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

PROFESSOR: Hey, everybody. It's my pleasure once again to welcome TB Schardl, who is the author of your taper compiler, to talk about the Cilk runtime system.

TAO SCHARDL: Thanks, Charles. Can anyone hear me in the back, seem good? OK. Thanks for the introduction. Today I'll be talking about the Cilk runtime system. This is pretty exciting for me. This is a lecture that's not about compilers. I get to talk about something a little different for once. It should be a fun lecture. Recently, as I understand it, you've been looking at storage allocation, both in the serial case as well as the parallel case.

And you've already done Cilk programming for a while, at this point. This lecture, honestly, is a bit of a non sequitur in terms of the overall flow of the course. And it's also an advanced topic. The Cilk runtime system is a pretty complicated piece of software. But nevertheless, I believe you should have enough background to at least start to understand and appreciate some of the aspects of the design of the Cilk runtime system.

So that's why we're talking about that today. Just to quickly recall something that you're all, I'm sure, intimately familiar with by this point, what's Cilk programming all about? Well, Cilk is a parallel programming language that allows you to make your software run faster using parallel processors. And to use Cilk, it's pretty straightforward. You may start with some serial code that runs in some running time-- we'll denote that as Ts for certain parts of the lecture.

If you wanted to run in parallel using Cilk, you just insert Cilk keywords in choice locations. For example, you can parallelize the outer loop in this matrix multiply kernel, and that will let your code run in time Tp on P processors. And ideally, Tp should be less than Ts. Now, just adding keywords is all you need to do to tell Cilk to execute the computation in parallel. What does Cilk do in light of those keywords?

At a very high level, Cilk and specifically its runtime system takes care of the task of scheduling and load balancing the computation on the parallel processors and on the multicore system in general. So after you've denoted logical parallel in the program using spawn, Cilk spawn, Cilk

sync, and Cilk four, the Cilk scheduler maps that computation onto the processors.

And it does so dynamically at runtime, based on whatever processing resources happen to be available, and still uses a randomized work stealing scheduler which guarantees that that mapping is efficient and the execution runs efficiently. Now you've all been using the Cilk platform for a while. In its basic usage, you write some Cilk code, possibly by parallelizing ordinary serial code, you feed that to a compiler, you get a binary, you run the binary the binary with some particular input on a multicore system.

You get parallel performance. Today, we're going to look at how exactly does Cilk work? What's the magic that goes on, hidden by the boxes on this diagram? And the very first thing to note is that this picture is a little bit-- the first simplification that we're going to break is that it's not really just Cilk source and the Cilk compiler. There's also a runtime system library, libcilkrts.so, in case you've seen that file or messages about that file on your system.

And really it's the compiler and the runtime library, that work together to implement Cilk's runtime system, to do the work stealing and do the efficient scheduling and load balancing. Now we might suspect that if you just take a look at the code that you get when you compile a Cilk program, that might tell you something about how Cilk works. Here's C pseudocode for the results when you compile a simple piece of Cilk code. It's a bit complicated. I think that's fair to say.

There's a lot going on here. There is one function in the original program, now there are two. There's some new variables, there's some calls to functions that look a little bit strange, there's a lot going on in the compiled results. This isn't exactly easy to interpret or understand, and this doesn't even bring into the picture the runtime system library. The runtime system library, you can find the source code online. It's a little less than 20,000 lines of code. It's also kind of complicated.

So rather than dive into the code directly, what we're going to do today is an attempt at a top-down approach to understanding how the Cilk runtime system works, and some of the design considerations. So we're going to start by talking about some of the required functionality that we need out of the Cilk runtime system, as well as some performance considerations for how the runtime system should work.

And then we'll take a look at how the worker deques in Cilk get implemented, how spawning

actually works, how stealing a computation works, and how synchronization works within Cilk. That all sound good? Any questions so far? This should all be review, more or less. OK, so let's talk a little bit about required functionality. You've seen this picture before, I hope. This picture illustrated the execution model of a Cilk program. Here we have everyone's favorite exponential time Fibonacci routine, parallelized using Cilk.

This is not an efficient way to compute Fibonacci numbers, but it's a nice didactic example for understanding parallel computation, especially the Cilk model. And as we saw many lectures ago, when you run this program on a given input, the execution of the program can be modeled as a computation dag. And this computation dag unfolds dynamically as the program executes.

But I want to stop and take a hard look at exactly what that dynamic execution looks like when we've got parallel processors and work stealing all coming into play. So we'll stick with this Fibonacci routine, and we'll imagine we've just got one processor on the system, to start. And we're just going to use this one processor to execute fib(4). And it's going to take some time to do it, just to make the story interesting.

So we start executing this computation, and that one processor is just going to execute the Fibonacci routine from beginning up to the Cilk spawn statement, as if it's ordinary serial code, because it is ordinary serial code. At this point the processor hits the Cilk spawn statement. What happens now? Anyone remember? What happens to the dag?

AUDIENCE: It branches down [INAUDIBLE]

TAO SCHARDL: It branches downward and spawns another process, more or less. The way we model that-- the Cilk spawn is of a routine fib of n minus 1. In this case, that'll be fib(3). And so, like an ordinary function call, we're going to get a brand new frame for fib(3). And that's going to have some strand that's available to execute. But the spawn is not your typical function call. It actually allows some other computation to run in parallel.

And so the way we model that in this picture is that we get a new frame for fib(3). There's a strand available to execute there. And the continuation, the green strand, is now available in the frame fib(4). But no one's necessarily executing it. It's just kind of faded in the picture. So once the spawn has occurred, what's the processor going to do? The processor is actually going to dive in and start executing fib(3), as if it were an ordinary function call.

Yes, there's a strand available within the frame of fib(4), but the processor isn't going to worry about that strand. It's just going to say, oh, fib(4) calls fib(3), going to start computing for fib(3). Sound good? And so the processor dives down from pink strand to pink strand. The instruction pointer for the processor returns to the beginning of the fib routine, because we're now calling fib once again. And this process repeats.

It executes the pink strand up until the Cilk spawn, just like ordinary serial code. The spawn occurs-- and we've already seen this picture before-- the spawn allows another strand to execute in parallel. But it also creates a frame for fib(2). And the processor dives into fib(2), resetting the instruction pointer to the beginning fib, P1 executes up to the spawn. Once again, we get another string to execute, as well as an invocation of fib(1). Processor dives even further.

