PROFESSOR: OK. Let's get started. So what I'm going to do next is switch gears to one interesting compiler, which is the StreamIt parallelizing compiler.

The main idea about StreamIt is the need for a common machine language. What we want to do normally is, in a language, you want to represent common architecture properties so you get good performance. You don't do it at a very high level of abstraction, so you base a lot of cycles dealing with the abstraction. But you want to abstract out the differences between machines to get the portability, otherwise you are going to just do assembly-hacking for one machine. Also you can't have things too complex, because a typical programmer cannot deal with very complex things that we ask them to do.

C and Fortran was a really nice common assembly language for imperative languages running on the unicore machines. The problem is this type of language is not a good common language for multicores, because it doesn't deal with, first of all, multiple cores. And as you keep changing the number of cores -- for example, automatic parallelizing compilers are not good to basically get really good parallelism out of that, even though we talk about that. Still a lot of work has to be done in there.

So what's the correct abstraction if you have multicore machines? The current offering, what you guys are doing, is things like OpenMP, MPI type stuff. You are hand-hacking the parallelism. Well, the issue with that. It's basically this explicit parallel construct. It's kind of added to languages like C -- that's what you're working on. And what this does is all these nice properties about composability, malleability, debuggability, portability -- all those things were kind of out of the window. And this is why this parallelizing is hard, because all these things makes life very difficult for the programmer. And it's a huge additional program burden. The programmer has to introduce parallelism, correctness, optimization -- it's all left to the programmer.

So what the program has to do in this kind of world -- what you are doing right now -- you have to feed all the granularity decisions. If things are too small you might get too much communication. If things are too large you might not get good load balancing, and stuff like. And then you deal with all the load balancing decisions. All those decisions are left for you guys.

You need to figure out what's local, what's not. And if you make a wrong decision it can cost you.

All the synchronization, and all the pain and suffering that comes from making a wrong decision. Things like race

conditions, deadlocks, and stuff like that.

And this, while MIT students can hack it, you can't go and convince a dull programmer to convert from writing nice simple Java application code to dealing with all these complexities.

So this is what kind of led to our research for the last five years to do StreamIt. What you want to do is move a bunch of these decisions to the compiler. Granularity, load balancing, locality, and synchronization -- [OBSCURED] And in today's talk I am going to talk to you about, after you write a StreamIt program -- as Bill pointed out, the nice parallel properties -- how do you actually go about getting this kind of parallelism.

So in StreamIt , in summary, it basically has regular and repeating computation in these filters. This is called a synchronous data flow model, because we know at compile time exactly how the data moves, how much each produces and consumes. And this has natural parallelism, and it exposes exactly what's going to happen to the compiler. And the compiler can do a lot of powerful transformations, as yesterday I pointed out.

The first thing is, because of synchronous data flow, we know at compile time exactly who needs to do what when. And that really helps transform. It's not like everything happens run-time dynamically.

So what does that mean? So what that means is each filter knows exactly how much to push and pop -- that's in a repeatable execution. And so what we can do is, we can come up to the static schedule that can be repeated multiple times. So let me tell you a little bit about what a static schedule means. So assume this filter pushes two, this filter pops three but pushes one, that filter pops two. So these are kind of rate pushes, it's not everybody producing-consuming at once.

So what's the schedule? So you can say -- OK, at the beginning it produces two items, but I can't consume that because I need three items. And then I do two of them, and I can consume the first three and then produce one there. And I have two left behind. I do one more that, and now I got three. And it consumes that and produces that. And then I can fire C. So the neat thing about this is, when I started there was nothing inside any of these buffers. And if I ran A, A, B, A, B, C, there's nothing inside the buffers again. So what I have is, I'm back to the starting positioning. And if I repeat this millions of times -- I keep the computation running nicely without any buffers accumulating or anything like that. So I can come up with this very nice schedule that says -- here's what I have to do. I have to run A actually three times, B twice, and C once, in this order, and if I do that I have a computation that keeps running. And that gives me a good global view on what can I parallelize, what can I load balance, all those things. Because things don't change.

