the case where everybody grabs their left, and now the right one's gone. So they put down their left one. And then they all go to grab the right one. Now they have the right one, and now they want the left one. But the left one not there.

So in theory, they can end up doing this for a very, very long time. But in practice what happens is because of scheduling overhead, and the phase of the moon and whatever else affects computing, you'll end up after a while, this become distinct from this pattern and eventually all the locks go through, and you're OK. But in the meantime, you spent a lot of time spinning, trying to grab locks.

And so this situation is called a livelock. And it's a lot more annoying than a deadlock because it happens-- it tends to be less reproducible. And it's harder to diagnose, because if you're looking at top or your favorite CPU monitor, you see that the program seems to be making progress. Maybe some of them actually get the lock and go through. But the rest of them don't. So it's hard to argue whether or not there's correctness issue.

Now there's various classes of solutions to the dining philosophers problem, but you guys can go look that up. I'm not going to go over them.

So another issue with synchronization is suppose you have two threads. Let's say this is a Mars Rover, and it's on Mars. And the two things that the programmer

wanted to do are send back its debugging log, so that the nerdier people at NASA, I suppose, can read them over, or whatever they do with logs. And the high priority, what everybody actually wants, is for the Rover to send back pictures.

Now both of these things require access to the radio that's supposed to transfer something. And only one thing can transmit at time. Otherwise, the transmission is garbled. So a reasonable approach might be to just lock the radio in each thread before you transmit, and then unlock after you transmit. So what could be a potential issue with this?

AUDIENCE: One might get all the time.

PROFESSOR: Let's say, the scheduler interrupts every 10 milliseconds, and then 1 out of the 10 times, it allows 10 logs to run. And then 9 out of the 10 times, it will allow some pictures to run. So everybody gets time scheduled.

So I guess the easiest way to illustrate this is what happens if it's currently running said logs, and it grabs this lock. And then while it's transmitting, let's say on the second megabyte, the scheduler says your time is up. And then it swaps in this program, and it tries to run.

So now when this program runs, it tries to lock the radio, but the radio's already locked. And the one that locked the radio is the send locks thread. And so it could sit there and spin for a while waiting for the lock to be available. But that's not going to happen. Because the guy that's holding the lock is a bug.

So in general, this is called starvation, where in the process of having a shared resource, you end up unfairly depriving someone else of a resource when you're not truly using it. Well, I guess that's not the-- I guess you are using it, but you're using it for longer than is fair for you.

Like in the scheduler you wanted 90% to one process and 10% to the other, but in reality you end up with a completely different set of priorities. And this inversion of priorities is commonly called priority inversion. Does that make sense? Cool.

So on a completely different topic. So Cilk hyperobjects were covered extensively in lecture. You guys had a problem set on it, although it was not received very warmly from what I see in the problems sets write ups. But I think all of the basic material for what a Cilk hyperobjects is and what it does can be found in the lecture material.

But more interesting, let's explore what you can do with hyperobjects that aren't in the documentation. So I'm going to define a piece of code here. And it's C++ or Cilk++ code.

And the first part of it defines a very simple structure called a point. And it has three members. Or actually it has two members, x and y, and it has three methods for setting the values of x and y and also getting the values of x and y. Question? No? OK.

And then there's also a hyperpoint class which contains some extra Cilk junk. And it also implements the same three methods. Set, get x, and get y. And if you look at the monoid that it contains the point monoid it implements a reduce function that just doesn't do anything. And for the public methods, the setters and getters, they grab the point represented by the local view to the reducer that it contains. And it calls the method that you called on that point.

So the question is what does this thing do, the hyperpoint object? Anybody have any wild guesses at this point? That's a no. So OK. Let's try using this code.

So to put the example into perspective. Suppose I have this artificially contrived function that takes an array of 100 points, and it tries to do an in place reversal of its elements. So one way that it can do that is you have a global temporary point.

And you, at every step of the iteration, first you step through half of the list. And then at every step of the iteration, you copy the ith element into the temporary variable. And then you set the ith from the other end to the-- you swap the two. And then you-- yeah, so if it is a three-variable swap, I guess is the simplest way of explaining it. It's a swap using a using a temporary variable. Anyone not get what that code does, despite my attempts to confuse you?

