PROFESSOR: So we have Arvind today going to talk about how actually do parallelism much closer to hardware and all the work we have been doing.

ARVIND: So first, just a clarification. This talk is not really about any techniques. This is some very primary ideas that I have some new ideas on parallel programming. The second thing is 5, 6 years ago I was going around the country giving a talk, why hardware design can't be left to hardware designers. Because I think the way they expressed their ideas is very, very low level from software point of view. So I have worked pretty hard on injecting some fairly sophisticated ideas from software into hardware design. But it often happens when you deal with powerful people, you get effected by it. So I have sort of come the full cirlce. Now I'm going to show you how hardware design can influence software design and that's really the idea behind this talk. So that's borrowing some ideas from hardware design.

Cell processor is a very good example of a system on a chip and this is what everybody believes the future will look like. That you'll have fairly symmetric things in some sense and that is needed by hardware because hardware gets out of hand if you have wires running all over the place. So people would like these things to be fairly regular in some sense, but that doesn't mean that all these styles have to be the same. So for example, you may have very application specific blocks on your chip. So for example, in your cell phone you have many, many blocks which are specialized, especially all the radio communication. Because if you did that stuff in software you may not be able to meet the performance and even if you can meet the performance your battery will drain instantaneously. I mean, it just takes too much power to do that kind of computation in software. So for various reasons we need application specific processing units and I'm sure cell processor has quite a few of these. By the way, these application specific units, they're often called ASICs. And you can think of them as a separate chip, but in due course of time they become blocks of a bigger thing. You may have general purpose processors.

Already, systems have more than one such thing. Cell processor has how many? Like 8.

PROFESSOR: Depends. Some have 8, but they don't [OBSCURED] So the current Playstations already list supposedly 7, but we've had only 6. So, the [OBSCURED].

PROFESSOR: And high-end cell phones have 2 general purpose processors and 2 DSPs on it, so you certainly have -- multiple processors you will have memory banks, et cetera. And you will have structured on-chip interconnection networks.

ARVIND: So I think this is not controllers. That this is what the trajectory will look like in the future. That doesn't mean that all the chips will look the same. You may have different mixes of things depending upon the application and one of the interesting things is can you put together one of these things fast enough because it's very, very expensive to design and implement these things. So really one issue that comes up all the time is can we rapidly produce high quality chips and surrounding systems and software for such thing?

So this talk, as I said, it's about ideas and what I'm going to do is I'm first going to tell you how I used to think about parallel programming. And this way is all about threads. You know, where are my threads? And I worked on this problem for a very, very long time. Like 20 years and this is not necessarily wrong, You know I'm sure in this course itself you have seen quite a bit about this and I think tomorrow's lecture is on Cilk so you will see even more about threads. And lately, I've been thinking about it differently. And really the main point I would like to get across somehow to you is now I think of parallel little programming module as a resource. What do you mean as a resource? I'll try to make it more concrete, this idea. And the whole issue here is I think this viewpoint is valuable, but it's not proven yet. You know, so it may not be right. So just a warning.

I've been working with this on students, in particular. Now those of you who have heard of transactional programming, lots of things I will say you'll say oh, you mean a transaction? Yes, I mean there are very, very strong connections for transactional stuff, but that's not the language I'm going to be using here.

These slides are color-coded. So when I was talking about old world thinking it's yellow and when it's new thinking then it's white. So well, I think you probably have already heard this. That the only reason for parallel programming used to be performance. This made programming extremely difficult. You had to know a lot about the machine. You were programming in code were not portable. Endless performance tuning, all this still goes on. I mean, very few of these things have changed. Parallel libraries were not composable. And this is one of the main things I worry about a lot and I think Allen was alluding to it too. Somebody has code, he just wants to run the damn thing. You know, even at factor of 2 speed that would be what fantastic. You know, often it runs slower. Very, very fun, machines. Very different to deal with heap structures and memory hierarchies. And then there is always the synchronization cost issue. In some sense parallel programming doesn't make sense if there is no synchronization going on and that's like this embarrsingly parallel computation you just start off and things never talk to each other. In even moderately interesting parallel programs there's always some communication between various tasks and when that happens synchronization cost becomes an issue and it is such a major issue right now that you have to think about it out front. Oh, if I synchronize too much then I know for a fact that my program's going to run like a dog. So you try to avoid synchronization and make things coarser and so on.

Really the whole issue used to be how to exploit hundreds of threads from software because new hardware will support many, many threads. But at the same time, it's important to understand that in this world there's always virtualization. So number of threads in your machine is never going to match exactly what you have in your application. So any programming model you have, virtualization is a very important aspect of it. That I have n threads and n may be dynamic, maybe varying with time in my application, but in hardware I have some fixed number of threads. Which is significantly smaller in general than what you have in your application. So the thing I was most interested in all those days was what I call implicit parallelism. And from the questions I was hearing about analysis you're doing of your programs, all those issues come up there.

You know, given some code can I figure out what can be done in parallel and just run it like that? So expect parallelism from programs written in sequential languages and people like some of the champions in this and I have also spent lots and lots of time on this research problem. So this is one of those areas which certainly has not suffered because of lack of research. There's tons and tons of research in this. You know, you can go to IBM and there will be 50 Ph.D.'s working on Fortran compiler. Trying to extract parallelism. But the success is limited. In fact, the main takeaway from this research is people have learned now how they should write their programs so that compiler has a chance of extracting parallelism. Because now people look at your code, they say, well, this is a bad code. Why? Because compiler can't analyze it. You become part of the equation in this.