So that's fine. This is just the processor doing more or less ordinary serial execution of this fib routine, but it's also allowing some strands to be executed in parallel. This is the one processor situation, looks pretty good so far. Right, and in the fib(1) case, it doesn't make it as far through the pink strand because, in fact, we hit the base case. But now let's bring in some more processors. Suppose that another processor finally shows up, says I'm bored, I want to do some work, and decides to steal some computation.

It's going to discover the green strand in the frame fib(4), and P2 is just going to jump in there and start executing that strand. And if we think really hard about what this means, P2 is another processor on the system. It has its own set of registers. It has its own instruction pointer. And so what Cilk somehow allows to happen is for P2 to just jump right into the middle of this fib(4) routine, which is already executing. It just sets the instruction pointer to point at that green instruction, at the call to fib of n minus 2.

And it's just going to pick up where processor 1 left off, when it executed up to this point in fib(4), somehow. In this case, it executes fib of n minus 2. That calls fib(2), creates a new strand, it's just an ordinary function call. It's going to descend into that new frame. It's going to return to the beginning of fib. All that's well and good. Another processor might come along and steal another piece of the computation. It steals another green strand, and so once again, this processor needs to jump into the middle of an executing function.

Its instruction pointer is just going to point at this call of the fib of n minus 2. Somehow, it's going to have the state of this executing function available, despite having independent

registers. And it needs to just start from this location, with all the parameters set appropriately, and start executing this function as if it's an ordinary function. It calls fib(3) minus 2 is 1. And now these processors might start executing in parallel.

P1 might return from its base case routine up to the parent call of fib of n minus 2 and start executing its continuation, because that wasn't stolen. Meanwhile, P3 descends into the execution of fib(1). And then in another step, P3 and P2 make some progress executing their computation. P2 encounters a Cilk spawn statement, which creates a new frame and allows another strand to execute in parallel. P3 encounters the base case routine and says, OK, it's time to return.

And all of that can happen in parallel, and somehow the Cilk system has to coordinate all of this. But we already have one mystery. How does a processor start executing from the middle of a running function? The running function and it's state lived on P1 initially, and then P2 and P3 somehow find that state, hop into the middle of the function, and just start running. That's kind of strange. How does that happen? How does the Cilk runtime system make that happen? This is one thing to consider.

Another thing to consider is what happens when we hit a sync. We'll talk about how these issues get addressed later on, but let's lay out all of the considerations upfront, before we-- just see how bad the problem is before we try to solve it bit by bit. So now, let's take this picture again and progress it a little bit further. Let's suppose that processor three decides to execute the return. It's going to return to an invocation of fib(3). And the return statement is a Cilk sync statement.

But processor three can't execute the sync because the computation of fib(2) in this case-- that's being done by processor one-- that computation is not done yet. So the execution can proceed past the sync. So somehow P3 needs to say, OK, there is a sync statement, but we can't execute beyond this point because, specifically, it's waiting on processor one. It doesn't care what processor two is doing. Processor two is having a dandy time executing fib(2) on the other side of the tree. Processor three shouldn't care.

So processor three can't do something like, OK, all processors need to stop, get to this point in the code, and then the execution can proceed. No, no, it just needs to wait on processor one. Somehow the Cilk system has to allow that fine grain synchronization to happen in this nested pattern. So how does a Cilk sync wait on only the nested sub computations within the

program? How does it figure out how to do that? How does the Cilk runtime system implement this?

So that's another consideration. OK, so at this point, we have three top level considerations. A single worker needs to be able to execute this program as if it's an ordinary serial program. Thieves have to be able to jump into the middle of executing functions and pick up from where they left off, from where other processors in the system left off. Syncs have to be able to stall functions appropriately, based only on those functions' nested child sub computations.

So we have three big considerations that we need to pick apart so far. That's not the whole story, though. Any ideas what other functionality we need to worry about, for implementing this Cilk system? It's kind of an open ended question, but any thoughts? We have serial execution, spawning, stealing, and syncing as top level concerns. Anyone remember some other features of Cilk that the runtime system magically makes happen, correctly? It's probably been a while since you've seen those. Yeah.

AUDIENCE: Cilk for loops divide and conquer?

TAO SCHARDL: The Cilk for loops divide and conquer. Somehow, the runtime system does have to implement Cilk fours. The Cilk fours end up getting implemented internally, with spawns and syncs. That's courtesy of the compiler. Yeah, courtesy of the compiler. So we wont look too hard at Cilk fors today, but that's definitely one concern. Good observation. Any other thoughts, sort of low level system details that Cilk needs to implement correctly?

Cache coherence-- it actually doesn't need to worry too much about cache coherence although, given the latest performance numbers I've seen from Cilk, maybe it should worry more about the cache. But it turns out the hardware does a pretty good job maintaining the cache coherence protocol itself. But good guess . It's not really a tough question, because it's really just calling back memories of old lectures.

I think you recently had a quiz on this material, so it's probably safe to say that all that material has been paged out of your brain at this point. So I'll just spoil the fun for you. Cilk has a notion of a cactus stack. So we talked a little bit about processors jumping into the middle of an executing function and somehow having the state of that function available. One consideration is registered state, but another consideration is the stack itself.

And Cilk supports the C's rule for pointers, namely that children can see pointers into parent

frames, but parents can't see pointers into child frames. Now each processor, each worker in a Cilk system, needs to have its own view of the stack. But those views aren't necessarily independent. In this picture, all five processors share the same view of the frame for Function A instantiation A, then processors three through five all share the same view for the instantiation of C.

So somehow, Cilk has to make all of those views available and consistent but not quite the same, sort of consistent as we get with cache coherence. Cilk somehow has to implement this cactus stack. So that's another consideration that we have to worry about. And then there's one more kind of funny detail. If we take another look at work stealing itself-- you may remember we had this picture from several lectures ago where we have processors on the system, each maintains its own deck of frames, and workers are allowed to steal frames from each other.

But if we take a look at how this all unfolds, yes we may have a processor that performs a call, and that'll push another frame for a called function onto its deque on the bottom. It may spawn, and that'll push a spawn frame onto the bottom of its deck. But if we fast forward a little bit and we get in up with a worker with nothing to do, that worker is going to go ahead and steal, picking another worker in the system at random. And it's going to steal from the top of the deque.