So as far as this space complexity of this procedure, really what it needs at minimum is enough memory to store the temporary point, and then one register to use to put in the temporary values. You can think of a solution that uses only one register to update values within your point structure.

And so let's suppose I want to do this in parallel. And so what I do is I call it parallel reverse, and I replace for with Cilk for. So am I done? What's wrong with my code?

AUDIENCE: Race.

PROFESSOR: Yes. Well first it takes that much-- it takes the same amount of space. But there is a data race on temp, because multiple threads could be updating temp. So now what I do is I replace my point with a hyperpoint. So now what does the hyperpoint do? So within each iteration of the Cilk for loop, so if that iteration has been-- if the work of that iteration has been stolen or spawned off. Then the hyperobject, because it has a reducer within it, it'll get a new local view which is a new point object to use. And if it's not still in that it continues running in the same thread, then it actually just continues using the local view that was previously instantiated.

So no matter what Cilk does in the background, you're guaranteed to not have any data races on temp, because whenever you spawn off a new thread, the hyperobject will instantiate a new temp object for you to use. And this is an example of what's called thread local storage, so like a temporary storage space for each thread. Any questions? Cool.

So on the quiz, you might be asked about various abusing of various other novel hyperobjects that don't do the typical thing, which is take a left and right and reduce them. They might do something a little bit quirky. And you might be asked to explain what they do.

AUDIENCE: Maybe?

PROFESSOR: Yes? Maybe. Maybe.

AUDIENCE: But this is purely for illustrating that you can do such a thing with hyperobjects.

Because you can always create a temporary variable there.

PROFESSOR: But suppose point was a much larger object. Sure, this is a contrived example. But yes. Anyone? Everyone gets this? So, in that case, why would that problem be more efficient than creating a new copy of it, say another point?

PROFESSOR: Because if you think about the case where this code runs on a single processor with no work stealing, this version does the exact same thing as the single threaded version. No extra objects are being created. And if you're running on however many processors and Cilk decides what to do in the background as far as how many threads to allocate for you, and your code doesn't have to know anything about that.

AUDIENCE: What's the monoid again?

PROFESSOR: The monoid is the associative operator that has an identity and a reduce operation and some other things that I forgot. Any more questions? Cool.

So switching gears, and this time I actually have a placeholder slide for it, next part is more about like the types of things that you might have learned from doing or experience you might have gained from doing your projects. And it primarily pertains to lab oriented, like in practice hands on things, as opposed to theoretical concepts.

So just as a fun warm up, everybody has hopefully at this point used some STL data structure. Vectors should definitely look familiar.

So let's compare some operations on these three. Because, as far as the API is concerned, both list-- all three of them support almost the same set of functions, as far as Insert, at pushback, pop front. All those operations are supported by these things.

So it's kind of-- You might ask why would you use one over the other? Well, it turns out that they have slightly different performance characteristics that are useful in noting. So anyone want to compare and contrast like what data structure a list corresponds to compared to a vector?

So which one would be like a linked list? List, yep. And then vector obviously is a dynamically growing array. And a deque-- how many people have used the deque? Cool. One.

How many people know what deque does even if they haven't used it? Three? Four? OK. So a deque is actually implemented similar to a vector, but it's dynamically growing on both ends.

OK, so let's start with filling in the table for the list. So inserting and removing at the end of a list. Constant time? Linear time? How many people say constant time? Linear? Half and half, wow. Don't rely on your neighbor during the test.

So it's actually constant time. If you think about appending to the end of a linked list, it's just a matter of couple pointer operations.

How about inserting and removing at the front? Constant? Cool. Linear? OK. I'll stop picking on people now.

So next is inserting and removing an arbitrary offset. So what I mean by that is you have an iterator to a particular point inside the list. Now I say insert this element five or n elements away from that iterator.

How long does that operation take? How many people say constant time? Three people. How many people say some sort of linear time?

I agree with you guys. It is indeed a linear time thing, because you have to traverse-- you have to actually fall the next pointers to the location or the previous pointers to the location that you want to insert. So although list might appear to support randomish axis, like you can add numbers to the iterator, it doesn't actually support that operation efficiently in the background.