Now the methodology I was pushing for a long time was functional languages. Programs in functional languages which may not obscure parallelism in the first place. I'm not going to talk about functional languages. It's immaterial, you know, whether this is a good idea or a bad idea, the fact of matter is in real life people's reaction is functional languages-- are you kidding me? So it doesn't get off ground. It's one of those things I can come and preach to you is good for your soul. And they leave me alone. I don't do movies with subtitles.

AUDIENCE: Sorry to temporarily derail you--

ARVIND: Sorry?

AUDIENCE: Sorry to temporarily derail you, so what kind of parallelism is easier to extract from a functional language? I can clearly see some task parallelism, can you also extract data parallelism or pipeline parallelism or what not?

ARVIND: It turns out that's actually not that right question to ask in the functional language because functional language doesn't obscure the parallelism that was there in the first place. So you already have a partial order on operations as opposed to a sequential order. You see when you write in Fortran even a simple thing like f of g of x, h of y. It may be obvious to a functional programmer that these things can be done in parallel. Perhaps, the execution of these can overlap with the evaluation of f, but that's not the semantics of Fortran or any other sequential language. The semantics is you're supposed to execute the program left to right, top to bottom. So you will go and execute g first and you'll go inside g and execute it and then you will go and execute h and then you will go and execute f. Why is that important? Because there are side effects, so it's conceivable that you do something in this which affects this and so on. So you're given a sequential order and then compiler does deep analysis, the kind you're doing right now it terms of dependence and anti-dependence and so on. Which says it's OK to do g and h in parallel. Well, if it was a functional program then you know by the semantics of the language g and h can be done in parallel. That doesn't mean it's a good idea to do it in parallel. So when we get to the second point of mapping this computation onto a given substrate off 2 processors or 10 processors then problems become similar. But the problem of detecting parallelism is very different. Because you don't talk of detecting parallelism in functional languages, it's already there. You don't obscure it. Does that make sense?

AUDIENCE: Yeah, maybe.

ARVIND: And it goes without saying that if your algorithm has low parallelism then functional languages or any other language and there's no magic-- you can't suck blood out of stone. You have to know your algorithm. You have to know what operations can be done in parallel. So language only is an aid. In some sense that if it was parallel I could express it more easily as a parallel program. And if it's a bad language, you obscure it and then you try to undo the damage.

Now as I said, there has been a lot of success in this in terms of dense matrix kind of stuff, but more irregular the computation harder it gets to do these things statically. So when you can't do it then people of course, designed explicitly parallel programming models and some of the most successful ones in this are data parallel and you'll hear about multithreading tomorrow in Cilk. And then there are very low level models where you are doing actual message passing between various modules that are sitting over there and I can expose you threads and synchronizations that are very low-level fork and joint, et cetera. So of course, high-level models are preferable because you're likely to make fewer errors in them. The question is, can we compile them? And in some cases what turns out is this is a good model, but it is not general enough. So if you have a data parallel algorithm, fine. I mean, you should use these things. But not all applications fit the data parallel model, multithreading in some sense is more general.

This is what I mean by a multithreaded model that you may have a main which spawn off things. So not only these things can be done in parallel, but execution of these can overlap the execution of f. So parents and children may be executing simultaneously. So if you look at this as an activation tree, this invokes a loop which has many, many iterations in it. When we talk of sequential computing you're always sitting on one branch of this. So that's the stack. This stack, this stack, stack frames. When you talk of parallel computing then part of this activation tree is concurrently active. So all problems get harder. Storage management gets harder. And what gets especially harder is that there is this global heap. You always have heap in interesting programs, global data structures. Now the problem is if this and this are active at the same time or this and this are active at the same time and they are both reading and writing into the data structure then naturally there's a race condition in this. Even if only 1 guy's writing and 5 people are reading it. It's a question of the guy should not read it too soon. So the moment you do any kind of parallel programming, synchronization issues arise immediately.

You know, you always have to worry about, how do I indicate that it's OK to read this? Well, you can follow some control ideas. You say, this guy was writing. He has finished execution, so it must be OK to write. Or you can go to very fine-grained synchronization, you can have some bit here which says, oh this data has been updated, now it's OK to unlock it and read it and so on. So all these ideas have been explored and they're still being explored in this context. The main takeaway is instead of a stack you will have a tree which is active at the same time. Many, many things are going on. And the second thing is that there is a competition. There is a race in reading the heap data structures and therefore you have to worry about that you don't read it too soon or you don't write it too soon because if you write it too soon you may clobber before somebody else gets a chance to read it. So these issues are quite serious.

Now it's really possible-- I mean, I claim you know, I have languages which can express computation very efficiently this way. Efficiently is not the right word. Very easily you can express computation in this way. But at the end of the day the goal is to take that stuff and run it somewhere on some parallel machine. Whether it was cell processor in the old days if would have been parallel machines of various ergs. And that has proven to be quite difficult, efficient mapping of these things because it turns out that you say, oh, you didn't tell me before you have only 2 processors. Then I would have expressed the computation differently. Or you didn't tell me that you have 1000 processors. So there is this kind of yes, there is virtualization. But it's not virtualization enough. Qualitatively matters a lot. You know, what is the target? And that effects how you would have expressed the computation. So I used to go around saying this all the time that parallel programming is so important. That really sequential programming will be taught as a special case of parallel programming because it will be all parallel. If you have only one processor then we can teach you some special tricks how to run it efficiently on one processor. So this as you can see, is largely an unrealized dream.