But it's not just going to steal the topmost item on the deque. It's actually going to steal a chunk of items from the deque. In particular, if it selects the third processor in this picture, third from the left, this thief is going to steal everything through the parent of the next spawned frame. It needs to take this whole stack of frames, and it's not clear a priori how many frames the worker is going to have to steal in this case. But nevertheless, it needs to take all those frames and resume execution.

After all, that bottom was a call frame that it just stole. That's where there's a continuation with work available to be done in parallel. And so, if we think about it, there are a lot of questions that arise. What's involved in stealing frames? What synchronization does this system have to implement? What happens to the stack? It looks like we just shifted some frames from one processor to another, but the first processor, the victim, still needs access to the data in that stack. So how does that part work, and how does any of this actually become efficient?

So now we have a pretty decent list of functionality that we need out of the Cilk runtime

system. We need serial execution to work. We need thieves to be able to jump into the middle of running functions. We need sinks to synchronize in this nested, fine grain way. We need to implement a cactus stack for all the workers to see. And these have to deal with mixtures of spawned frames and called frames that may be available when they steal a computation.

So that's a bunch of considerations. Is this the whole picture? Well, there's a little bit more to it than that. So before I give you an answers, I'm just going to keep raising questions. And now I want to raise some questions concerning the performance of the system. How do we want to design the system to get good parallel execution times?

Well if we take a look at the work stealing bounds for Cilk, the Cilk's work stealing scheduler achieves an expected running time of Tp, on P processors, which is proportional to the work of the computation divided by the number of processors, plus something on the order of the span of the computation. Now if we take a look at this running time bound, we can decompose it into two pieces. The T1 over P part, that's really the time that the parallel workers on the system spend doing actual work.

They're P of those workers, they're all making progress on the work of the computation. That comes out to T of one over P. The other part of the bound, order T infinity, that's a time that turns out to be the time that workers spend stealing computation from each other. And ideally, what we want when we paralyze a program using Cilk, is we want to see this program achieve linear speedup. That means that if we give the program more processors to run, if we increase P, we want to see the execution time decrease, linearly, with P.

And that means we want the of the workers in the Cilk system to spend most of the time doing useful work. We don't want the workers spending a lot of time stealing from each other. In fact, we want even more than this. We don't just want work divided by number of processors. We really care about how the performance compares to the running time of the original serial code that we were given, that we parallelized.

That original serial code ran in time Ts of S. And now we paralyze it using Cilk spawn, Cilk sync, or in this case, Cilk for. And ideally, with sufficient parallelism, we'll guarantee that the running time is going to be Ts of P proportional to the work of a processor, T1 divided by P. But we really want to speed up compared to Ts of S. So that's our goal. We want Tp to be proportional to Ts of S over P. That says that we want the serial running time to be pretty close to the work of the parallel computation.

So the one processor running time of our Cilk code, ideally, should look pretty close to the running time of the original serial code. So just to put these pieces together, if we were originally given a serial program that ran on time Ts of S, and we parallelize it using Cilk, we end up with a parallel program with work T1 and span T infinity. We want to achieve linear speed up on P processors, compared to the original serial running time.

In order to do that, we need two things. We need ample parallelism. T1 one over T infinity should be a lot bigger than P. And we've seen why that's the case in lectures past. We also want what's called high work efficiency. We want the ratio of the serial running time divided by the work of the still computation to be pretty close to one, as close as possible. Now, the Cilk runtime system is designed with these two observations in mind.

And in particular, the Cilk runtime system says, suppose that we have a Cilk program that has ample parallelism. It has efficient parallelism to make good use of the available parallel processors. Then in implementing the Cilk runtime, we have a goal to maintain high work efficiency. And to maintain high work efficiency, the Cilk runtime system abides by what's called the work first principle, which is to optimize the ordinary serial execution of the program, even at the expense of some additional cost to steals.

Now at 30,000 feet, the way that the Cilk runtime system implements the work first principle and makes all these components work is by dividing the job between both the compiler and the runtime system library. The compiler uses a handful of small data structures, including workers and stack frames, and implements optimized fast paths for execution of functions, which should be executed when no steals occur.

The runtime system library handles issues with the parallel execution. And uses larger data structures that maintain parallel running time state. And it handles slower paths of execution, in particular when seals actually occur. So those are all the considerations. We have a lot of functionality requirements and we have some performance considerations. We want to optimize the work, even at the expense of some steals. Let's finally take a look at how Cilk works. How do we deal with all these problems?

I imagine some you may have some ideas as to how you might tackle one issue or another, but let's see what really happens. Let's start from the beginning. How do we implement a worker deque? Now for this discussion, we're going to use a running example with just a really, really simple, Cilk routine. It's not even as complicated as fib. We're going to have a

function foo that, at one point, spawns a function bar, in the continuation calls baz, performs a sync, and then returns.

And just to establish some terminology, foo will be what we call a spawning function, meaning that foo is capable of executing a Cilk spawn statement. The function bar is spawned by foo. We can see that from the Cilk spawn in front of bar. And the call to baz occurs in the continuation of that Cilk spawn, simple picture. Everyone good so far? Any questions about the functionality requirements, terminology, performance considerations? OK.

So now we're going to take a hard look at just one worker and we're going to say, conceptually, we have this deque-like structure which has spawned frames and called frames. Let's ignore the rest of the workers on the system. Let's not worry about-- well, we'll worry a little bit about how steals can work, but we're just going to focus on the actions that one worker performs. How do we implement this deque?

And we want the worker to operate on its own deck, a lot like a stack. It's going to push and pop frames from the bottom up the deque. Steals need to be able to transfer ownership of several consecutive frames from the top of the deque. And thieves need to be able to resume a continuation. So the way that the Cilk system does this, to bring this concept into an implementation, is that it's going to implement the deque externally from the actual call stack.

Those frames will still be in a stack somewhere and they'll be managed, roughly speaking, with a standard calling convention. But the worker is going to maintain a separate deque data structure, which will contain pointers into this stack. And the worker itself will maintain the deque using head and tail pointers. Now in addition to this picture, the frames that are available to be stolen-- the frames that have computation that a thief can come along and execute-- those frames will store an additional local structure that will contain information as necessary for stealing to occur.

