PROFESSOR: So we'll get started. So today we are going to dive into some parallel architectures.

So the way, if you look at the big world, is there's -- just counting parallelism, you can do it implicitly, either by hardware or the compiler. So the user won't see it. It will be done behind the user's back, but can be done by hardware or compiler. Or explicitly, visible to the user.

So the hardware part is done in superscalar processors, and all those things will have explicitly parallel architecture. So what I am going to do is spend some time just talking about implicitly parallel superscalar processors. Because probably the entire time you guys were born till now, this has been the mainstream, people are building these things, and we are use to it. And now we are kind of doing a switch. Then we'll go into explicit parallelism processors and kind of look at different types in there, and get a feel for the big picture.

So let's start at implicitly parallel superscalar processors. So there are two types of superscalar processors. One is what we call statically scheduled. Those are kind of simpler ones, where you use compiler techniques to figure out where the parallelism is. And what happens is the computer keeps executing, intead of one instruction at a time, the few instructions next to each other in one bunch. Like a bundle after bundle type thing.

On the other hand, dynamically scheduled processors -- things like the current Pentiums -- are a lot more complicated. They have to extract instruction level parallelism. ILP doesn't mean integer linear programming, it's instruction level parallelism. Schedule them as soon as operands become available, when the data is able to run these instructions. Then there's just a bunch of things that get more about parallelism, things like rename registers to eliminate some dependences. You execute things out of order. If later instructions the operands become available early, you'll get those things done instead of waiting. You can speculate to execute. I'll go through a little bit in detail to kind of explain what these things might be.

Why is this not going down. Oops.

So if you look at a normal pipeline. So this is a 004 type pipeline. What I have is a very simplistic four stage pipeline in there. So a normal microprocessor, a single-issue, will do something like this. And if you look at it, there's still a little bit of parallelism here. Because you don't wait till the first thing finishes to go to the second thing.

If you look at a superscalar, you have something like this. This is an in-order superscalar. What happens is in every cycle instead of doing one, you fetch two, you decode two, you execute two, and so on and so forth. In an out-of-order super-scalar, these are not going in these very nice boundaries. You have a fetch unit that fetches like hundreds ahead, and it keeps issuing as soon as things are fetched and decoded to the execute unit. And it's a lot more of a complex picture in there. I'm not going to show too much of the picture there, because it's a very complicated thing.

So the first thing the processor has to do is, it has to look for true data dependences. True data dependence says that this instruction in fact is using something produced by the previous guy. So this is important because if the two instructions are data dependent, they cannot be executed simultaneously. You to wait till the first guy finishes to get the second guy. It cannot be completely overlapped, and you can't execute them in out-of-order. You have to make sure the data comes in before you actually use it.

In computer architecture jargon, this is called a pipeline hazard. And this is called a Read After Write hazard, or RAW hazard. What that means is that the write has to finish before you can do the read. In a microprocessor, people try very hard to minimize the time you have to wait to do that, and you really have to honor that.

In hardware/software what you have to do is you have to preserve this program ordering. The program has to be executed sequentially, determined by the source program. So if the source program says some order of doing things, you better -- if there's some reason for doing that, you better actually adhere to that order. You can't go and just do things in a haphazard way.

And dependences are basically a fact of the program, so what you got. If you're lucky you'll get a program without too many dependences, but most probably you'll get programs that have a lot of dependences. That's normal.

There's a lot of importance of the data dependence. It indicates the possibility of these hazards, how these dependences have to work. And it determines the order in which the results might be calculated, because if you need the result of that to do the next, you have what you call a dependency chain. And you have to excute that in that order. And because of the dependency chain, it sets an upper bound of how much parallelism that can be possibly expected. If you can say in all your program there's nothing dependent -- every instruction just can go any time -- then you can say the best computer will get done in one cycle, because everything can run. But if you say the next instruction is dependent on the previous one, the next instruction is dependent on the previous one, you have a chain. And no matter how good the hardware, you have to wait till that chain finishes. And you don't get that much parallelism.

So the goal is to exploit parallelism by preserving the program order where it affects the outcome of the program. So if we want to have a look and feel like the program is run on a nice single-issue machine that does one instruction after another after another, that's the world we are looking in. And then we are doing all this underneath to kind of get performance, but give that abstraction.