One more additional thing about StreamIt is we can look at more elements than I am consuming.

Question?

AUDIENCE: How common is it in your typical code that you can actually produce a static schedule like that?

PROFESSOR: In a lot of DSV code this is very common. A lot of DSV code right now that goes into hardware and software, they have very common properties. But even things that are not common, what has a very large chunk of the program has this static property, and there are some few places that has dynamic property. So it's like, when you write a normal program you don't write a branch instruction after every instruction. You have a few hundred instructions and a branch, a few tens of instructions and a branch type thing. So what you can think about it is that those instructions without a branch can get optimized the hell out of them, and then you do a branch dynamically.

So you can think about it like this. What's the largest chunks you can find that you don't have this uncertainty until run-time? Then you can optimize the hell out of it, and then you can deal with this run-time issues -- basically branches, or control for changes, or direct your rate changes at run-time. If we have 10-90 rule, if you get 90% of the things are in a nice thing, and if you get good performance on that -- hey, it has a big impact. So in our language you can deal with dynamism, but our analysis is basically trying to find the largest static chunk and analyze.

So most of the time that basically said we start with empty and end with empty. But the trouble is, a lot of times we actually can look beyond the number of what we consume. So what you have to do is kind of do initial schedule that you don't start with empty, you basically consume something -- you start with something like this. So the next time in something comes -- three things come into this one -- I can actually pick four and pop three. So you go through the first thing kind of priming everything with the amount of data needed, and then you go to the static schedule.

This kind of gives you a feel for what you'll get. This is a neat thing, I know exactly what's going on in here. So now how do I run this parallelism? This is something actually Rodric pointed out before, there are three types of parallelism we can deal with. So here's my stream program in here, and I do some filters, scatter-gather in here.

The first site of parallelism is task parallelism. What that means is the programmer said, there are three things that can run parallelly before I join them together. So this is a programmer-specified parallelism. And you have a nice data parallel messenger presentation.

The second part is data parallelism. What that means is, you have some of these things that don't depend on the previous run of that one. So there's no invocation, dependency across multiple invocations. These are called stateless filters, there's no state that keeps changing. If the state kept changing, you had to wait till the previous

one finishes to run the next one. So if you have a stateless filter -- assume that it's data parallel -- what you can do is you can basically take that, replicate it many, many times, and when the data comes -- parallel is in it every data -- and it will compute and parallelly get out here.

The final thing is pipeline parallelism. So you can feed this one into this one, this one into this one, and then douse across in a pipeline fashion. And you can get multiple things execution. So we have these three types of parallelism in here, and the interesting thing is if you have stateful filters, you can't run this data parallel. Actually the only parallelism you can get is pipeline parallelism.

So traditionally task parallelism is fork/join parallelism, that you guys are doing right now. Data parallelism is loop parallelism. And pipeline parallelism mainly was done in hardware. If you have done something like Verilog or VHDL you'll do a lot of pipeline parallelism. So kind of combining these three ideas from different communities all into one, because I think programs can have each part in there.

So now, how do you go and exploit this? How do you go take advantage of that? So I'll talk a little bit of baseline techniques, and then talk about what StreamIt compiler does today.

So assume I have a program like this. The hardest thing is there are two tasks in here. The programs are given, you don't have to worry anything about that. And what you can do is assign them into different cores and run it. Neat. You can think what a fork /join parallelism is, you come here you fork, you do this thing, and you join in here. So the interesting thing is if you have two cores. You probably got a 2x speedup in this one. This is really neat because there are two things in here. The problem is, how about if you have a lot more different number of cores? Or if the next generation has double the number of cores, and I'm stuck with the program you've written for the current generation? So this not that great, interesting.