Does this make sense? Questions so far? Ordinary call stack, deque lives outside of it, worker points at the deque, pretty simple design. So I mentioned that the compiler used relatively lightweight structures. This is essentially one of them. And if we take a look at the implementation of the Cilk runtime system, this is the essence of it. There are some additional implementation details, but these are the core-- this is, in a sense, the core piece of the design.

So the rest is just details. The Intel Cilk Plus runtime system takes this design and elaborates

on it in a variety of ways. And we're going to take a look at those elaborations. First off, what we'll see is that every spawned subcomputation ends up being executed within its own helper function, which the compiler will generate. That's called a spawn helper function. And then the runtime system is going to maintain a few basic data structures as the workers execute their work.

There'll be a structure for the worker, which will look similar to what we just saw in the previous slide. There'll be a Cilk stack frame structure for each instantiation of a spawning function, some function that can perform and spawn. And there'll be a stack-frame structure for each spawn helper, each instantiation that is spawned. Now if we take another look at the compiled code we had before, some of it starts to make some sense.

Originally, we had our spawning function foo and a statement that spawned off, called a bar. And in the C pseudocode of the compiled results, we see that we have two functions. The first function foo-- that's our spawning function-- it's got a bunch of stuff in it, and we'll figure out what that's doing in a second. But there's a second function, and that second function is the spawn helper. And that spawn helper actually contains a statement which calls bar and ultimately saves the result.

Make sense? Now we're starting to understand some of the confusing C pseudocode we saw before. And if we take a look at each of these routines we see, indeed, there is a stack frame structure. And so in Intel Cilk Plus it's called a Cilk RTS stack frame, very creative name, I know. And it's just added as an extra local variable in each of these functions. You got one inside of foo, because that's a spawning function, and you get one inside of the spawn helper.

Now if we dive into the Cilk stack frame structure itself, by cracking open the source code for the Intel Cilk Plus runtime, we see that there are a lot of fields in the structure. The main fields are as follows-- there is a buffer, a context buffer, and that's going to contain enough information to resume a function at a continuation, particularly to mean after a Cilk spawn or, in fact, after a Cilk sync statement.

There's an additional integer in the stack frame called flags, which will summarize the state of the Cilk stack rate, and we'll see a little bit more about that later. And there's going to be a pointer to a parent Cilk stack frame that's somewhere above this Cilk RTS stack frame, somewhere in the call stack. So these Cilk RTS stack frames, these are the extra bit of state that the Cilk runtime system adds to the ordinary call stack. So if we take a look at the actual

worker structure, it's a lot like what we saw before.

We have a deque that's external to the call stack. The Cilk worker maintains head and tail pointers to the deque. The Cilk workers are also going to maintain a pointer to the current Cilk RTS stack frame, which will tend to be somewhere near the bottom of the stack. OK, so those are the basic data structures that a single worker is going to maintain. That includes the deque. Let's see them all in action, shall we? Any questions about that so far, before we start watching pointers fly? Yeah.

AUDIENCE: I guess with the previous slide, there were arrows on the workers' call stack. What do you [INAUDIBLE]?

TAO SCHARDL: What do the arrows among the elements on the call stack mean? So in this picture of the call stack, function instantiations are actually in green, and local variables-- specifically the Cilk RTS stack frames-- those show up in beige. So foo SF is the Cilk RTS stack frame inside the instantiation of foo. It's just a local variable that's also stored in the stack, right? Now, the Cilk RTS stack frame maintains a parent pointer, and it maintains a pointer up to some Cilk RTS stack frame above it on the stack.

It's just another local variable, also stored in the stack. So when we step away and look at the whole call stack with all the function frames and the Cilk RTS stack frames, that's where we get the pointers climbing up the stack. We're good? Other questions? All right, let's make some pointers fly. OK, this is going to be kind of a letdown, because the first thing we're going to look at is some code. So we're not going to have pointers flying just yet.

We can take a look at the code for the spawning function foo, at this point. And there's a lot of extra code in here, clearly. I've highlighted a lot of stuff on this slide, and all the highlighted material is related to the execution of the Cilk runtime system. But basically, if we look at this code, we can understand each of these pieces. Each of them has some role to play in making the Cilk runtime system work.

So at the very beginning, we have our Cilk stack frame structure. And there's a call to this enter frame function, which all that really does is initialize the stack frame. That's all the function is doing. Later on, we find that there's this set jump routine-- we'll talk a lot more about set jump in a bit-- that, at this point, we can say the set jump prepares the function for a spawn. And inside the conditional, where the set jump occurs as a predicate, we have a call to spawn bar.

If we remember from a couple of slides ago, spawn bar was our spawn helper function. So we're here, we're just invoking the spawn helper. Later on in the code, we have another blob of conditionals with a Cilk RTS sync call, deep inside. All that code performs a sync. We'll talk about that a bit near the end of lecture. And finally, at the end of the spawning function, we have a call to pop frame, which just cleans up the Cilk stack frame structure within this function. And then there's a call to leave frame, which essentially cleans up the deque.

That's the spawning function. This is the spawn helper. It looks somewhat similar. I've added extra whitespace just to make the slide a little bit prettier. And in some ways, it's similar to the spawning function itself. We have a Cilk RTS stack frame [INAUDIBLE] spawn helper, another call to enter frame, which is just a little bit different. But essentially, it initializes the stack frame. Its reason to be is similar to the enter frame call we saw before.

There's a call to Cilk RTS detach, which performs a bunch of updates on the deque. Then there is the actual invocation of the spawn subroutine. This is where we're calling bar. And finally, at the end of the function, there is a call to pop frame, to clean up the stack structure, and a call to leave frame, which will clean up the deck and possibly return. It'll try to return. We'll see more about that. So let's watch all of this in action. Question? OK, cool.

Let's see all of this in action. We'll start off with a pretty boring picture. All we've got on our call stack is main, and our Cilk worker has nothing on its deque. But now we suppose that main calls our responding function foo, and the spawning function foo contains a Cilk RTS stack frame. What we're going to do in the Cilk worker, what that enter frame call is going to perform, all it's going to do is update the current stack frame. We now have a Cilk RTS stack frame, make sure the worker points at it, that's all.