Now the next one is also kind of tricky. Checking the size. Like checking how many elements are calling the dot size method of a standard list.

How many people think that's constant? Cool. About four or five. How many people

think that's linear? One.

How many people think it's possible to have a linked list data structure that's forced constant time? There we go. Well, unfortunately, the STL only requires you to have a big O of n time for this. In fact, I peeked inside the code for glibc the lib standard C++ that is used on the cloud machines. And in fact, what they do is they actually walk through everything from start to end, and they count up the number of times.

So in fact, it is a linear time operation, although I'm sure somewhere out in the world, there exists a C library, C++ library that does linear time, or constant time, size look ups. So you might want to be careful about that one.

On a different note, there's actually a dot empty method for checking whether or not a list is empty, and for any type of STL collection that is guaranteed to be constant time. So if you just want to know whether something's empty, don't use dot size equals 0.

What about reversing a linked list in place? Linear? More than linear? It's not a technical term, and Charles is probably going to hurt me now. Linear? Yes. Linear.

What about in place sorting of a linked list? How many people think it's N log(N)? One or two daring souls. Three daring souls. A well informed daring soul. How many people think it's more than N log(N)? One.

OK. So actually it is N log(N). And that's actually not specified by the specs. But it happens to be. And it turns out that there is a variant merge sort that works on link list that is N log(N). I didn't know that, but anyway.

So how about let's fill this in for the vector. Inserting and removing an element at the end-- constant time? Linear time? Nobody says linear time. So it's amortized constant time, I guess. I'll just call it constant time.

What about inserting or removing at the front of a vector-- constant time? Linear time? Come on, guys. I'm getting like three hands for linear time and zero for constant time. I guess that's reasonable.

So it does, in fact, insert one element. It puts the element in and then it grows the vector in. It copies everything in. So don't insert at the front, don't insert or remove at the front of a vector. It's a very bad idea if you do that a lot.

What about inserting or removing at an arbitrary location inside a vector? Well, that one's fairly obvious to be some sort of linear time too, because it has to shift all of the elements after it.

Checking the size of the vector? Some people are getting pessimistic about the STL. That's my personal opinion, but I won't transfer that onto you.

So it turns out that for a vector, yes, it is constant time to check the size. So that's OK.

In place reversal, linear time. And sorting is N log(N), which is not surprising by any means. So a deque? Inserting or removing at the end-- constant? Sure, constant. And inserting or removing at the front? Cool. Constant.

So that's the primary difference between a deque and a vector. And for everything else, it behaves exactly like a vector. Does that makes sense? Yes.

AUDIENCE: So is list like a doubly linked list?

PROFESSOR: A standard list is in fact a doubly linked list. So it supports both forward and reverse traversal. There's a standard S list, which is a singly linked list, and it only supports forward traversal. Any other questions?

Moving on to a slightly different topic, can anyone point out what is bad about this code? And I mean from a performance standpoint.

AUDIENCE: It's O of n.

PROFESSOR: All right. Someone said it, so it's O of n. So the vector that's passed in here is passed in by value, which means whenever you call this function, it'll actually make a brand new copy of the vector and copy all of the elements. So it's linear time with respect to the number of elements in the vector.

PROFESSOR: So how do you fix that?

AUDIENCE: You can pass in a pointer.

PROFESSOR: So there's two ways. So on the left is the original code, and on the right are two different ways to pass in just a pointer to the same thing. So you can do a reference pass, which is the first version above. And then you can use a dot to access the methods.

Or you can pass in by pointer which everyone is already familiar with. And that's where you have to use arrows to access the elements. And then you also have to change the way that you call the function. So oftentimes if you just catch something in code, like say the final project where something is passed by value, where it should be passed by reference, oftentimes the easiest way to modify the code is to just change it to use a reference rather than a pointer.

So the next thing that I want to cover is a question type that will be on the final, will be on the quiz. And I'll give some examples of problems of this form. But most importantly, I want everybody to be familiar with the form of the question, the rules involving the question, so nobody accidentally loses points for not reading the instructions, which nobody seems to do on quizzes.