So there are other dependences that we can do better. There are two types of name dependences. That means there's no real program use of data, but there are limited resources in the program. And you have resource contentions. So the two types of resources are registers and memory.

So linear resource contentions. The first name dependence is what we call anti-dependence. Anti-dependence means that -- what I need to do is, I want to write this register. But in the previous instruction I'm actually reading the register. Because I'm writing the next one, I'm not really using the value. But I cannot write it until I have read that value. Because the minute I write it, I lose the previous value. And if I haven't used it, I'm out of luck. So there might be a case that I have a register, that I'm reading the register and rewriting it some new value. But I have to wait till the reading is done before I do this new write. And that's called anti-dependence. So what that means is we have to wait to run this instruction until this is all -- you can't do it all before that.

So this is called a Write After Read, as I said, in the architecture jargon.

The other dependences have what you call output dependence. Two guys are writing the register, and then I'm reading it. So I want to read the value the last guy wrote. So if I reorder that, I get a wrong value. Actually you can even do better in here. How can you do better in here?

AUDIENCE: You can eliminate I.

PROFESSOR: Yeah. You can elimiate the first one, because nobody's using that value. So you can go even further and further, but this is also a hazard. This is called a Write After Write hazard. And the interesting thing is by doing what you call register renaming, you can eliminate these things. So why do both have to use the same register? In these two, if I use a different register I don't have that dependency. And so a lot of times in software, and also in modern superscalar hardware, there's this huge amount of hardware resources that actually do register renaming. So they realized that anti-dependence is output dependent, and said -- "Wait minute. Why do I even have to do that? I can use a different register." So even though you have -- Intel basically [UNINTELLIGIBLE] accessory only have eight registers. They are about 100 registers behind. Hardware registers just basically let you do this reordering and renaming -- register renaming.

So the other type of depencence is control dependence. So what that means is if you have a program like this, you have to preserve the program ordering. And what that means is S1 is control dependent on p1. Because depending on what p1 is, it will depend on this one to get excuted. S2 is control dependent on p2, but not p1. So it doesn't matter what p1 does, S2 will execute only if p2 is true. So there's a control dependence in there.

Another interesting thing is control dependence may -- you don't need to preserve it all the time. You might be able to do things out of this order. Basically, what you can do is if you are willing to do more work, you can say -- "Well, I will do this. I don't know that I really need it, because I don't know whether the p2 is true or not. But I'll just keep doing it. And then if I really wanted, I'll actually have the results ready for me." And that's called speculative execution. So you can do speculation. You speculatively think that you will need something, and go do it.

Speculation provides you with a lot of increased ILP, because it can overcome control dependence by executing through branches, before even you know where the branch is going. And a lot of times you can go through both directions, and say -- "Wait a minute, I don't know which way I'm going. I'll do both sides." And I know at least one side you are going, and that will be useful. And you can go more and more, and soon you see that you are doing so much more work than actually will be useful.

So the first level of speculation is -- speculation basically says, you go, you fetch, issue, and execute everything. You do the end of the thing without just committing your weight into the commit to make sure that the right thing actually happened. So this is the full speculation.

There's a little bit of less speculation called dynamic scheduling. If you look at a microprocessor, one of the biggest problems is the pipeline stall is a branch. You can't keep even a pipeline going, even in a single-issue machine, if there's a branch, because the branch condition gets resolved. Not after the next instruction has to get fetched. So if you do a normal thing, you just have to reinstall the pipeline.

So what dynamic scheduling or a branch predictor sometimes does is, it will say I will predict where the branch is going. So I might not have fed board direction, but I will speculatively go fetch down one path, because it looks like it which it's going. For many times, like for example in a loop, 99% of the time you are going in the backage, because you don't go through that. And then if you predict that you are mostly [UNINTELLIGIBLE]. So the branch predictors are pretty good at finding these kind of cases. There are very few branches that are kind of 50-50. Most branches have a preferred path. If you find the preferred path you can go through that, and you don't pay any penalty. The penalty is if you made a mistake, you had to kind of back up a few times. So you can at least do in one direction. Most hardware do that, even the simplest things do that. But if you do good speculation you go both. You say -- "Eh, there's a chance if I go down that path I'm going to lose a lot. So I'll do that, too." So that does a lot of expensive stuff.