So we ran it on the Raw processor we have -- it has 16 cores in there -- that we have been building, and this is actually running a simulator of that. What you find is, is a bunch of StreamIt programs we have we kind of get performance like basically close to two, because that's the kind of parallelism people have written. In fact, some programs even slowed down in there, because what happens in here is the parallelism and synchronization is not matched with the target -- because it's matched with the program. Because you wrote a program because your parallelism in there was what you thought was right for the algorithm. We didn't want you to give any consideration to the machine you are running, and it didn't match the machine, basically, if you just got the parallelism. And you just don't do that right.

So one thing we have noticed for a lot of streaming programs, to answer your question, is there are a lot of data parallelism. In fact, in this filter -- in this program -- what you can do is you can find data parallel filters, and parallelize them. So you can take each filter, run it on every core for awile. Get the data back. Go to the next filter,

write on every go-while, get that back. So what you can do is, if you have four cores in here, you can each replicate all this four times. Run these four for a while, and then these four, these four, these four, these four. OK? So that's the nice way to do that. So the nice thing about doing that is you have a lot of nice in the load balancing, because each are doing the same amount of work for a while. And after it accumulates enough data you go to the next one, do for a while, and then like that. And each group basically will occupy the entire machine -- you just go down this group like that.

And so we ran it, it started even slower. Why? It should have a lot more parallelism, because all those filters were data-parallel. So you sort of gettting stuck with two, now we can easily run a parallelism of 16, because data parallelism you can just put it any amount in there. But we are running slow.

AUDIENCE: Communication overhead?

PROFESSOR: Yeah, it could mainly be communication overhead, because what happens is you run this for a small amount of time. You had to send it all over the place, collect it back again, send it all over the place, collect it back again. The problem is there's too much synchronization and communication. Because every person at the end is like this global barrier, and the data has to go shuffling around. And that doesn't help.

So the other part, what you can do in the baseline is what you call hardware pipeline. What that means is you can actually do pipeline parallelism. The way you can do that is you can look at the amount of work each filters contain, and you can combine them together in a way that the number of filters is going to be just about the number of tiles available. Most programs have more filters than the number of cores. So you review combined filters, to give us a number of filters, is just either the same, or one or two less than the number of cores available. In a way that you combine them so each of them will probably have close to the same amount of work. The problem is if when you combine it's very hard to get the same amount of work.

And if you assume eight cores, you can do this combination and we can say -- aha, if I do this combination, I have one, two, three, four, five, six, seven. Eight cores, I can get seven of them. Hopefully each of them have the same amount of work, and I can run that. And then we assign this to one filter and say -- "You own this one, you run it forever. You get the data from the guy who owns this one, and you produce at this one." And if you have more cores you can actually keep doing some of that. If you have enough filters you can each combine them and do that.

So we perform, and we got this. Not that bad. So what might be the problems here?

AUDIENCE: Hardware locality. You want to make sure that the communicating filters are close to each other.

PROFESSOR: Yeah, that we can deal with. It's not a big locality [OBSCURED] What's the other problem? The bigger problem.

AUDIENCE: [NOISE] load balance.

PROFESSOR: Load balance is the biggest problem, because the problem is you are combining different types of things together, and you are hoping that each chunk you get combined togeher will have an almost identical amount of work. And that's very hard to achieve most of the time, because dynamically things keep changing. The nice thing about loops is, most of the time if you have a loop or state if you replicate it many times, it's the same amount of code, same amount of work. It nicely balances out. Hardware -- combining different things becomes actually much harder.

So again, parallelism and synchronization are not really matched to the target. So the StreamIt compiler right now does two, three things. I'll go through details. Coarsen the granularity of things. So what happens is if you have small filters it combines them together to get the large stateless areas. It data parallelizes when possible. And it does software pipelining, that's a pipeline parallelism. I'll go through all these things in detail. And you can get about 11x's speedup by doing all those things.

So coarsen the stream graph. So you look at this stream graph and say -- wait a minute, I have a bunch of data-parallel parts. And before what I did was I take each data-parallel part, when 16 then came or get together, went 16 came together, went 16. Why? I have put too much communication.