So you are given two versions of the same function written in C or C++. And you are to assume that your compiler-- well, the compiler, as far as like what the CPU is executing is a literal interpretation of the code you see before and after.

By this point, everybody is probably familiar with some of the simple tricks a compiler does, like if you say int x equals 1 plus 1 there is no circumstance in which the compiler actually puts two 1s in two different registers and tells it to add it and then put it in x. So you are assuming that your compiler is doing a literal translation. And then you are asked to determine three properties about the optimization, so going from the before to the after.

So the first thing is whether or not the optimization is legal. And when we define

legal, we mean that the optimized version always has the same-- preserves the same behavior as the other version. It should be relatively simple to infer from the code snippets that we give you, what the intended purpose of a function is. And you are to determine whether or not the optimized version acts like the unoptimized version, as far as the correctness of the result.

Second, you are asked to determine whether or not this optimized version is faster than the previous version. And by faster, we mean faster in most conceivable cases. It's a short answer question so you might want to explain what you mean. But we do expect a yes or no answer to start off the question. And it should be pretty clear whether the answer is yes or no.

And then finally, I'm sure by this point everybody knows that when you compile with gcc -02 or -03, the compiler does a lot of things for you. So there's oftentimes no point in writing certain optimizations by hand, when you know when you hit the compile button, the compiler's going to do it in the background anyway.

So this question asks whether or not you can reasonably assume that the optimizing compilers, like the ones that you're using for your projects, would do this optimization for you or should you take the effort to actually do this optimization by hand. And then, legalese, if the optimization is illegal, let's not answer whether or not it's faster or automatic. And if the optimization actually makes things slower, then let's not answer whether the compiler does it for you.

Some of us have more cynical views of the compiler than others. So we'll just disregard that. Any question on the rules?

AUDIENCE: How are we supposed to know what types of things the compiler's going to do, can we just read the documentation?

PROFESSOR: So there was a lecture that Samaan gave called What Compilers Can and Cannot Do.

AUDIENCE: So it's just stuff from that lecture?

PROFESSOR: I would say most of it is that lecture. And then other things-- we might throw in a thing or two that you guys hopefully learn from. So throughout the term, we've been misguiding you on those walk through P-sets, suggesting you to do things that actually don't help. And you guys have gotten pretty pissed at us about that probably.

But there was a method to the madness. I mean, there was something to be learned from that, whether or not we actually meant to do it. So hopefully some level of intuition would answer it, and you would get the bulk of the points for just understanding the points raised in the lecture. Any other questions about the rules? All right.

So I have a couple of examples of these types of questions. So the first one is this. So it should be fairly obvious what the code does. It takes your input, and it wants to mod it by 256. So first question is this a legal optimization? How many people say yes? A lot of people. Anybody say no? Reed almost scared me for a moment there. He had his hand up.

So yes, it is indeed legal. And it's a standard bit hack for doing a modulo by a power of two.

AUDIENCE: What if it's 257?

PROFESSOR: I was fortunate enough to make it a uint64. So next question is given that it's legal, is it faster to do the and instead of the mod? How many people say yes? How many people say no? So a majority of the class says yes, and some people say no. So what would be a reason why you think it's not faster?

AUDIENCE: Well, I didn't say it's not faster faster. It's probably the same time, because I assume the instruction for modding is not actually dividing by it.

PROFESSOR: So for the other people who said no, how many people agree with that line of reasoning? All right. Seems like everybody. So unfortunately, it is faster. And it turns out, I compiled down both of these pieces of code under-- well, actually, the compiler-- but anyway, I'll get to that next. Not giving away any answers.

But so I executed the assembly instructions corresponding to both of these operations. And it does turn out that the compiler, the CPU, when you ask it to take the mod of an arbitrary number, usually it's an instruction called div mod. And what that does is that it does a divided by b, and it puts the dividend in one register and it puts the modulo in another register.

And we do that for, let's say, 2 to the 63rd modulo 256, because it actually goes through all the clock cycles for doing that division process. And then it dumps out the modulo at the end. So yes, this actually does tend to be a lot faster.

Now next question is will the compiler do this for you? How many people say yes? Most of the class. Anybody say no? Good. We all trust the compiler to some extent. So yeah, it turns out that gcc, starting from 01, will do this for you. Any questions? All right.