And basically this is more for data flow model. So as soon as data get available you don't think too much about control, you keep firing that.

So today's superscalar processors have huge amount of speculation. You speculate on everything. Branch prediction. You assume all the branches, multilevel down you predict, and go that. Value prediction. You look at it and say -- "Hey, I think it's going to be two." And in fact there's a paper that says about 80% of program values are zero. And then you say -- "OK. I'll think it's zero, and it'll go on. And if it is not zero, I'll have to come back and do that." So things like that.

AUDIENCE: Do you know what percentage of the time it has to go back?

PROFESSOR: A lot of times I think it is probably an 80-20 type thing, but if you do too much you're always backing up. But you can at least do a few things down assuming it's zero. So things like that. People, try to take advantage of the statistical nature of programs. And you are mining every day. So basically there's no -- it's almost at the entropy. So every information is kind of taken advantage in the program, but what that means is you are wasting a lot of time cycles. So the conventional wisdom was -- "You have Moore's slope. You keep getting these transistors. There's nothing to do with it, so let me do more other work. We'll predicate, we'll do additional work, we'll go through multipe branches, we'll assume things are zero. Because what's wasted? Because it's extra work, if it is wrong we just give it up."

So that's the way it went, and the thing is it's very inefficient. Because a lot of times you are doing -- think about even a simple cache. If you have 4-way as a cache. Every cycle when you're doing a memory fetch, you are fetching on all four, assuming one of it will have hit. Even if you have a cache hit where only one bank is hit, and all the other three banks are not hit. So you are just doing a lot more extra work just to get one thing. Of course because if you wait to figure out which bank, it's going to add a little bit more delay. So you won't do it parallelly. You know that's it's going to be one of the lines, so you just go fetch everything and then later decide which one you want.

So things like that really waste energy. And what has been happening in the last 10 years is you double the amount of transistors, and you add 5% more performance gain. Because statistically you have mined most of the lower hanging fruit, there's nothing much left. So you're getting to a point that has a little bit of a statistical significance, and you go after that. So of course, most of the time it's wrong.

So this leads to this chart that actually yesterday I also pointed out. So you are going from hot plate to nuclear reactor, to rocket nozzle. We tend to be going in that direction. That is the path, because we are just doing all these wasteful things. And right now, the power consumption on processors is significant enough in both things like laptops -- because the battery's not getting faster -- as well as things like Google. So doing this extra useless work is actually starting to impact.

So for example, if you look at something like Pentium. You have 11 stages of instructions. You can execute 3 x86 instructions per cycle. So you're doing this huge superscalar thing, but something that had been creeping in lately is also some amount of explicit parallelism. So they introduced things like MMX and SSE instructions. They are explicit parallelism, visible to the user. So it's not hiding trying to get parallelism. So we have been slowly moving to this kind of model, saying if you want performance you have to do something manual. So people who really cared about performance had to deal with that.

And of course, we put multiple chips together to build a multiprocessor -- it's not in a single chip -- that actually do parallel processing. So for about three, four years if you buy a workstation it had two processors sitting in there. So dual processor, quad processor machines came about, and people started using that. So it's not like we are doing this shift abruptly, we have been going that direction. For people who really cared about performance, actually had to deal with that and were actually using that.

OK. So let's switch gears a little bit and do explicit parallelism. So this is kind of where we are -- where we are today, where we are switching. So basically, these are the machines that parallelism exposed to software -- either compiler. So you might not see it as a user, but it exposes some layer of software. And there are many different forms of it. From very loosely coupled multiprocessors sitting on a board, or even sitting on multipe machines -- things like a cluster of workstations -- to very tightly coupled machines. So we'll go through, and figure out what are all the flavors of these things.

AUDIENCE: Excuse me.

PROFESSOR: Mhmm?

AUDIENCE: So does it mean that since there being the level parallelism, the processor can exploit the fact that the compiler knows the higher level instructions? Does that make any difference?