Fast forward a little bit, and foo encounters this call to Cilk spawn a bar. And in the C pseudocode that's compiled for foo, we have a set jump routine. This set jump is kind of a magical function. This is the function that allows thieves to steal the continuation. And in particular, the set jump takes, as an argument, a buffer. In this case, it's the context buffer that we have in the Cilk RTS stack frame. And what the set jump will do is it will store information that's necessary to resume the function at the location of the set jump.

And it stores that information into the buffer. Can anyone guess what that information might be?

AUDIENCE: The instruction points at [INAUDIBLE].

TAO SCHARDL: Instruction pointer or stock pointer, I believe both of those are in the frame. Yeah, both of those are in the frame. Good, what else?

AUDIENCE: All the registers are in use.

TAO SCHARDL: All the registers are currently in use. Does it need all the registers? You're absolutely on the right track, but is there any way it could restrict the set of registers it needs to save?

AUDIENCE: The registers are used later in the execution.

TAO SCHARDL: That's part of it. Set jump isn't that clever though, so it just stores a predetermined set of registers. But there is another way to restrict the set.

AUDIENCE: [INAUDIBLE]

TAO SCHARDL: Only registers uses parameters in the called function, yeah, close enough. Callee-saved registers. So registers that the function might-- that it's the responsibility of foo to save, this goes all the way back to that discussion in lecture, I don't remember which small number, talking about the calling convention. These registers need to be saved, as well as the instruction pointer and various stack pointers. Those are what gets saved into the buffer.

The other registers, well, we're about to call a function, it's up to that other function to save the registers appropriately. So we don't need to worry about those. So all good? Any questions about that? All right, so this set jump routine, let's take it for granted that when we call a set jump on this given buffer, it returns zero. That's a good lie for now. We'll just run with it. So set jump returs zero.

The condition says, if not zero-- which turns out to be true-- and so the next thing that happens is this call to the spawn helper, spawn_bar, in this case. When we call spawn_bar, what happens to our stack? So this should look pretty routine. We're doing a function call, and so we push the frame for the called function onto the stack. And that called function, spawn bar, contains a local variable, which is this [INAUDIBLE] stack frame. So that also gets pushed onto the stack, pretty straightforward.

We've seen function calls many times before. This should look pretty familiar. Now we do this Cilk RTS enter frame fast routine. And I mentioned before that that's going to update the

worker structure. So what's going to happen here? Well, we have a brand new Cilk RTS stack frame on the stack. Any guesses as to what change we make? What would enter frame do?

AUDIENCE: [INAUDIBLE]

TAO SCHARDL: Point current stack frame to spawn in bar stack frame, you're right. Anything else? Hope I got this animation right. What are the various fields within the stack frame? And what did-- sorry, I don't know your name. What's your name?

AUDIENCE: I'm Greg.

TAO SCHARDL: Greg, what did Greg ask about before, when we saw an earlier picture of the call stack?

AUDIENCE: Set a pointer to the parent.

TAO SCHARDL: Set a pointer to the parent, exactly. So what we're going to do is we're going to take this call stack, we'll do the enter frame fast routine. That establishes this parent pointer in our brand new stack frame. And we update the worker's current stack frame to point at the bottom. Yeah, question?

AUDIENCE: How does enter frame know what the parent is?

TAO SCHARDL: How does enter frame know what the parent is? Good question. Enter frame knows the worker. Or rather, enter frame can do a call, which will give it access to the Cilk worker structure. And because it can do a call, it can read the current stack frame pointer in the worker.

AUDIENCE: So we do [INAUDIBLE] before we change the current [INAUDIBLE]?

TAO SCHARDL: Yeah, in this case we do. So we add the parent pointer, then we delete and update. So, good catch. Any other questions? Cool. All right, now we encounter this thing, Cilk RTS detach. This one's kind of exciting. Finally we get to do something to the deque. Any guesses what we do? How do we update the deque? Here's a hint. Cilk RTS detach allows-- this is the function that allows some computation to be stolen.

Once Cilk RTS detach is done executing, a thief could come along and steal the continuation of the Cilk spawn. So what would Cilk RTS detach do to our worker and its structures? Yeah, in the back.

AUDIENCE: Push the stack frame to the worker deque?

TAO SCHARDL: Push the stack frame to the worker deque, specifically at the tail. Right, I gave it away by clicking the animation, oh well. Now the thing that's available to be stolen is inside of foo. So what ends up getting pushed onto the deque is not the current stack frame, but in fact its immediate parent, so the stack frame of foo. That gets pushed onto the tail of the deque. And we now push something onto the tail of a deque. And so we advance the tail pointer. Still good, everyone? I see some nods.

I see at least one nod. I'll take it. But feel free to ask questions, of course. And then of course there is this invocation of bar. This does what you might expect. It calls bar, no magic here. Well, no new magic here. OK, fast forward, let's suppose that bar finally returns.

And now we return to the statement after bar in the spawn helper. That statement is the pop frame. Actually, since we just returned from bar, we need to get rid of bar from the stack frame. Good, now we can execute the pop frame.

What would the pop frame do? It's going to clean up the stack frame structure. So what would that entail, any guesses?

AUDIENCE: I guess it would move the current stack frame back to the parent stack frame?

TAO SCHARDL: Move the current stack frame back to the parent, very good. I think that's largely it. I guess there's one other thing it can do. It's kind of optional, given that it's going to garbage the memory anyway. So it updates the current stack frame to point to the parent, and now it no longer needs that parent pointer. So it can clean that up, in principle. And then there's this call to Cilk RTS leave frame. This is magic-- well, not really, but it's not obvious. This is a function call that may or may not return.

Welcome to the Cilk runtime system. You end up with calls to functions that you may never return from. This happens all the time. And the Cilk RTS leave frame may or may not return, based entirely on what's on the status of the deque, what content is currently sitting on the workers' deque. Anyone have a guess as to why the leave frame routine might not return, in the conventional sense?

AUDIENCE: There's nothing else for the worker to do, so it'll sit there spinning.

TAO SCHARDL: If there's nothing left to do on the deck, then it's going to-- sorry, say again?

AUDIENCE: It'll just wait until there's work you can steal?