Next question. So you have a list. And you pushed in. You pushed at the end. It's 42, twice. And then you returned the result of popping the front element and then popping the back element, subtracted.

So first of all, is this a legal operation? Let me give everybody a chance to think.

AUDIENCE: Legal optimization as opposed to an operation.

PROFESSOR: Yeah. How many people vote yes, this is going legal? Roughly half the class. How many people say no, this is not legal? Two people. OK, so why would it not be legal? You can't do that. Yes?

AUDIENCE: There happens to be wastefulness on some problems.

PROFESSOR: Well, that could certainly happen. But for the purpose of these problems-- see this is why I'm bringing up these problems-- so for the purpose of these problems, assume there's no man behind the curtain doing things. For any given problem, you could say that's someone attaching a debugger to your program and changing the memory between instructions. So let's just assume that's happening.

AUDIENCE: Also, who's local, right?

PROFESSOR: Yeah. Yea, Foo is actually local, so if anybody else's doing anything to it, they're a very bad person. So fine. It's a legal optimization. Now. is it faster? How many people think that returning zero is slower than pushing two things up?

AUDIENCE: [LAUGHTER]

AUDIENCE: Depends upon how big zero is.

PROFESSOR: That's true. So OK, so yes, obviously, it's faster to just directly return zero. And finally, is this an automatic optimization? How many people say yes? Cool. Nobody said yes. How many people say no? People who say yes, if you could work for compiler companies, I would appreciate that a lot.

So unfortunately, the compiler does not make-- cannot reason through how an STL list works, and realize that what you're doing is an no op. It turns out that for a related set of questions, like have any of your mentors said anything to-- so my mentor said this to me last year. And I was pretty interested when I tried it out.

Have you ever used a for loop to set everything in an array to equals 0? Most people seem to have done that. Who has used the memset or heard of the memset set API call before? A good portion of the class. OK, so some people.

So a little bit of background then. Mem set instruction takes in an array or a pointer rather, and a constant, and the size of the array. And it sets that many elements of the array to that constant.

So it turns out that depending on the machine that you're using, there is more efficient and less efficient ways of setting all the elements of an array to 0. And functions like mem set are actually are several pages long of hand optimized assembly to be optimal for your architecture.

So it's always tempting to take a for loop that iterates through an array, setting everything to 0, and changing it to a single mem set instruction. And it turns out that for both gcc and icc, Intel C compiler, when you compile at sufficient optimization

level, the compiler actually does this for you. It actually looks through the arrays. It actually realizes that you're setting all the members in the array to a constant, and it does that optimization for you by replacing it with a memset call.

So just as a general question, why could it be advantageous for you to write the four loop as is, and have the compiler do the optimization for you?

AUDIENCE: Because you later might add code in the loop.

PROFESSOR: True. Any other reasons?

PROFESSOR: I didn't hear question.

PROFESSOR: You might want to add code that changes the behavior inside the loop.

AUDIENCE: It's much more readable to have a for loop and you just read it, and I get that's what he's doing.

PROFESSOR: Yes, indeed. Looking over some of your code submissions, it seems to be a non-trivial task of figuring out the size of something when you call malloc or memset. And actually, it sometimes is a non-trivial operation to figure out how big something is in bytes, especially for structs and things like that. So it actually is easier to let the compiler do the optimization for you, and you don't make any embarrassing mistakes in the process.

So next question. So here you have a helper function. And in the after version, seems to do the same thing, does it? So who thinks this is a legal optimization? Half the class. Anyone think it's illegal?

So yeah, it is legal. The only thing that I've done is that I've copy pasted basically the function body in the bottom example. And it's relatively easy to prove that or infer that the two do the same thing.

So is it faster? How many people say yes? How many people say no? One person said no. Why is it not faster?

AUDIENCE: I mean, I've said that it is the same speed, because so the only thing that would make it slower would be the-- you need to actually do like a compass switch. Because you need to call that function every time. But I think it could provide the concept for you-- like basically, it's a static function. It would inline it even though you don't have to declare it.