PROFESSOR: It goes both ways. So what the processor knows is it know values for everything. So it has full exact knowledge of what's going on. Compiler is an abstraction. In that sense, processor wins in those. On the other hand, compile time doesn't affect the run time. So the compiler has a much bigger view of the world. Even the most aggressive processor can't look ahead more than 100 instructions. On the other hand, the compiler sees ahead of millions of instructions. And so the compiler has the ability to kind of get the big picture and do things -- global kind of things. But on the other hand, it loses out when it doesn't have information. Whereas when you do the hardware parallelism, you have full information.

AUDIENCE: You don't have to give up one at the loss of the other.

PROFESSOR: The thing is, I don't think we have a good way of combining both very well. Because the thing is, sometimes global optimization needs local information, and that's not available at run time. And global optimization is very costly, so you can't say -- "OK, I'm going to do it any time." So I think it's kind of even hybrid things. There's no nice mesh in there.

So if you think a little bit about parallelism, one interesting thing is this Little's Law. Little's Law says parallelism is a multiplication of throughput vs. latency. So the way to think about that is the parallelism is dictated by the program in some sense. The program has a certain amount of parallelism. So if you have a thing that has a lot of latency to get to the result, what that means is there's a certain amount of throughput you can satisfy. Whereas if you have a thing that has a very low latency operation, you can go much wider. So if you look at Intel processors, what they have done is the superscalars -- they have actually, to get things faster they have a very long latency. Because they know they couldn't go more than three or four wide. So they went like 55 the pipeline, three wide. Because you can go fast, so they assume the parallelism fits here. So still you need a lot of parallelism. So you say -- "Three, why should [UNINTELLIGIBLE] issue machine. [UNINTELLIGIBLE] three it's no big deal." But no, if you have 55 the pipeline you need to have 165 parallel instructions on the fly any given time. So that's the thing. Even in the moder machine, there's about hundreds of instruction on the fly, because the pipeline is so large. So if you said 3-issue, it's not that. I mean this happens in there.

So this gives designers a lot of flexibiilty in where you are expanding. And in some ways you can have a lot -- there are some machines that are a lot wider, but the latency is -- For example, if you look at an Itanium. It's clock cycle is about half the time of the Pentium, because it has a lot less latency but a lot wider. So you can do these kind of tradeoffs.

Types of parallelism. There are four categorizations here. So one categorization is, you have pipeline. You do the same thing in a pipeline fashion. So you do the same instruction. You do a little bit, and you start another copy of another copy of another copy. So you kind of pipeline the same thing down here. Kind of a vector machine -- we'll go through categories that kind of fit in here.

Another category is data-level parallelism. What that means is, in a given cycle you do the same thing many many many many -- the same instructions for many many things. And then next cycle you do something different, stuff like that.

Thread-level parallelism breaks in the other way. Thread-level parallelism says -- "I am not connecting the cycles, they are independent. Each thread can go do something different."

And instruction-level parallelism is kind of a combination. What you are doing is, you are doing cycle by cycle -- they are connected -- and each cycle you do some kind of a combination of operations.

So if you look at this closely. So pipelining hits here. Data parallel execution, things like SIMD execution hits here. Thread-level parallelism. Instruction-level parallelism. So before a models of parallelism, what software people see kind of fits also in this architecture picture.

So when you are designing a parallel machine, what do you have to worry about? The first thing is communication. That's the begin -- the problem in here. How do parallel operations communicate the data results? Because it's not only an issue of bandwith, it's an issue of latency. The thing about bandwith is that had been increasing by Moore's Law. Latency, speed of light. So as I pointed out, there's no Moore's Law on speed of light, and you have to deal with that.

Synchronization. So when people do different things, how do you synchronize at some point? Because you can't keep going on different paths, at some point you have to come together. What's the cost? What are the processes of going it? Some stuff it's very explicit -- you have to deal with that. Some machines it's built in, so every cycle you are synchronizing. So sometimes it makes it easier for you, sometimes it makes it more inefficient. So you have to figure out what is in here.

Resource management. The thing about parallelism is you have a lot of things going on, and managing that is a very important issue. Because sometimes if you put things in the wrong place, the cost of doing that might be much higher. That really reduces the benefit of doing that.

And finally the scalability. How do you build process that not only can do 2x parallelism, but can do thousand? How can you keep growing with Moore's Law. So there are some ways you can get really good numbers, small numbers, but as you go bigger and bigger you can't scale.