TAO SCHARDL: Right, if there's nothing on the deque, then it has nowhere to return to. And so naturally, as we've seen from Cilk workers in the past, it discovers there's nothing on the deque, there's no work to do, time to turn to a life of crime, and try to steal work from someone else. So there are two possible scenarios. The pop could succeed and execution continues as normal, or it fails and it becomes a thief. Now which of these two cases do you think is more important for the runtime system to optimize?

Success, case one, exactly, so why is that?

AUDIENCE: [INAUDIBLE]

TAO SCHARDL: At least, we hope so, yeah. We assume-- this hearkens all the way back to that work first principle-- we assume that in the common case, workers are doing useful work, they're not just spending their time stealing from each other. And therefore, ideally, we want to assume that the worker will do what's normal, just an ordinary serial execution. In a normal serial execution, there is something on the deque, the pop succeeds, that's case one.

So what we'll see is that the runtime system, in fact, does a little bit of optimization on case one. Let's talk about something a little more exciting. How about stealing computation. We like stealing stuff from each other. Yes?

AUDIENCE: [INAUDIBLE]

TAO SCHARDL: Where does it return the results? So where does it return the result in the spawn bar? The answer you can kind of see two lines above this. So in this case, in the original Cilk code, we had X equals Cilk spawn of bar. And here, what are the parameters to our spawn bar function? X and N. Now N is the input to bar, right? So what's X?

AUDIENCE: [INAUDIBLE]

TAO SCHARDL: You can rewind a little bit and see that you are correct. There we go. Yeah, so the original Cilk code, we had X equals Cilk spawn bar. That's the same X. All that Cilk does is pass a pointer to the memory allocated for that variable down to the spawn helper. And now the spawn helper, when it calls bar and that returns, it gets stored into that storage in the parent stack frame. Good catch. Good observation. Any questions about that? Does that make sense? Cool.

Probably used too many animations in these slides. All right, now let's talk about stealing. How does a worker steal computation? Now the conceptual diagram we had before saw this one worker, with nothing on its deque, take a couple of frames from another workers deque and just slide them on over. What does that actually look like in the implementation?

Well, we're still going to take from the top of the deque, but now we have a picture that's a little bit more accurate in terms of the structures that are really implemented in the system. So we have the call stack of the victim, and the victim also has a deque data structure and a Cilk worker data structure, with head and tail pointers and a current stack frame. So what happens when a thief comes along out of nowhere? It's bored, it has nothing on its deque. Head and tail pointers both point to the top.

Current stack frame has nothing. What's the thief going to do? Any guesses? How does this thief take the content from the worker's deque?

AUDIENCE: The worker sets their current stack frame to the one that [INAUDIBLE]

TAO SCHARDL: Exactly right, yeah. Sorry, was that-- I didn't mean to interrupt. All right, cool. So the red highlighting should give a little bit of a hint. The current stack frame in the thief is going to end up pointing to the stack frame at the top of the deque, pointed to by the top of the deque. And the head of the deque needs to be updated. So let's just see all those pointers shuffle. The thief is going to target the head of the deque. It's going to deque that item from the top of the deck.

It's going to set the current stack frame to point to that item, and it will delete the pointer on the deque. That make sense? Cool. Now the victim and the thief are on different processors, and this scenario involves shuffling a lot of pointers around. So if we think about this process, there needs to be some way to handle the concurrent accesses that are going to occur on the head of the deque.

You haven't talked about synchronization yet in this class, that's going to be a couple lectures down the road. I'll give you a couple of spoilers for those synchronization lectures. First off, synchronization is expensive. And second, reasoning about synchronization is a source of massive headaches. Congratulations, you now know those two lectures. No, I'm just kidding. Go to the lectures, you'll learn a lot, they're great.

In the Cilk runtime system, the way that those concurrent accesses are handled is by using a protocol known as the THE protocol. This is pseudo code for most of the logic in the THE protocol. There's a protocol that the worker, executing work normally, follows. And there is the protocol for the thief. I'm not going to walk through all the lines of code here and describe what they do. I'll just give you the very high level view of this protocol.

From the thief's perspective, the thief always grabs a lock on the deque before doing any operations on the deque. Always acquire the lock first. For the worker, it's a little bit more optimized. So what the worker will do is optimistically try to pop something from the bottom of the deque. And only if it looks like that pop operation fails does the worker do something more complicated. Only then does it try to acquire a lock on the deque, then try to pop something off, see if it really succeeds or fails, and possibly turn to a life of crime.

So the worker's protocol looks longer, but that's just because the worker implements a special case, which is optimized for the common case. This is essentially where the leave frame routine, that we saw before, is optimized for case one, optimized for the pop from the deque succeeding. Any questions about that? Seem clear from 30,000 feet? Cool. OK, so that's how a worker steals work from the top of the victim's deque.

Now, that thief needs to resume a continuation. And this is that whole process about jumping into the middle of an executing function. It already has a frame, it already has a [INAUDIBLE] state going on, and all that was established by a different processor. So somehow that thief has to magically come up with the right state and start executing that function. How does that happen? Well, this has to do with a routine that's the complement of the set jump routine we saw before.

The complement of set jump is what's called long jump. So Cilk uses, in particular Cilk thieves, use the long jump function in order to resume a stolen continuation. Previously, in our spawning function foo, we had this set jump call. And that set jump saved some state to a local buffer, in particular the buffer in the stack frame of foo. Now the thief has just created this Cilk worker structure, where the current stack frame is pointing at the stack frame of foo.

And so what the thief will do is it'll execute a call, it'll execute the statement, it will execute the long jump function, passing that particular stack frame's buffer and an additional argument, and that long jump will take the registered state stored in the buffer, put that registered state into the worker, and then let the worker proceed. That make sense? Any questions about

that?

This is kind of a wacky routine because, if you remember, one of the registers stored in that buffer is an instruction pointer. And so it's going to read the instruction pointer out of the buffer. It's also going to read a bunch of callee-saved registers and stack pointers out of the buffer. And it is going to say, that's my register state now, that's what the thief says. It just stole that register state. And it's going to set its RAP to be the RAP it just read.

So what does that mean for where the long jump routine returns?

AUDIENCE: It returns into the stack frame above the [INAUDIBLE]

TAO SCHARDL: Returns the stack frame above the one it just stole. More or less, but more specifically, where in that function does it return?