PROFESSOR: OK. So if you recall the rules for the question, so basically we want you to interpret the code, as is, like a very literal interpretation as in-- the point was raised that for this function call, what you actually want to do is assume that the compiler is setting up a function called by putting the arguments in the right register and jumping to the address of that function above. So for the case of this-- for the matter-- for the purpose of this question, yes, it's much faster to just do a multiply compared to setting up a call stack and tearing it down after the function is done.

Now the next question is is this automatic, which was more or less answered. And that's yes. So at a sufficient optimization level which is -03 or -02 with dash f inline functions, the compiler will actually use a heuristic that it has for determining whether or not a function is a good candidate for inlining. And for good candidates, it'll do the inlining for you.

Now the specific heuristics vary from compiler to compiler. But for this specific problem that we gave you, it's a prime candidate for inlining, because it's declared as static, which means it's local to this C file. And it does a very simple task, a one-liner, so it almost certainly will be inline for you. Any questions? Cool.

So this one. Question?

AUDIENCE: I don't get it. Could you explain again? I know I've asked this before, like what's static, the static function? And it's different than in Java?

PROFESSOR: OK. So the question was what's a static function in C. And so in C, a static function means when you declare a function as static, it means that the function is local to the current file that you're work-- the current module, the dot-o file, which is almost always your dot-c file. So like the function only exists within this dot-c file. A function

in another dot-c file cannot call this function.

So it basically allows the compiler to assume that there's no external effects on this function that are not within this file. Remember the compiler compiles object files one by one, and then the linker is the one that brings it all together.

AUDIENCE: So the important thing is that it knows that it can just delete helper after it's gotten rid of it.

PROFESSOR: Right, right. That's another good point. So in this case, after inlining, the compiler actually won't bother to put in the function helper inside the actual object file. So you save on my code space. Because the compiler knows that nowhere else this function can be called.

AUDIENCE: What about static in C++?

PROFESSOR: Static in C++ when used inside a class means that the function is a static member function. So it's a function that's tied to the class itself, rather than an object, and instance of the class.

AUDIENCE: Just like Java.

PROFESSOR: Just like Java. More questions? Cool. So back to this code.

First of all, is anyone not familiar with the syntax or wants the syntax explained? OK, cool. So first question, is it legal? How may people say yes? Half the class.

How many people say no. Roughly half the class. Well, I and the people grading the quiz will side with the people who say no, because down here when you're doing-- this is the correct version of the XOR swap. I'm not trying to trick anyone there.

But it turns out-- well, I'd rather not assume that t, the arbitrary type name, has an operator XOR defined for it that has all the properties that you need in order for it to do this. Like the simplest example is if I pass in a double, this would not work. Double does not have an operator XOR defined for it. Some people look upset.

AUDIENCE: [LAUGHTER]

PROFESSOR: A list too would not have an XOR defined for it. Anyone disagree with me? That's not a challenge. So yeah, some of these questions require you to do a little bit of reasoning, like reasonable reasoning, as far as what to assume for the purposes of the question. So watch out for that.

So in this case, it's not legal, so you don't have to answer the next two questions. So OK. So let's fix this. And by fix it, I mean not use templates. So now we're just operating on ints. And ints do have the proper operator XOR that has all the good properties that you want. Yes?

AUDIENCE: What's the and notation of the variable.

PROFESSOR: It's a reference. So it's a reference to it. Because you need references, because you actually want to swap the two items that are being called in. Right?

AUDIENCE: Can you use like pointeres?

PROFESSOR: Yes, so you can also use a pointer for it. Pointers and references, C will treat this use of references the same way as if you use pointers.

So references are syntactic sugar. And what it basically allows you to do is write code that looks like you're operating on standard intervals, standard integers, but in fact that it's a pointer to the value that the name of the variable that's passed in here. Otherwise this code would be riddled with a bunch of stars.

Now is this legal? How many people say yes? Most of the class. How many people say no? One person. Yes?

AUDIENCE: If A and B are the same variable, you set it to zero.

PROFESSOR: Yes, exactly. The XOR law has a nasty side effect where you want to make sure that A and B are not the same variable. Not they don't contain the same-- they can contain the same value in different variables, but they can't be the same variable or the same register or the same memory location. Otherwise, when you do A equals

Ax or B, that's 0. And now both A and B point to 0. And there's nothing you can do to get your old value back.