Can I combine data-parallel things into one gigantic unit when possible? Of course, you don't want to combine a data-parallel part with a non-data-parallel part. Then the entire thing becomes sequential, and that's not helpful. So in here what we found is these four cannot be combined, because if you combime them the entire thing becomes sequential. So what we have to do is, you can combine this way. So all those things are data-parallel, all those things are data-parallel. And even though they are data-parallel if you combine them they become non-data-parallel, because this is actually doing peeking, it's looking at more than one, and so it's looking at somebody else's iteration work. So you can't combine them. So what the benefits of doing this is you reduce global communication basically.

And the next thing is you want data parallelizing to four cores. And this one fits four ways in there. But the interesting thing is, when you go in this one you realize there's some task parallelism. We know there are two tasks that have the same amount of work in here. So facing this four ways, and facing this four ways, and giving the entire machine to this one, and giving the entire machine to this one, might not be the best idea. What you want to do is you want to face it two ways. And then basically give the entire machine to all of these running at the same time, because they're load balanced -- because they are the same thing repeated. And you can do the

same thing in here.

OK. So that's what the compiler does automatically, and it preserves task parallelism. So if you are task parallelism you don't need -- the thing about that is the parallelism you need, you don't need too much parallelism. You need enough parallelism to make the machine happy. If you have too much parallelism you end up in other problems, like synchronization. So this gives enough parallelism to keep the entire machine happy, but not too much.

And by doing that actually we get pretty good performance. There are a few cases where this hardware parallelism wins out, these two, but most of them -- actually this last one we can recover -- do it pretty well.

OK. So what's left here is -- so this is good parallelism and low synchronization. But there's one thing, when you are doing data parallelism there are places where there are filters that cannot be parallelized -- they are stateful filters. Because you can't run the data parallelism, and according to Amdahl's Law that's actually going to basically kill you, because that's just waiting there and you can't do too much.

I'm going to show that using this separate program -- so this number is the amount of work that each of them has to do. So this is actually a lot of work, a lot of work -- this does a little work in each of these filters. So if you look at that, these are data parallel but it doesn't do any much work. Just parallelizing this doesn't help you. And these are data parallel. And these actually do enough work. Actually we can go and say I am replicating this four times, and I'm OK. I'm getting actually good performance in here.

Now what we have is a program like this. And so if you are not doing anything else that we have data parallelism in. So what happens in the first cycle you run these two. And then you run data parallel this one, and then you run these, and then you run data parallel this one. And if you look at that, what happens is we have a bunch of holes in here. Because at that point when you are running that part of the program there's not enough parallelism, and you only have two things in there. And when you're running this you can run this task parallelism in here, but there's nothing else you can do in here. And so you get basically 21 time steps each -- time minutes basically will run into that program.

But here we can do better. What we can do is we can take and try to move that there, and kind of compress them. But the interesting thing is these things are not data parallel. So how do I do that? So the way to do that is taking advantage of pipeline parallelism. So what you can do is you can take this filter in here. Since each of the entire graph can run only sequentially -- this has to run after this -- you can look at the filters running separately like that, and kind of say, instead of running this and this and this, why don't I run this iterations of this one. This iterations of this invocation. And this interations of this one. And this iterations on the next one. And I'm still maintaining -- because when I'm running this even though the -- I'm not running anything data parallel here because these ones

were already done previously, so I can actually use that value. And so I can maintain that dependency, but I'm running things from the different iterations. And so what I need to do is, I need to kind of do a prologue to kind of set everything up in there. And then I can do that and I don't have any kind of dependence among these things. So now what I can do is I can basically take thes two and basically lay out anything anywhere in those groups, because they are in different iterations and since I am pipelining these I don't have any dependence in there. So I end up in this kind of a thing, and basically much compress in here.

And by doing that what you actually get is a really nice performance. The only place that this actually wins -- hardware pipelining, and this little bit in there. But the rest you get a really good win in here.