AUDIENCE: That's an old idea, but not one to be dismissed.

ARVIND: Not one to be dismissed. Absolutely. So the question is, has the situation changed? And the situation has certainly changed. So multicores have arrived. This is a big deal. I mean, Microsoft wants to exploit parallelism. There can be no bigger indication than this. And there is explosion of cell phones. And if you don't understand what that means economically, you know there are 100 million PCs that will be sold this year and 950 million cell phones that'll be sold this year. This is the same kind of transition that is taking place that happened in the early 80s when we went from mainframes and mini- computers to micros. So what happens there determines what happens everywhere else. So it's quite possible then what happens on these small devices, hand-held devices will determine the architecture of everything else just because of the numbers. People are willing to invest a lot more money into this explosion of game boxes.

AUDIENCE: [OBSCURED]

ARVIND: I'm sorry?

AUDIENCE: The explosion of laptops is also happening.

ARVIND: Yeah, but still that number is 100 million. While cell phone is 950 million, at least in 2000.

AUDIENCE: These are cell processors.

ARVIND: Right. Absolutely. So look, if I'm talking to you as a teacher I mean, my message to my colleagues in the department is it's no longer good enough to say well we'll teach you parallelism when you're a sophomore. I mean, it is the thing that people want to deal with. You say, what do you mean telling me how to program one processor. I have 10 of them in my pocket. So I think we have to deal with this. You know, how do we introduce parallelism? What method we should have for teaching parallel programming so that is the default programming model in your head? So it's all about parallelism now. No longer an advanced topic. It has to be the first topic.

Just another word of caution in this. So let's look at cell phones. So mine has some bugs in it so sometimes it misses a call when I'm surfing the web. It doesn't rate. So this is the deep question now, so whose fault is it? So for example, to what extent the phone call software should be aware of web surfing software or vice versa. So now you have to understand how these things are done. Of course, how the phone calls are made, that software has been evolving, right? You can look at it all in the last 15 years and you'll be able to trace a continuum. You know how phone calls are made and that's not trivial among the software on your cell phone. Well, when it comes to web surfing software I can guarantee you it was not written from scratch. People went to your PC, they say, what do you do on your web? They take all that software say, let's port it on the cell phone.

Now the guy who wrote the web software never thought that you would be talking on telephone while you're surfing the web or at least not on the same device. And vice versa, the guy who wrote the phone software never thought you would be surfing the web while you're talking on the phone. I mean even 1995 or 1998 if somebody had said, you'll be surfing the web at the same time. It would have sounded pretty bizarre.

Is it merely a scheduling issue? So this is the answer saying ah, they didn't process the interrupt properly. I'm sorry, this is not just a scheduling issue. I don't even know the semantics of this. So for example, if you're surfing the web and the phone rings do you really want to stop surfing the web and pick up the phone? I mean, what's so special about phone? There is nothing being indicated from the language point of view. It may be application requirement that you want to stop it or not stop it and so on. Is it a performance issue? That oh, it's because you have only 2 processors in this. If you had 4 this problem will go away. Or if you could just run it faster it will go away. I don't think any of these answers are right. I mean, the bottom line in this is sequential modules are often used in concurrent environments with unforeseen consequences. So your mindset has to be that I'm going to put together a parallel program with lots of existing things. Because you can't keep giving the answer oh, let's just rewrite it again from scratch. Because amount of software is so enormous in this. What we should worry about now is how we should design these modules. How we should write pieces of software or design hardware so that we can put together an ensemble of them, a collection of such modules so that they do something useful. And it's all about parallelism. And how to do all this in a parallel setting.

OK, so new goals as opposed to old goals. So I don't want to think in terms of decomposition. I don't want to think in terms of here is my clever algorithm, how do I decompose it so that this runs here and this runs here and so on. Instead I want to think in terms of synthesis. That you've already given me lots of modules, which are very useful. Which perhaps are written by domain experts. They know something about that stuff. And can I put together a bigger application very quickly from it? And you know, another favorite example of mine in this regard is FFTW and linear algebra packages. Both have been optimized to the hilt, right? And if somebody gives you an application which calls FFT and uses enough linear algebra I guarantee you that program will not be easy to write in spite of the existence of both these packages, which are extremely well optimized. So there is something we're not paying attention to is, how do we bring parallel modules together in such a way so that the functionality, the performance, et cetera are unpredictable in some sense?

AUDIENCE: I'm sorry. What's your point? I think I agree with your point, I just want to make sure I understand your point. That FFTW linear algorithm is not enough and so there will be--

ARVIND: No, no. I think it's very well done, FFTW. Linear algebra is very well done. But we still haven't provided any good way of thinking in terms of two parallel things. You know, what happens when they interact? So if my program is calling FFTW often and then it's calling linear algebra often.

AUDIENCE: You weren't here when I mentioned that but I was actually bringing up the point of linear algebra and FFTW working together. Just working.