AUDIENCE: Can you check for equivalence of recordkeeping in C++.

PROFESSOR: Indeed, you can. If you take the address of the reference, it gives you the address of the original value. So this would be the proper way, I guess, of doing an XOR swap. So now that we've gotten over all the trick questions, is this actually faster? How many people say yes? One, two, three, four. How many people say no? Cool. Most of the class.

So this is another example of a gray area question. So in this class, we primarily talk about the cloud machines, as far as the performance analysis that we've done. So on the cloud machines, and this was mentioned in Charles's bit hacks lecture which was, well, the second lecture in the course. This trick is not faster on the cloud machines.

Because what you end up doing is you create a lot of dependencies between these operands, so that the pipeline must execute these one by one serially. So you introduce a lot of delays by that. And plus now that we actually want you to write a correct program, you have to do this check, and that's an extra branch.

And finally these XORs have to be done by the ALU, and up to five execution ports. Only three of them are ALUs. The other two can do memory operations just fine. So it turns out that for a combination of these reasons, this version indeed with the temporary register actually does run faster. Any questions? Yes.

AUDIENCE: For branches, we assume advanced features, like it will preload the exceptions and only invalidates the thing when certain branches fail.

PROFESSOR: Sure, you can assume cool properties about the chip, such as speculative execution, prefetching, and so on. But the problem is those tricks are fine when you're doing things sparsely as in you're sometimes doing work and other times not doing work.

But like speculative execution, like when you execute both sides of a branch, you're using more or less double your execution or resources. And if you have all processors running some intense piece of code that's doing swapping, you don't have any extra resources to dedicate towards running code that might not be useful. Any other questions? Cool.

So finally, I got tired of making slides. Well, actually that's not true. It's more I'd rather concentrate the slides on things that I think are more useful than other things, as far as per preparation for your quiz.

All your time is valuable. And so there's other topics that are fair game for the quiz, because they've been mentioned in lectures up to this point. But I haven't prepared any slides on them.

And the fractal trees were covered by the guest lecture by Bradley. And the slides are up. I suggest familiarizing yourself with the basic workings of how the data structure that he described works. And lock free data structures were covered on your take home P set, and also in a lecture.

So be prepared to answer potential questions about that. And then finally the mechanism behind the Cilk runtime, the work stealing scheduler, how it's implemented on the Linux side, and also the basics of what a Cilk hyperobject is, what a Cilk hyperobject has to have. Advantages and disadvantages of using them and so on. Yes.

AUDIENCE: Are the grades for the P-set out?

PROFESSOR: No, they are not out yet.

AUDIENCE: Are there solutions for the P-set?

PROFESSOR: Not yet. But I believe we have something prepared for that, right? Somewhat? Yeah, it's somewhat prepared. If you have any questions about like questions on the pset that you want answer to, feel free to stop by office hours or email us and we can help you.

Yeah, apologies for not getting that P set back to you in time. Any other questions relating to the quiz. It's really all the material I have as far as reviewing. Yes?

AUDIENCE: Are you going to put these slides up?

AUDIENCE: Yeah, I'll put these slides up right after the lecture. So as far as a general notes, I guess you guys turned in your finals yesterday, and you guys have a design report that you also turned in. Your POSSE members should have contacted you by now, as far as scheduling a design review session. Since this project is actually due on next Thursday, it might be a good idea to get this done by the end of the week. Otherwise, the design review won't be a very useful process.

AUDIENCE: There are a couple of students without a mentor right now here. We are fixing that.

PROFESSOR: Right. There's a couple of students caught in limbo right now. Sorry about that.

PROFESSOR: If you were-- previously dropped on Stellar, send us an email. If you haven't been to drop through this course, there are no longer drops allowed. Yeah,

AUDIENCE: So it's your responsibility. I'm going to put on my mean hat. It's your responsibility to schedule a meeting with your POSSE member, and make sure that it happens by the end of the week. And if it doesn't look like it's going to happen, or you're having difficulties getting in touch with your mentor, or you don't have one yet, please drop us an email or come by office hours and make sure that we know. All right. That's it.