OK. So what this does is basically now we got a program that when the programmer never thought anything about what the hardware is -- just wrote abstract graph and data streaming. And given Raw, we automatically actually mapped into it, and figured out what is the right balance, right communication, right synchronization, and got really good performance. And you're getting something like 11x performance. If you do hard hand, if you work hard probably you can do a little bit better. But this is good, because you don't hand-do anything. The killer thing is now I can probably take this set of programs -- which we are actually working on -- is you can take them to Cell which has, depending on the day, six cores, seven cores, eight cores, and we can basically get to matching the number of cores in there. So this is it because right now what happens is you have to basically hand code all those things, and this can automate all that process.

So that's the idea, is can you do this -- which we haven't really proved and this is our research -- write once, use anywhere. So write this program once in this abstract way. You have to really don't think about full parallelism. You have to think about some amount of parallelism, how this can be put into a stream graph, but you are not dealing with synchronization, load balancing, performance. You don't have to deal with that. And then the compiler will automatically do all these things behind you, and get really good performance.

And the reason I showed this was -- I'll just play one more slide I think -- showed this was it's not a simple thing. The compiler actually has to do a bunch of work, the work that you used to do before. Things like figuring out what's the right granularity, what's the right mix of operations, what type of transformations you need to do to get there. But at some point we did three things -- coarse-grained, data parallel, and software pipelining. And by doing these three we can actually get a really good performance in most of the programs we have. So what we are hoping is basically this kind of techniques can in fact help programmers to get multicore performance without really going and dealing in the grunge level of details you guys do. You guys will appreciate that, and hopefully will think of making -- because now at the end of this class, you will know all the pain and suffering the programmers go through to get there. And the interesting thing would be to in fact look at the ways to basically reduce that pain

and suffering.

So that's what I have today. So this was, as I promised, a short lecture -- the second one. Any questions?

AUDIENCE: So if we've got enough data parallelism we'll have the same software pipeline jumping on each tile? Is that right?

PROFESSOR: Yes.

AUDIENCE: OK. So if you do that how does it scale up to something that has higher communication costs than Raw? By doing this software pipelining you have to do all of your communication off tile.

PROFESSOR: So the interest in there right now is we haven't done any kind of hardware pipelining. We are kind of doing -- everybody's getting a lot of data moving in there. The neat thing about right now is, even with the SP in Cell and even Raw, the number of tiles are still small enough that a lot of communication -- unless way too much communication -- it doesn't really overwhelm you. Because everybody's nearby, you can send things. They talk a little bit about in Cell that near enableness helps, but not that much. But as we go into larger and larger cores, it's going to become an issue. Near enables become much easier to communicate, and you can't do global things in there. And at that point you will actually have to do some hardware pipelining. You can't just assume that at some point everybody's going to get some data and go to something. So what you need to do is have different chunks that the only communication that would be between these chunks would be kind of a pipeline communication. So you don't mix data around. So as we go into larger and larger cores you need to start doing techniques like that.

The interesting thing here is even though what you had to change was the compiler -- hopefully the program stays the same -- right now it's not an easy issue, because our compiler has 10 times more core than the program, so it's easier in the program. But if you look at something C, the core base is millions of times larger than the size of the compiler. So at some point they'll be switched. It's easier to change the compiler to kind of keep up to date. That's what happened in C. Every generation you change the compiler, you don't ask programmers where to code the application. So can you make these kind of things as the multicores become different -- bigger, have different features. You change the compiler to get the performance, but have the same code base. That's the goal for portability.

AUDIENCE: Have you tried applying StreamIt or the streaming model in general, to codes that are not not very clearly stream-based but using the streaming model to make communication explicit, such as scientific codes. Or, for example, the kinds of parallelizable loops that you covered in the first half of the lecture.

PROFESSOR: Some of those things, when you have free form simple communication can map into streaming. So for example, one thing we are doing is things like right now MPEG. Some part of the MPEG is nicely StreamIt, but

when you actually go inside the MPEG and dealing with the frame, it's basically a big array, and you're doing that. So how do you chunkify the arrays, and basically deal with it in a streaming order? There's some interesting things you can do.