AUDIENCE: Just after the call.

TAO SCHARDL: Which call?

AUDIENCE: [INAUDIBLE]

TAO SCHARDL: To the spawn bar, here? Almost, very, very close, very, very close. What ends up happening is that the long jump effectively returns from the set jump a second time. This is the weird protocol between set jump and long jump. Set jump, you pass it a buffer, it saves and registers state, and then it returns. And it returns immediately, and on its directed vocation, that set jump call returns the value zero, as we mentioned before.

Now if you invoke a long jump using the same buffer, that causes the processor to effectively return from the same set jump call. They use the same buffer. But now it's going to return with a different value, and it's going to return with the value specified in the second argument. So invoking long jump of buffer X returns from that set jump with the value X. So when the thief executes a long jump with the appropriate buffer, and the second argument is one, what happens?

Can anyone walk me through this? Oh, it's on the slide, OK. So now that set jump effectively returns a second time, but now it returns with a value one. And now the predicate gets evaluated. So if not one, which would be if false, well don't do the consequent, because the predicate was false. And that means it's going to skip the call to spawn bar, and it'll just fall

through and execute the stuff right after that conditional, which happens to be the continuation of the spawn.

That's kind of neat. I think that's kind of neat, being unbiased. Anyone else think that's kind of neat? Excellent. Anyone desperately confused about this set jump, long jump nonsense? Any questions you want to ask, or just generally confused about why these things exist in modern computing? Yeah.

AUDIENCE: Is there any reason you couldn't just add, like, [INAUDIBLE] to the instruction point and jump over the call, instead?

TAO SCHARDL: Is there any reason you couldn't just add some fixed offset to the instruction pointer to jump over the call? In principle, I think, if you can statically compute the distance you need to jump, then you can just add that to RIP and let the long jump do its thing. Or rather, the thief will just adopt that RIP and end up in the right place. What's done here is-- basically, this was the protocol that the existing set jump and long jump routines implement.

And I imagine it's a bit more flexible of a protocol than what you strictly need for the Cilk runtime. And so, you know, it ends up working out. But if you can statically compute that offset, there's no reason in principle you couldn't adopt a different approach. So, good observation. Any questions? Any other questions? It's fine to be generally confused why their routines, set jump and long jump, with this wacky behavior. Compiler writers have that reaction all the time. These are a nightmare to compile.

Anyway, OK, so we've seen how a thief can take some computation off of a victim's deque, and we've seen how the thief can jump right into the middle of an executing function with the appropriate register state. Is this the end of the story? Is there anything else we need to talk about, with respect to stealing? Or, more pointedly, what else do we not need to talk about with respect to stealing? You're welcome to answer, if you like. OK.

Hey, remember that list of concerns we had at the beginning? List of requirements is what it was called. We will talk about syncs, but not just yet. What other thing was brought up? Remember this slide from a previous lecture? Here's another hint. So the register state is certainly part of the state of an executing function. What else defines a state of an executing function? Where doe the other state of the function live?

It lives on the stack, so what is there to talk about regarding the stack?

AUDIENCE: Cactus stack.

TAO SCHARDL: The cactus stack, exactly. So you mentioned before that thieves need to implement this cactus stack abstraction for the Cilk runtime system. Why exactly do we need this cactus stack? What's wrong with just having the thief use the victim's stack?

AUDIENCE: [INAUDIBLE]

TAO SCHARDL: The victim might just free up a bunch of stuff and then it's no longer accessible. So it can free some amount of stuff, in particular everything up to the function foo, but in fact it can't return from the function foo because some other-- well, assuming that the Cilk RTS leave frame thing is implemented-- the function foo is no longer in the stack, it won't ever reach it. So it won't return from the function foo while another worker is working on it. But good observation.

There is something else that can go wrong if the thief just directly uses the victim's stack. Well, let's take a hint from the slide we have so far. So the example that's going to be shown is that the thief steals the continuation of foo, and then the thief is going to call a function baz. So the thief is using the victim's stack, and then it calls a function baz. What goes wrong?

AUDIENCE: The victim has called something, but underneath, there is some other function stack [INAUDIBLE]

TAO SCHARDL: Exactly. The victim in this picture, for example, has some other functions on its stack below foo. So if the thief does any function calls and is using the same stack, it's going to scribble all over the state of, in this case spawn bar, and bar, which the victim is trying to use and maintain. So the thief will end up corrupting the victim stack. And if you think about it, it's also possible for the victim to call the thief stack. They can't share a stack, but they do want to share some amount of data on the stack.

They do both care about the state of foo, and that needs to be consistent across all the workers. But we at least need a separate call stack for the thief. We'd rather not do unnecessary work in order to initialize this call stack, however. We really need this call stack for things that the thief might invoke, local variables the thief might need, or functions that the thief might call or spawn. OK, so how do we implement the cactus stack?

We have a victim stack, we have a thief stack, and we have a pretty cute trick, in my opinion. So the thief steals its continuation. It's going to do a little bit of magic with its stack pointers.

What it's going to do is it's going to use the RBP it was given, which points out the victim stack, and it's going to set the stack pointer to point at its own stack. So RBP is over there, and RSP, for the thief, is pointing to the beginning of the thief's call stack.

And that is basically fine. The thief can access all the state in the function foo, as offsets from RBP, but if the thief needs to do any function calls, we have a calling convention that involves saving RBP and updating RSP in order to execute the call. So in particular, the thief calls the function baz, it saves its current value of RBP onto its own stack, it advances RSP, it says RBP equals RSP, it pushes the stack frame for baz onto the stack, and it advances RSP a little bit further.

And just like that, the thief is churning away on its own stack. So just with this magic of RBP pointing there and RSP pointing here, we got our cactus stack. Everyone follow that? Anyone desperately confused by this stack pointer? Who thinks this is kind of a neat trick? All right, cool. Anyone think this is a really mundane trick? Hopefully no one thinks it's a mundane trick. OK, there's like half a hand there, that's fine. I think this is a neat trick, just messing around with the stack pointers.

Are there any worries about using RBP and RSP this way? Any concerns that you might think of from using these two stack pointers as described? In a past lecture, briefly mentioned was a compiler optimization for dealing with stacks. Yeah.