ARVIND: Exactly. Absolutely. I'm saying in that sense, you see, this is a difference way of thinking because the old way of thinking well, give me your application. Now let's decompose it. The point is I don't have experts who can write FFTW as well as Matel wrote it. Or who's linear algebra packages are being written. I want to use those things. I want to synthesize large parallel programs quickly. Yes?

AUDIENCE: How does the Apple iPhone work with--

ARVIND: I'm sorry, how does?

AUDIENCE: The iPhone.

AUDIENCE: Apple iPhone.

ARVIND: Apple iPhone?

AUDIENCE: Yes, they have the operating system on there--

ARVIND: I think from our point of view that's the same thing. Is just that the user interface is guaranteed to be far superior because in my phone when I see the icons I'm a techie and I still can't understand half of them. So I like Apples idea. You know, phone, pictures.

[OBSCURED]

PROFESSOR: The way Apple might be with that is going to not let anybody program it. It's going to have a very restrictive development program internally so they can basically say, I'm going to take this person from this person and put it together. They will try to do everything in one unified way. So to some extent it might work, but then you suddenly realize Apple hired programmers to keep up with all the needs. So it may even start to have other people running things.

ARVIND: Internally their software is--

AUDIENCE: So is there something parallel going on in there?

ARVIND: Guaranteed there will be parallel stuff going on there. Guaranteed because all the communication stuff has to go on in parallel with the computation stuff. So I mean, phones have to lots of things in parallel if for no other reason just because of power. It takes less power when you do things in parallel as opposed to when you do sequentially because then you have to run much faster in that.

OK, so a method of designing and connecting modules has that functionality and performance are predictable. And must facilitate natural description of concurrent systems. A method of refining individual modules into hardware or software for systems on a chip. So refinement is a highly technical word. What that means is that you have written a program and you move on to rewrite it, but nobody can tell from outside that you rewrote it. All right, so you refine it and you can to refine it into hardware. I mean, you may implement it in hardware in this as opposed to transformation. Transformation is automatic. We're doing something, it's guaranteed it's correct. Refinement you may have to work a little bit more to show that it is correct in some sense. It's a correct refinement of whatever you were doing.

So basically, just to get the application going, which as Allen is pointing out is the major task and this is true regardless of what people say. When you get to complex system, if people start talking about performance you're 90% there. That means the damn thing works. So even the people may say, oh, performance will be lousy. Most of the energy is just consumed in getting it to work. So never forget that. So a refinement idea becomes very important because you want to take as many things as possible, which are available. Make it work and then individually refine; modular refinement of things.

AUDIENCE: Maybe just also underscore, just to emphasis what you're saying Arvind, our salesmen will go and say explicitly that they can sell a factor of 2. Just like you were saying. Just a factor of 2 on 100 processors or whatever, they can sell that if it works. If it gives the right answer.

AUDIENCE: It's the ease.

ARVIND: The ease and confidence that it's really doing what you expect it to do.

AUDIENCE: So it's really performance second, ease of use first. We've got it backwards in academia.

ARVIND: But everybody set designs on 2 multicores. So why multicore problems becomes hard and this where hardware and software starts diverging a little bit. So what happens in hardware design? Everything is in parallel. It's sitting right there, this blog does this, this blog does that. So there's no confusion in your mind what's going on in parallel. If you have to multiplex some thing in hardware, that first we'll do this then we'll do that, that's also expressed explicitly in the design. Software is not like that. You know, software have n threads, but I have only 10 in my hardware. So there is another layer of software or operating system or whatever you want to call it, runtime system. Whose sole job is to virtualize or to do time multiplexing of underlying resources and that makes the problem harder. You know, that makes the problem harder whether you're doing it in a way that preserves some performance goals. Whether you're doing it so that you want cause deadlocks, so there are many, many issues that come up in this. And basically, when I said this is an ideas kind of a talk, I do know how to do this and I still haven't worked enough on this problem of how to take this kind of methodology where I can write these parallel modules, compose them. How to map it onto multicores.

OK, so now let me tell you something technical about this stuff. So this hardware inspired methodology for synthesizing parallel programs. Now I'm going to use some fancy words. So this is a rule-based specifications or what I call guarded atomic actions. I will explain in a second what that means. It will let you think about parallel systems in a very different way. So it's like saying look, if this register is 2 and this register is 3, you can add them and put them here, 5. And you can do this any time you want. So this is like a rule. Invariant. You're saying, I don't care what's happening in the machine. This is always safe to do.

Now what I'm going to ask you to do is take a huge intellectual jump from this. That if you give me enough such rules you would have completely described the hardware, what it does. Now why am I thinking like this? Because these invariants are stated properly. It's always possible to understand the behavior of the system as if one things at a time is happening. When 2 plus 3 is becoming 5, if you want you can have this mental picture-- the rest of the world is frozen. Nothing is changing there, only that change is taking place. And you can think it terms of these small, small changes and all legal behaviors will be explainable as some sequence of these changes. It's possible to describe all kinds of hardware using things like that. Composition of modules with guarded interfaces. So this is the language called Bluespec and it has a very strong echoes of a language that was designed in the 80s by two guys, Chandry and Misra, the language called Unity. But they never used it for hardware design. Their goals were different and they didn't really have a concept of a module in that language.