So here's a classic classification of parallel machines. This has been [? divided ?] up by Mike Flynn in 1966. So he came up with four ways of classifying a machine. First he looked at how instruction and data is issued. So one thing is single instruction, single data. So there's single instruction given each cycle, and it affects single data. This is your conventional uniprocessor.

Then came a SIMD machine -- single instruction, multiple data. So what that means is the given instruction affects multiple data in here. So things like -- there are two types, distributed memory and shared memory. I'll go to this distinction later. So there are a bunch of machines. In the good old times this was a useful trick, because the sequencer -- or what ran the instructions -- was a pretty substantial piece of hardware. So you build one of them and use it for many, many data. Even today data in a Pentium if you are doing a SIMD instruction, you just issue one instruction, it affects multiple data, and you can get a nice reuse of instruction decoding. Reduce the instruction bandwidth by doing SIMD.

Then you go to MIMD, which is Multiple Instruction, Multiple Data. So we have multiple instruction streams each affecting its own data. So each data streams, instruction streams separately. So things like message passing machines, coherent and non-coherent shared memory. I'll go into details of coherence and non-coherence later. There are multiple categories within that too.

And then finally, there's kind of a misnomer, MISD. There hasn't been a single machine. It doesn't make sense to have multiple instructions work on single data. So this classification, right now -- question?

AUDIENCE: I've heard that [INAUDIBLE]

PROFESSOR: Multiple instruction, single data? I don't know. You can try to fit something there just to have something, but it doesn't fit really well into this kind of thinking.

So I don't like that thinking. I was thinking how should I do it, so I came up with a new way of classifying. So what my classification is, what's the last thing you are sharing? Because when you are running something, if it is some single machine, some thing has to be shared, and some things have to be separated. So are you sharing instructions, are you sharing the sequencer, are you sharing the memory, are you sharing the network? So this kind of fits many things nicely into this model. So let's go through this model and see different things in there.

So let's look at shared instruction processors. So there had been a lot of work in the good old days. Did anybody know Goodyear actually made supercomputers? Not only did they make tires, for a long time they were actually making processors. GE made processors, stuff like that. And so a long time ago this was a very interesting proposition, because there was a huge amount of hardware that has to be dedicated to doing the sequence and running the instruction. So just to share that was a really interesting concept. So people built machines that basically -- single instruction stream affecting multiple data in there. I think very well-known machines are things like Thinking Machines CM-1, Maspar MP-1 -- which had 16,000 processors. Small processors -- 4-bit processors, you can only do 4-bit computation. And then every cycle you can do 16,000 of them, 4-bit things in here. It really fits in to the kind of things they could build in hardware those days. And there's one controller in there. So it is just a neat thing, because you can do a lot of work if you actually can match it in that form.

So the way you look at that is, to run this array you have this array controller. And then you have processing elements, a huge amount of them. And you have each processor mainly had distributed memory -- each has its own memory. And so given instruction, everybody did the same thing to memory or arithmetic in there. And then you had also interconnect network, so you can actually send it. A lot of these things have the [? near-enabled ?] communication. You can send data you near enable, so everybody kind of shifts the 2-D or some kind of torus mapping in there. And if you can program that, you can get really good performance in there.

And each cycle, it's very synchronous. So each cycle everybody does the same thing -- go to the next thing, do the same thing.

So the next very interesting machine is this Cray-1. I think this is one of the first successful supercomputers out there. So here's the Cray-1, it is this kind of round seat type thing sitting in here. Everybody know what was under the seat?

AUDIENCE: Cooling.

PROFESSOR: Cooling. So here's a photo. I don't think you can see that -- you can probably look at it when I put this on the web -- was this entire cooling mechanism. In fact Seymour Cray at one time said one of his most important innovations in this machine is how to cool the thing. And this is a generation, again, that power was a big thing. So each of these columns had this huge amount of boards going, and in the middle had all the wiring going. So we had this huge mess of wiring in the middle -- [UNINTELLIGIBLE] -- and then you had all these boards in there in each of these. So this is the Cray-1 processor.