There will be some stuff that doesn't fit that. Things like pattern recognition type stuff, where what you want to do is you want to -- assume you're trying to -- good example. You're trying to feature a condition in a video. And what happens is the number of features, can you match or connect two features, or match and connect a thousand features. And then each feature you need to do some processing. And that is a very dynamic thing. And that doesn't really fit into streaming order right now.

And so the interesting thing is, the problem we have been doing is we are trying to fit everything into one. So right now the object-oriented model is it basically -- everything has to fit in there. But what you're finding is there are many things that don't really fit nicely. And you'll do these very crazy looking things just to get every program to fit into the object-oriented model. That doesn't really work. I think the right way to work is, is there might be multiple models. There's a streaming model, there's some kind of a threaded model, there might be different ones -- I don't know what other models are.

So the key thing is your program might have a large chunky model, another chunky model. Don't try to come up with -- right now what we have is we have a kitchen sink type of language. It tries to support everything at the same time. And that doesn't really work because then you have to think about and say -- OK done, can I have a pointer here? And I need to think about all the possible models kind of colliding in the same space.

AUDIENCE: On the other hand, the object-oriented model is much more generalized to me. It's not the best model for many things, but it's much more generalizable than some models. And having a single model cuts down on the number of semantic barriers you have to cross --

PROFESSOR: I don't know but --

AUDIENCE: Semantic barriers incur both programmer overhead and run-time overhead.

PROFESSOR: See the problem with right now with all the semantic barriers, is object-oriented model plus a huge number of libraries. If you want to do OpenGL, it's object-oriented but you have no library. If you want to do something else, you have to learn the library.

What the right thing would be, instead of trying to learn the libraries is learn kind of a subset language. So you have nice semantics, you have nice syntax in there, you have nice error checking, nice optimization within that syntax. Because the trouble is right now everything is in this just gigantic language, and you can't do anything.

And in the program you don't even know, because you can mix and match in really bad ways. The mix and match gives you a lot of power, but it can actually really hurt. And a lot of people don't need it. Like for example in C, people doubt it was really crucial for you to access any part of memory anywhere you want. You just can go and just access any program, anywhere, anytime in there. If you look at it, nobody takes advantage of that. How many times do you write the program an say -- "Hey, I want to go access the other guy's stack from this part." That doesn't work. You have a variable and you use a variable.

AUDIENCE: It still [OBSCURED]

PROFESSOR: Yeah, but the thing is because of that power, it creates a lot of problems for a compiler -- because it needs to prove that you're not doing that, which is hard. And also, if you make a mistake the program is like -- "Yeah, this looks like right. It still matches my semantics and syntax, I'll let you do that." But what you realize is that's not something people do -- just stick with your variable. And if you don't go to variables -- that's what type-safe languages do -- it's probably more for bugs than a feature.

And the same kind of thing having efficiency in language, is you can do everything at the same time. Why can't you have a language that you can go with this kind of context. I'm in the streaming context now. I say this is my streaming context. I am in a threaded context. Then what that does is, I have to learn the full set of features, but I restrict what I'm using here. That can probably realistically improve your program building, because you don't have to worry about --

AUDIENCE: It gives the programmer time to get to know each language.

PROFESSOR: But right now you have to do that. If you look at C# it has all these features. It has streaming features, it has threaded features, it has every possible object-oriented feature.

AUDIENCE: Right, but there's a compact central model which covers most things. You can pull in additional features and fit them [OBSCURED]. You can do pointer manipulation in C#, but you bracket things into an unsafe block. And then the compiler knows in there you're doing really bad things.