So let me show you 3 examples here. Greatest common divisor, just to get off ground and then I will show you 2 very different problems: airlines reservation query problem and the video codec H.264. We won't have time to get into ordered list, but if somebody wants to discuss that I've be happy to show you. OK, so now let's look at how do I think of my design of my program in Bluespec? I always think in terms of a bunch of modules. And what's a module. A module is going to have some state elements in it. If you're thinking software just think each module guards some variables, which nobody else can touch. You're the only one. So you have a variable xyz, he has some abcd, et cetera. So each module has some. That's what I'm showing in red. If you were thinking hardware, think of thse things as stateful elements like registers or flip flops of memories. But the main point is if it is here then it's not here. You own something, this module owns something, this module owns something. One. Second thing is every module has internal rules for manipulating the state and the rules are always going to be of the form that if some condition is true on this you're allowed to make the following changes to that. So that's what the rule looks like. Some condition, if it holds that by action I mean, change the state. Change the state of those variables. And it's all or nothing. So if you have 2 variables and if you want to change both of them in a rule then the execution of a rule means either both of them will change or neither will change. There's nothing in between you can see. That's disallowed by this model.

I mean, the reason I have modules is they're not totally disjoined. You can actually access the state. Both read it and write it, but you can't just arbitrarily reach it and do something. You have to go through some interface methods the moment you talk to some other modules. So this is the standard information hiding principle. You know, abstract data types, whatever you want to call it. You know, so there's an interface through which you will enter and if you're just reading the values I think we call them read methods or value methods and if you're going to effect the state of one of the modules we'll call those action methods. Because you're actually going to change the state in some module.

So now, very strange execution model here. Very, very different from sequential execution model. So repeatedly you do the following. You pick a rule to execute. Any rule, I don't care. And select a rule to execute, compute the state, what updates should be made, make the state update. And then go and do it again, so repeatedly you do this. So this is a highly nondeterministic thing, selective rule. There may be a gazillion rules in your system. So system privileges are provided to control the selection if you want. But it's a detail that we won't get into to. So does everybody get the model right here? That you have a lot of state elements and you have lot of rules. Many rules may be applicable, pick one of them to execute. Execute and that will change the state. Ask the same question again, which rules are enabled? No, pick any rule amongst them and keep doing it.

Now semantics say one rule at a time. In any implementation we're going to do many, many rule in parallel. But you don't have to worry about that. That will be done all automatically. That we can do many rules in parallel. OK, so now let's look at GCD. So this is an ordinary GCD program. You can write it in any language you want if y is zero then x otherwise if this is greater than GCD of y, x. Otherwise you subtract. So no problem and I'm sure you know how it executes. So if I was to take GCD of 6,15 you will see there is a recursive call that'll go on because 6 is less than 15 so it will be this. You get this and you get this, et cetera. Ultimately you get 3 as an answer. Everybody is with me? You know you all know GCD? You can all write it in your favorite language.

Now the question is, what does it mean to execute this program in a concurrent setting? We were not thinking parallel or sequential, right? This is GCD. Someone who's teaching this class says run it in parallel. You say, what do you mean, run it in parallel? Oh, you mean that maybe I should try to do several GCD calls, recursive calls in parallel? That also doesn't make too much sense in this. Perhaps, this is what someone meant. You know that if he has some program where there are 2 calls to GCD, he would like you to do those 2 in parallel. I mean, what does it mean to do GCD in parallel?

Now let me contrast this with how you would think about this problem as a hardware person. So your job is to build a GCD machine. You're going to make millions of dollars, GCD is very popular. So you build this GCD machine. You know, it has 2 inputs, it has an output. And this will be my module I'm going to sell this intellectual property to everyone. OK, so now let me talk of parallel invocations of this. You see in hardware there can be no confusion, either you have 2 GCD boxes or you have 1 GCD box. I mean, you can have as many boxes as you want, but there is no confusion about that. You know how many GCDs you have. So in some sense, if you have many of them, you're talking of independent calls. Who knows what about recursive calls? I mean, that's internal story. You know, that's a different level of questions. So this question will automatically get asked.

In hardware you will ask the question, does the answer come out immediately? Does it come out in particular time? Why's this question important? Because if it's going to take some time then while it's computing the question I'm going to ask is can I give another set of inputs? Are you with me? I mean, that's a legitimate question to ask. If it's going to take half an hour to compute the GCD can I give it another input while it's thinking? Can the machine be shared by 2 different users? Can it be pipelined? So you agree that all these questions are meaningful in hardware setting? And I claim all these questions are also meaningful in software setting. We just don't think like that. This is exactly the problem we are having right now when we say that we have optimized something and we call it again. We think off it as starting a fresh, maybe not. Maybe not. You know, maybe we should think of it in terms of resources. And the point I want to make is I want to think of GCD module as a resource. Even in software I want to think of it as a resource. So that there is no ambiguity in your mind whether you have 1 GCD or you have 2 GCDs. If you're going to multiplex it we'll write it differently. If you're going to have 2 independent ones, which can go on in parallel we'll write it differently.

OK, so that's the idea I want to borrow from the hardware side of this. So how would you do-- yes?

AUDIENCE: But how would you decide how many GCDs to instantiate? Would that be is this model left up to the programmer?

ARVIND: Well, you see that's the big difference between hardware and software. That in hardware you have no design until you have taken that decision.