AUDIENCE: Do you know your little -- your laptop is way faster than that Cray --

PROFESSOR: Yeah. Did you have the clock speed in here?

[INTERPOSING VOICES]

AUDIENCE: 80 MHz.

PROFESSOR: So, yeah. And that cost like $10 million or something like that at that time. Moore's Law, it's just amazing. If you think if you apply Moore's Law to any other thing we have, it can't do the comparison. We are very fortunate to be part of that generation.

AUDIENCE: But did it have PowerPoint?

PROFESSOR: So what it had, was it had these three type of registers. It had scalar registers, address registers, and vector registers. The key thing there is the vector register. So if you want to do things fast -- no, fast is not the word. You can do a lot of computation in a short amount of time by using the vector registers.

So the way to look at that is normally when you go to the execute stage you do one thing. In a vector register what happens is it got pipelined. So execute state happened one word next, next, next. You can do up to 64 or even bigger. I think it was 64 length, length 64 things. So you can -- so that instruction. So you do a few of these, and then this state keeps going on and on and on, for 64. And then you can pipeline in the way that you can start another one. Actually, this will use the same executioner, so you have to wait till that finishes to start. So you can pipeline to get a huge amount of things going through the pipeline. And so each cycle you can graduate many, many things going on.

AUDIENCE: Can I ask you a quick question? Something I'm trying to get straight in my head. My notion -- and I don't think I'm right on this, that's why I'm asking you -- is machines like the Cray, I know you were talking about some of the vector operations, those were by and large a relatively small set of operations. Like dot products, and vector time scalar. On the other hand, when you look at the SIMD machines, they had a much richer set of operations.

PROFESSOR: I think with scatter-gather and things like conditional execution, I think vector machines could be a fairly large -- I mean it's painful.

AUDIENCE: [INAUDIBLE]

PROFESSOR: The SIMD instruction is Pentium. I think that is mainly targeting single processing type stuff. They don't have real scatter-gather.

AUDIENCE: And the Cell processor?

PROFESSOR: Cell is distributed memory.

AUDIENCE: Yeah, but on one the -- what do they call them, the --

PROFESSOR: I don't think you can scatter-gather either. It's just basically, you have to align words in, word out. IBM is always about doing align. So in even AltiVec, you can't even do unaligned access. You had to do aligned access. So if there is no run align, you had to pay a big penalty in there.

So if you look at how this happens. So you have this entire pipeline thing. When things get started the first value is at this point done in one clock cycle. The next value is halfway through that. Another value is in some part of a -- is also pipelined, the alias pipeline. And other values are kind of feeding nicely into that. So if you have one -- this is called one lane. You can have multiple lanes, and then what you can do is each cycle you get 40 [UNINTELLIGIBLE] And the next ones are in the middle of that, next ones are in middle. So what you have is a very pipelined machine, so you can kind of pipeline things in there. So you can have either one lane, or multiple lanes pipeline coming out.

So if you look at the architecture, what you had is you have some kind of vector registers feeding into these kind of functional units. So at a given time, in this one you might be able to get eight results out, because everything gets pipelined. But the same thing is happening in there.

Clear how vector machines work? So it's not really parallelism, it's basically -- especially if you are one -- it's a superpipelined thing. But given one instruction, it will crank out many, many, many things for that instruction. And doing parallelism is easy in here too, because it's the same thing happening to very regular data sets. So there's no notion of asynchronizations and all these weird things. It's just a very simple pattern.

So the next thing is the shared sequencer processor. So here it's similar to the vector machines because each cycle you issue a single instruction. But the instruction is a wide instruction. It had multiple operations in these same instructions. So what it says is -- "I have multiple execution units, I have memory in a separate unit, and each instruction I will tell each unit what to do." And so something you might have -- two integer units, two memory/load store units, two floating-point units. Each cycle you tell each of them what to do. So you just kind of keep issuing an instruction that affects many of them. So sometimes what happens is if this has latency of four, you might have to wait till this is done to do the next instruction. So if one guy takes long, everybody has to kind of wait till that. So it's very synchronous going. So things like synchronization stuff were not an issue in here.

So if you look at a pipeline, this is what happens. So you have this instruction. It's an instruction, but you are fetching a wide instruction. You are not researching a simple instruction. You decode the entire thing, but you can decode it separately. And then you go execute on each execution unit.