PROFESSOR: That's a nice thing, because you can have unsafe. But can you have something like -- this is my streaming part. OK. Can I do something like that, so I don't have to worry about other? The key thing is, is there a way where -- because right now, my feeling is if you look at the object-oriented part. So if you are doing, for example, Windows programming, you can spend about a week and learn all the object-oriented concepts. And you have to spend probably a year to learn all the libraries on top of that. That's the old action these days. It's basically the building blocks have become too low, and then everything else is kind of an unorganized mess on top of that. Can you put more abstraction things that easy? Hey, I'm talking about research, this is one possible

angle. I mean there might -- you can think, I know there are messes that I think in there.

My feeling is what we do well is when things get too complicated we build abstraction layers. And the interesting thing there is, we build this high level programming language abstraction layer. And then now we have built so much crud on top of that without any nice abstraction layers, I think it's probably time to think through what there could be at the abstraction level. Things like, it's hitting -- that is where parallelism is really hitting. Because that layer, the object-oriented layer, doesn't really support that well. And it's all kind of ad hoc on top of that. And so that says something. Yes, it's usable. We had this argument -- assembly languages programmers -- for two decades. There are people who were swearing by assembly languages. They could write it two times smaller, two times faster than anything you can write in high level language. It's probably still true today. But at the end there were things that high level languages won out. I think we are probably in another layer like that. I don't know, probably will go with that argument. You can always point to something saying this is something you cannot do. If there are still things -- like structured programs and unstructured programs, we talked about that. That argument went for a decade.

AUDIENCE: The question I would pose is can you formulate a kitchen sink language at a parallelizable level of abstraction?

PROFESSOR: Ah. That's interesting, because parallelization is -- one of the biggest things people have to figure out is composability. You can't have two parallel regions as a black box put together. You start running into deadlocks and all those other issues in there. Most of the things that you work is the abstraction works, because then you can compose at a higher level abstraction. You can have interface and say -- here are something interface, I don't know what's underneath, I compose at the interface level. And then the next guy composes at the higher level, and everything is hidden. We don't know how to do that in parallelism right now. We need to combine two things, it runs into problems. And the minute you figure that one out -- if somebody can figure out what's the right abstraction that is composable, parallel abstraction -- I think that will solve a huge amount of problems.

AUDIENCE: Isn't it Fortress that attempted to do something that's parallelizable and the kitchen sink, but that then leaves all the parallelizable --

AUDIENCE: I'm saying how terrible [OBSCURED] programmers.

PROFESSOR: But I would say right now is a very exciting time, because there's a big problem and nobody knows the solution. And I think for industry they lose a lot of sleep over that, but for academia it's the best time. Because we don't care, we don't have to make money out of these things, we don't have to get production out of it. But these actually have a very open problem that a lot of people care about. And I think this is fun partly because of that.

I think this huge open problem that if you talk to people like Intels and Microsoft, a lot of people worry a lot about they don't know how to deal with the future in 5-10 years time. They don't see this is scaling what they're doing. And from Intel's point of view, they made money by making Moore's Law available for people to use. They know how to make it available, but they don't know how to make people use it.

From Microsoft's point of view, their current development methodology is almost at this breaking point. And if you look at the last time this happening -- so things like Windows 3.0, where their current development methodology doesn't really scale, and they really revamped it. They came up with all this process, and that had lasted until now. For the last two Office and Vista, just realized they can't really scale that up. So they are already in trouble, because they can't write the next big goal is just two times bigger than Vista, and hopefully get it working. But on top of that, they have it thrown this multicore thing, and that really puts huge amount of burden. So they are worried about that.

So from both their point of view, everybody's clamoring for a solution. And things like last time around -- I'll talk about this in the future -- last time around when that happened, it created a huge amount of opportunities, and bunch of people who sold it kind of became famous. Becuase they say -- we came up with a solution, and that people started using and stuff like that. Right now, everybody's kind of waiting for somebody to come up and say here's the solution, here's a solution.

And there are a lot of -- Fortress type exports is one, and what we are doing is one. And hopefully some of you will end up doing something interesting that might solve it. This is why it's fun. I think we haven't had this much of an interesting time in programming languages, parallelism, architecture in the last two decades.

With that, I'll stop my talk.