AUDIENCE: Right. I mean, hardware designs once it's built, it's built. Software designs you want the same module of software to seamlessly run on different kinds of hardware.

ARVIND: That doesn't mean that I can't recompile it or can't resynthesize it or something. So it may be same source description in software.

AUDIENCE: Right, if your source subscription specifies 2 GCD modules and you have 8 cores, you're not necessarily going to take very good advantage of this, so how do you--

ARVIND: No, no. The way I would like to think about that is that it'll be highly parameterized code and you will plug in some information like that and resynthesize the parallel program.

AUDIENCE: So the software or some sort of macro meta software has to determine the extent to which the problem should be distributed.

ARVIND: Well, you can think like that, but really this integration will be much tighter than you think. So for example, let me just give you some more insight from the hardware side. So in the hardware side it's standard practice that will have some RTL code, I'll have some code. And I plug in the library. What kind of gates have I got? Have I got 2 input gates, 4 input gates, whatever. There will be a huge library. The same code can be compiled using different libraries. So you choose something at synthesis time. Another thing that'll happen in hardware is what is called generic statements. So you may have a loop which assigns a register file with n registers. Which will conceptually makes sense, it will make sense in simulation, but when it comes to synthesizing it you have to specify n. So I'm going towards that kind of methodology. That you will have highly parameterized code, but many parameters you have to specify before you will say, now my program is ready to run on this parallel machine. And this is totally unexplored. As I said, this is an idea level them being.

AUDIENCE: So this adds a problem that is not in traditional hardware synthesis, mainly picking all the n's. Ideally you wouldn't want to have to have the programmer to come up with creative algorithms because they're going to have lots and lots of .

ARVIND: Absolutely. We may have lots of defaults. We may other level of smart programs, which look at this and they instantly go and set many parameters. You know, so there will be a practical side, many issues like that. But main point I'm trying to make is that I really want to synthesize. I want to instantiate a program, synthesize a program for a given configuration of hardware.

Let's look at the GCD in Bluespec. If I'm going to find the GCD of 2 numbers I need 2 registers, x and y; so that's what this is. Make me a register. If you want, make me 2 variables. You know, make me variable x and y, initial values are zero. So this is what is called the state of the module. It knows about and x and y, nobody else knows about x and y. What are the dynamics of it? How do I compute this? So internal behavior is being described and this is exactly what you saw earlier in saying that there is a swap rule. If x is greater than y and y is not equal to zero then you swap x and y, so this is a parallel composition. x gets y, y gets x. In this system all the reads take place instantaneously and then you do all the writes at the end of it. Don't read this sequentially, you read x and y in parallel and then you go and update x and y. Does this rule make sense? I mean, if you know anything GCD-- if x is greater than y than you're going to swap it and subtraction, if x is less than or equal to y than y gets y minus x. So this is very interesting. Now you have 2 registers x and y and I've given 2 rules and I'm saying you can apply these rules anytime you want. Just repeatedly keep doing these rules.

Now what does the outside world know about this? What do you want to advertise about GCD to the outside world? You don't want to give away your secret. This is your intellectual property of how you're computing the GCD. You just want to be use GCD. So the outside world can say, oh find me start it. With a and b. x gets a, y gets b. It's going to give you 2 parameters, but we may be busy computing so we have a guard here which says, if y is zero-- this is internal, outside world doesn't see this. If y is zero then only then can you start it, otherwise you can't start it. Otherwise it means it's busy computing. OK, similarly when is the result available? When the y is zero then you have the result and you will return x.

AUDIENCE: [OBSCURED]

ARVIND: That's right. So that'll be more sophisticated. And that's exactly the kind of decision I do want the designers to take because you may want to put a five hole. You may just keep spitting out results. You can make it as sophisticated as you want. Tag it if you want. No predetermined thing.

So first question I ask you, what happened to those recursive calls? You know, if you go back to the previous slide there was a recursively called GCD, which you were all comfortable with. There is no GCD call here.

AUDIENCE: [OBSCURED]

ARVIND: Right, so there's a notion of a cycle event or something. Look at it now, update it, look at it again. You know, so there's like an infinite loop here, which is always going. Which is exactly how hardware works. It doesn't have to be synchronous, but if you want to have a simple model in your head just think, there's the clock. You know, look at this state, pick one route to execute, update it, then it's the next clock cycle. Keep doing this repeatedly. It's details here, but I can take such a description and actually produce that machine for you, which does the GCD. What I'm much more interested in is, this is how I want to think about my GCD now. So I've hidden everything inside it. And it has 2 methods: start and result and there is a very strong notion of when a method is ready-- because it may be busy computing, it doesn't want to listen to you. So there is this notion of something being ready and if it's an action method then you're only allowed to enable it when it's ready. So this is a high-level protocol that compiler will enforce. It'll never ever set this, it'll never execute this unless it is ready to be executed. And of course, when you're going to enable it then you have to give me 2 arguments that go with it. And what does this really mean? The result is valid. So if are to do types et cetera, this would easily be baby type or something. Tag union type that validates is being coded here.