One interesting problem here was this was not really scalable. What happened here is each functional unit, if you had one single register file, has to access the register file. So each function would say -- "I am using register R1," "I am using R3," "I am using R5." So what has to happen is the register file has to have -- basically, if you have eight functional units, 16 outports and 8 inports coming in. And then of course, when you build a register file it has a scale, so it had huge scalability issues. So it's a quadratically scalable register function. Question?

AUDIENCE: The sequencer [INAUDIBLE PHRASE]

PROFESSOR: Yeah. It's basically you had to wait till everybody's done, there's nothing going out of any order.

And memory also. Since everybody's going to memory, this is not scalable. So people try to build -- you can do four, eight wide, but beyond that this register and memory interconnect became a big mess to build.

And so one kind of modification thing people did was called Clustered VLIW. So what happens is you have a very wide instruction in here. It goes to not one cluster, but different clusters. Each cluster has its own register file, its own kind of memory interconnect going on there. And what that means is if you want to do intercluster communication, you have to go to a very special communication network. So you don't have this bandwidth expansion register. So you only have, we'll say two execution units, so you only have to have four out and one in to the register filing cycle. And then if you want other communication, you have a much lower bandwidth interconnect that you'll have to go through that. So what this does is you kind of expose more complexity to the compiler and software, and the rationale here is most programs have locality. It's like everybody always wants to to communicate with everybody else, so there are some locality in here. So you can basically cluster things that are local together and put it in here, and then when other things have to be communicated you can use this communication and go about doing that. So this is kind of the state of the art in this technology. And something like -- what I didn't put -- Itanium kind of fits in here. Itanium processor.

So then we go to shared network. There has been a lot of work in here. People have been building multiprocessors for a long time, because it's a very easy thing to build. So what you do is -- if you look at it, you have a processor unit that connects its own memory. And it's like a multiple [UNINTELLIGIBLE] Then it has a very tightly connected network interface that goes to interconnect network. So we can even think about a workstation farm as this type of a machine. But of course, the network is a pretty slow one that requres an ethernet connector. But people build things that have much faster networks in there. This was designed in a way you can build hundreds and thousands of these things -- nodes in here. So today if you look at the top 500 supercomputers, a bunch of them fits into this category because it's very easy to scale and build very large.

AUDIENCE: Are you doing SMPs in this list, or some other place?

PROFESSOR: SMP is mostly shared memory, so shared network. I'll do shared memory next.

But there are problems with it. All the data layout has to be handled by software, or by the programmer basically. If you want something outside your memory, you had to do very explicit communication. Not only you, the other guy who has the data actually has to cooperate to send it to you. And he needs to know that now you have the data. All of that management is your problem. And that makes programming these kind of things very difficult, which you'll probably figure out by the time you're done with Cell. So Cell has a lot of these issues, too.

The problem here is not dealing with most of the data, but the kind of corner cases that you don't know about that much. There's no nice safe way, of saying -- "I don't know where, who's going to access it. I'll let the hardware take care of it." There's no hardware, you have to take of it yourself.

And also message passing has a very high overhead. Most of the time in order to do message, you have to invoke some kind of a kernel thing. You have to actually do a kernel switch that will call the network -- it's operaing system involves a process, basically, to get a message in there. And also when you get a message out you have to do some kind of interrupt or polling, and that's a bunch of copies out of kernel. And this became a pretty expensive proposition. So you can't send messages the size of one [UNINTELLIGIBLE] so you had to accumulate a huge amount of things to send out to amortize the cost of doing that.

Sending can be somewhat cheap, but receiving is a lot more expensive. Because receiving you have to multiplex. You have no idea who it's coming to. So you have to receive, you have to figure out who is supposed to get it.

Especially if you are running multiple applications, it might be for someone's application. You had to contact [UNINTELLIGIBLE] So it's a big mess.