AUDIENCE: [INAUDIBLE] We were offsetting [INAUDIBLE]

TAO SCHARDL: Right, there was a compiler optimization that said, in certain cases you don't need both the base pointer and the stack pointer. You can do all offsets. I think it's actually off the stack pointer, and then the base pointer becomes an additional general purpose register. That optimization clearly does not work if you need the base pointer stack pointer to do this wacky trick. The answer is that the Cilk compiler specifically says, if this function has a continuation that could be stolen, don't do that optimization.

It's super illegal, it's very bad, don't do the optimization. So that ends up being the answer. And it costs us a general purpose register for Cilk functions, not the biggest loss in the world, all right. There's a little bit of time left, so we can talk about synchronizing computation. I'll give you a brief version of this. This part gets fairly complicated, and so I'll give you a high level summary of how all of this works.

So just to page back in some context, we have this scenario where different processors are executing different parts of our computation dag, and one processor might encounter a Cilk sync statement that it can't execute because some other processor is busy executing a spawn subcomputation. Now, in this case, P3 is waiting on P1 to finish its execution before the sync can proceed. And synchronization needs to happen, really, only on the subcomputation that P1 is executing.

P2 shouldn't play a role in this. So what exactly happens when a worker reaches a Cilk sync before all the spawned subcomputations return? Well, we'd like the worker to become a thief. We'd rather the worker not just sit there and wait until all the spawned subcomputations return. That's a waste of a perfectly good worker. But we also can't let the worker's current function frame disappear.

There is a spawned subcomputation that's using that frame. That frame is its parent. It may be accessing state in that frame, it may be trying to save a return value to some location in that frame. And so the frame has to persist, even if the worker that's working on the frame goes off and becomes a thief. Moreover, in the future, that subcomputation, we believe, should return. And that worker must resume the frame and actually execute past the Cilk sync.

Finally, the Cilk sync should only apply to the nested subcomputations underneath its function, not the program in general. And so we don't allow ourselves synchronization, just among all the workers, wholesale. We don't say, OK, we've hit a sync, every worker in the system must reach some point in the execution. We only care about this nested synchronization.

So if we think about this, and we're talking about nested synchronization for computations under a function, we have this notion of cactus stack, we have this notion of a tree of function invocations. We may immediately start to think about, well, what if we just maintain some state, in a tree, to keep track of who needs this to synchronize with whom, which computations are waiting on which other computations to finish? And, in fact, that's essentially what the Cilk runtime system does.

It maintains a tree of states called full frames, and those full frames store state for the parallel subcomputations. And those full frames keep track of which subcomputations are standing and how they relate to each other. This is a high level picture of a full frame. There are lots of details highlighted, to be honest.

But at 30,000 feet, a full frame keeps track of a bunch of information for the parallel execution-

- I know, I'm giving you the quick version of this-- including pointers to parent frames and possibly pointers to child frames, or at least the number of outstanding child frames. The processors, when there's a system, work on what are called active full frames. In the diagram, those full frames are the rounded rectangles highlighted in dark blue. Other full frames in the system are, what we call, suspended. They're waiting on some subcomputation to return.

That's what a full frame tree can look like under, some execution. Let's see how a full frame tree can come into being, just by working through an animation. So suppose we have some worker with a bunch of spawned and called frames on its deque. No other workers have anything on their deques. And finally, some worker wants to steal. And I'll admit, this animation is crafted slightly, just to make the pictures a little bit nicer. It can look more complicated in practice, don't worry, if that was actually a worry of yours.

So what's going to happen, the thief is going to take some frames from the top of the victim's deque. And it's actually going to steal not just those frames, but the whole full frame structure along with it. The full frame structure is just represented with this rounded rectangle. In fact, it's a constant size thing. But the thief is going to take the whole full frame structure. And it's going to give the victim a brand new full frame and establish the child to parent pointer in the victim's new full frame.

That's kind of weird. It's not obvious why the thief would take the full frame as it's stealing computation, at least not from one step. But we can see why it helps, just given one more step. So let's fast forward this picture a little bit, and now we have another worker try to steal some computation, and we have a little bit more stuff going on. So this worker might randomly select the last worker on the right, steal computation from the top of its deque, and it's going to steal the full frame along with the deque frames.

And because it stole the full frame, all pointers to that full frame from any child subcomputations are still valid. The child's computation on the left still points to the correct full frame. The full frame that was stolen has the parent context of that child, and so we need to make sure that pointer is still good. If it created a new full frame for itself, then you would have to update the child pointers somehow, and that requires more synchronization and a more complicated protocol.

Synchronization is expensive, protocols are complicated. This ends up saving some complexity. And then it creates a frame for the child, and we can see this process unfold just a

little bit further. And we'll hold off for a few steals, we end up with a tree. We have two children pointing to one parent, and one of those children has its own child. Great. Now suppose that some worker says, oh, I encountered a sync, can I synchronize?

In this case, the worker has an outstanding child computation so it can't synchronize. And so we can't recycle the full frame, we can't recycle any of the stack for this child. And so, instead, the worker will suspend this full frame, turning it from dark blue to light blue in our picture, and it goes and becomes a thief. The program has ample parallelism. What do we expect to typically happen when the program execution reaches a Cilk sync?

We're kind of out of time, so I think I'm just going to spoil the answer for this, unless anyone has a guess handy. So what's the common case for a Cilk sync? For the sake of time, the common case is that the executing function has no outstanding children. All the workers on the system were busy doing their own thing, there is no synchronization that's necessary. And so how does the runtime optimize this case?

It ends up having the full frame, uses some bits of an associated stack frame, in particular the flag field. And that's why, when we look at the compiled code for a Cilk sync, we see some conditions that evaluate the flags within the local stack frame. That's just an optimization to say, if you don't need a sync, don't do any computation, otherwise some steals really did occur, go ahead and execute the Cilk RTS sync routine.

There are a bunch of other runtime features. If you take a look at that picture for a long time, you may be dissatisfied with what that implies about some of the protocols. And there's a lot more code within the runtime system itself, to implement a variety of other features such as support for C++ exceptions, reducer hyperobjects, and a form of IDs called pedigrees. We won't talk about that today. I'm actually all out of time. Thanks for listening to all this about the Cilk runtime system. Feel free to ask any questions after class.