So really to the world you are just advertising this interface. There is a GCD interface and it has 2 methods. An action method, which is called start and a result method, which is just a value kind of a thing. And in order to do this you have to give me an int a into b in this. An end of interface. Now this is borrowing a lot from modern programming languages so for example, I can easily make it polymorphic. It doesn't have to be ints here. You know, what is the meaning of int? You can specify the type in this. So it'll be a's of type t, b's of type t et cetera. What are the examples of type t here? It could be a 32 bit integer, it could be 16 bit integer, it could be 17 bit integer, whatever you want and whatever makes sense. So this is the kind of thing that you always take care of when you synthesize things. It won't synthesize until you specify the type, but the description remains the same. And you will instantiate it for a given type. This I'm showing you for one reason. If you are coming from the hardware side really these ideas and types and sensations are very sophisticated. I mean, this is getting as advanced as you can have in software.

The other very interesting thing is, and this is the abstract data type kind of thinking, you can go and completely change the implementation of this. If you have a clever algorithm for doing GCD, fine, go ahead and do it. You know, it doesn't effect the users of this. So this is a very important property for composition. All I insist on is that every method has this ready thing coming on so that I don't make mistakes in wiring it. I will involve this method only when it's ready. So in some sense if you forget about the ready thing, I'm just borrowing ideas from object oriented languages. You know, you have a class, you have methods on it, and all I'm saying is oh, you should manipulate this state in a very systematic manner using these methods. But then I'm injecting some hardware idea here, which doesn't exist in software today. That is, oh, a method may not be ready. And even that can be captured very abstractly and done properly in this. Yep?

AUDIENCE: So then in the case of making models I was thinking of spy nature. So if you want to basically synthesize the hardware you may also better efficiency by having set of synchronized protocol by taking the output [OBSCURED] number of cycles.

ARVIND: I think, yes. That'll be a low-level detail that if it is synchronized then you will actually try to manipulate it like that.

AUDIENCE: [OBSCURED]

ARVIND: OK, right. So let me just quickly say what the languages is. So you have modules, then you have state variables, and you have rules and you have action methods and read methods. What I wanted to show you is what is the language of actions. So when I write an action here or an action here, what is it? So simplest action is assignment to a variable, assigment to a register. This is a conditional action, so if predicate is true then do this action, otherwise no action, no change in state. This is a parallel composition. Do a1 and a2 in parallel. Sequential composition, the effects of this are visible to this, et cetera. This is a call to a method of one of the module. And then there is a guard. And you have an expression language, which is there's nothing special here. So this is just simply you can read a variable, you can have constants. These are just names and there's nothing special going on here.

Let me explain you guards. So people find guards versus if's confusing. And this is one way to understand it. So guards affect the surroundings. If I wrote here a1 when p1 in parallel with a2 think of guard as something to do with resources. I really don't want you to do a1 unless p1 is true. p1 made a fact that the module was busy. Some predicate here. But I want the effect of this whole thing to be atomic. So if any part of it can't be done then the whole thing can't be done. And therefore the affect of guard is as if you wrote a1 in parallel with a2 when p1. In other words, guard on anything becomes guard on everything in that parallel composition. Do you understand what I just said? This is very important for composition of guards. Because I want the atomicity of the whole thing to be preserved. So if anything can't be done that means the whole thing can't be done. On the other hand, conditional action is just conditional action. So if I have if p1 than a1 in parallel with a2, well that's like saying if p1 was true then I want to do a1 and a2 in parallel, otherwise I just want to do a2. So there is a very big difference between conditional actions and guards. Guards are something about resources. I need it for proper composition of atomic actions.

AUDIENCE: I have question. So are they entirely equal into. Because this is maybe a low-level question, but in the first case-- so the first case in the left. Can't you start doing a2 and then roll it back or something [OBSCURED]

ARVIND: Oh, I think there may be many tricks you can play in implentation. You can be optimistic, you can--

[INTERPOSING VOICES]

ARVIND: No. No. No. Semantically this is this defining the semantics. I mean, this is the algebra. This has to be true because I'm saying this is what a guard is.

AUDIENCE: Right, so the semantically equal but not [OBSCURED].

ARVIND: Well I mean, that's probably not the right way to say it. I mean, semantics are being defined. You have choice in implementations. I mean I'm not telling you how to implement this. Let me go on with this.

So here is a problem that was posed to me by Jayadev Misra. This is quasi realistic problem. Ask for codes from 2 airlines. If one code is below $300, buy immediately. $300 is sort of the cheapest ticket you get these dates. Buy the lower quote if over $300. But as some-- say your patience runs out, say I can't wait anymore. So after one minute buy from whosoever has quoted otherwise flag error. Is this a realistic scenario? You can express it using threads. And you can write a scheduler to do this, et cetera. Those things are not as succint as you would like. I mean, you'd be surprised, in this simple a problem how complicated and how many questions will arise if you express this as a threaded computation. Now let me show you what you will do-- in Bluuspec.

OK, so we are going to make a module, which does what? Make get quotes. This module's job is to get quotes. So what kind of state elements do you expect it to have? Well airline a may be quoting some value, airline b may be quoting some value. Whether we are done or not there's a timer, which we're going to be bumping, et cetera. And they'll be rules you know, get a quote from a, which will be when not done do something. Executes when a responds. Get b, timeout, et cetera. There are there ruls of this sort. Let me just show you one methodd.

So method, this is how you will start it-- you have this module get quotes and you're saying book me a ticket and this is your request r. Now obviously this can only be done when the module is not busy. If module is busy the you can't do it. So what will you do when you get such a request? Tell me this makes sense. So what is this saying? Try to get a request from a in parallel. Try to get a request from b. Now you are busy. The module is busy so done equals false here. And I'm initializing this. I'm just saying that a quote right now is infinity and b quote is infinity and timer is zero. Fair? And when will I get ticket? Well, when done then you'll return the ticket from this.

Now let's look at example of a rule. So you may have a rule like this, pick cheapest. You see the interesting thing in this is you'll be able to read this independently of what else is going on in this system. And let's see, what does this rule say? Is it succint? Rules says when you are not done and if the quote from a is not infinite that means a has quoted. And b has also quoted, then what should happen? If a is less than b then buy the ticket from a. Otherwise you'll buy the ticket from b and you're done. I'm not going to write all these rules, but I think you should be able to-- I'm certain you can write these rules. What's happening here that is just separation of concerns. It's all concurrenct stuff. Yes?

AUDIENCE: [OBSCURED]

ARVIND: Parallel?

AUDIENCE: Or like, for lack of better words the type in the name.

ARVIND: Sorry, say it again.

AUDIENCE: So as the ampersand, how does that interact with the--

PROFESSOR: Can you hold that question first because we will run out of space.

AUDIENCE: So the ampersand after not done that is, what sort of ordering is that?

ARVIND: It's just, I mean, associative, commutative. Kind of thing. It's all happening in parallel.

AUDIENCE: [OBSCURED].

ARVIND: This thing, right? I mean, think of it like this, these are variables and you're checking their values right now. So it's just a combination kind of a query that you have here.

AUDIENCE: So I guess you're checking there's new activity happening.

ARVIND: That's right.

AUDIENCE: Doesn't matter which order you check.

AUDIENCE: So I guess what I'm trying to get is that the whole block at the top is executed in parallel with the block at the bottom?

ARVIND: No. Those things are not in parallel.

AUDIENCE: [UNINTELLIGIBLE PHRASE]

ARVIND: Now I just want to show you this because it's completely different from airline reservations. You know, H.264 this is a codec that is being used everywhere and we'll be used even more. And this is a typical data flow diagram you will find for this kind of thing. You don't have to understand anything about this except that conceptually you just have five flows for communicating between them and in all such things the goal is that it has to be able to sustain a certain amount of rate at which dat is to be processed and it's usually forgiving in terms of latencies. So if latency increases a little bit here and there it's usually OK-- when things go in and come out. A data flow like network. And another reason why this example is fascinating is because it's done regularly both in hardware and software and any mixture of the two. So for example, when it runs on your PC it's being done 100% in software and I think on iPods there are special purpose hardware for doing it and therefore the battery can last much longer. And on cell phones it's kind of in between in this and it's a question of what frame rate you want to process? If you're procssing very high frame rate then it'll have to be done in hardware.

So when we got into this thing we said, OK, we wanted to build hardware for doing this. And reference codes that are available, it's 80,000 lines and it sort of gives you a heart attack. You know, you read this and you say, whoa! And it doesn't need the performance. I mean, that reference code, if it ran on your laptop you won't be able to watch a movie with this. And there is the Linux version of it, which gives you even a bigger heart attack because it's 200,000 lies and many different codecs are mixed in this. The biggest problem here it is none of these codes reflect that picture I showed you of the previous diagram. Because the way people approach it in software is they take some input from here and they push it as far as they can. And that's it. Then they will take the next input and push it. So different boxes keep modifying it as they see fit and as a reader you can't tell, oh, this is this part and this is this part. I mean, there are no 54Q's. Why? Because 54Q's are expensive in software. You know, it will involve copying. You know, you'll write it here then you'll read it again, which is a bad idea in software. So in some sense, the software will thinking already entered. In this style of coding there is no model of concurrency. So it's not clear to me when you give me that quote what all I can do in parallel in order to preserve the semantics of this. I mean the problem is much, much harder here. So you will just do some high-level analysis of the code, but you have lost a lot of information that was all present in the source-- could've been present in the source. And my claim is it can be done differently.

So this has been done by Chun-Chieh and this I'm showing you a month old slide, but he has done a lot more work since then and the total code in Bluespec is 9000 lines for this contrasting it with 80,000 versus 200,000 kind of stuff and this is telling you the amount of code in various blocks in this. And since it's done in Bluespec you can take it and you can go and implement it all in hardware, so you can synthesize it. It actually does now 45 frames per second, actually 90 frames even per second today. You know, so this is a month old. This is the main point. Once you have done this now you can refine it. You have a different preference. You say, oh, I want to do this differently. So you can go and rewrite any module again. It will be very, very easy to do this in the system. It is very difficult to do this in the reference code, so that's one point to be made. The second thing is you don't have to do everything in hardware. If you wanted you could have implemented any part of it in software, so you can take the same kind of a description and generate software from it as well. So each module can be refined separately. Behaviors of modules are composable. You know, you can take these things and it's predictable what will happen when you compose these things in terms of performance and in terms of resources also, it's very clear what's happening. In hardware the resources will be area and time and so on in this.

So takeaway. Parallel programming should be based on well-defined modules and parallel composition of such modules. Modules must embody a notion of resources and consequently, sharing in time multiplex use. This is a controversial point. And guard atomic actions and modules with guarded interfaces provide a solid foundation for doing so.

PROFESSOR: Thank you. Now we ran a little bit late, so if you have questions you can ask about programs.