That is where people went to shared memory processors, because it became easier message method to use. So that is basically the SMPs Alan was talking about. The nice thing is it will work with any data placement. It might work very slowly, but at least it will work. So it makes it very easy to take your existing application and first getting it working, because it's just working there. You can choose to optimize only critical sections. You can say -- "OK, this section I know it's very important. I will do the right thing, I will place it properly everything." And the rest of it I can just leave alone, and it will go and get the data and do it right. You can run sequentially, of course, but at least the memory part I don't have to deal with it. If some other memory just once in a while accesses that data that you have actually parallelized, it will actually work. So you only have to worry about the [UNINTELLIGIBLE] that you are parallelizing.

And you can communicate using load store instructions. You don't have to get always in order to do that. And it's a lot lower overhead. So 5 to 10 cycles, instead of hundreds to thousands cycles to do that. And most of these messages actually stoplight some instructions to do this communication very fast. There's a thing called fetch&op, and a thing called load linked/store conditional operations. There are these very special operations where if you are waiting for somebody else, you can do it very fast. So if two people are communicating. So people came up with these very fast operations that are low cost, as a last -- if the data's available it will happen very fast. Synchronization.

And when you are starting to build a large system, you can actually give a logically shared view of memory, but the underlying hardware can be still distributed memory. So there's a thing called -- I will get into when you do synchronization -- directory-based cache coherence. So you give a nice, simple view of memory. But of course memory is really disbributed. So that kind of gives the best of both worlds. So you can keep scaling and build large machines, but the view is a very simple view of machines.

So there are two categories in here. One is non-cache coherent, and then hardware cache coherence. So non-cache coherence kind of gives a view of memory as a single address space. But you had to deal with that if you write something to get there early to me, you had to explicitly say -- "Now send it to that person." But we're still in a single address space. It doesn't give the full benefits of a shared memory machine. It's kind of inbetween distributed memory. In distributed memory basically everybody's in a different address space, so you had to map by sending a message. Here, you just say I have to flush and send it to the other guy.

Some of the early machines, as well as some big machines, were no hardware cache coherence. Things like supercomputers were built in this way because it's very easy to build. And the nice thing here is if you know your applications well, if you are running good parallel large applications, and you are actually knowing what the communication patterns are -- you can actually do it. And you don't have to pay the hardware overhead to have this nice hardware support in there. However, a lot of small scale machines -- for example, most people's workstations are stuffy, it's probably now two Pentium Quad machines -- actually add memory. Because if you are trying to do the starting things it's much easier to do shared memory. And also it's easier to bulid small shared memory machines. And people talk about using a bus-based machine, and also using a large scale directory-based machine in here.

So for bus-based machines, how do you do shared memory? So there's a protocol, what we call a snoopy cache protocol. What that means is, every time you modify your location somewhere -- so of course you have that in your cache -- you tell everybody in the world who's using a busing, "I modified that." And then if somebody else also has that memory location. That person says, "Oops, he modified it." Either he invalidates it or gets the modified copy.

If you are using something new, you have to go and snoop. And you can ask everybody and say -- "Wait a minute, does anybody have a copy of this?" And some more complicated protocols have saying -- "I don't have any, I have a copy but it's only read-only. So I'm just reading it, I'm not modifying it." Then multiple people can have the same copy, because everybody's reading and it's OK. And then there's the next thing -- "OK, I am actually trying to modify this thing." And then only I can have the copy. So some data you can give to multiple people as a read copy, and then when you are trying to write everybody gets disinvited, only the person who has write has access to it. And there are a lot of complicated protocols how if you write it, and then somebody else wants to write it, how do you get to that person? And of course you have to keep it consistent with memory. So there is a lot of work in how to get these things all working, but that's the kind of basic idea.

So directory-based machines are very different. In directory-based machines mainly there's a notion of a home node. So everybody has local space in memory, you keep some part of your memory. And of course you have a cache also. So you have a notion that this memory belongs to you. And every time I want to do something with that memory I had to ask you. I had to get your permission. "I want that memory, can you give it to me?"

And so there are two things. That person has a directory [UNINTELLIGIBLE] say -- "OK, this memory is in me. I am the one who right now owns it, and I have the copy." Or it will say -- "You want to copy that memory to this other guy to write, and here is that person's address or that machine's name." Or if multiple people have taken this copy and are reading it. So when somebody asks me for a copy -- assume you ask to read this copy. And if I have given it to nobody to read, or if I have given it to other people to read, so I say -- "OK, here's a copy. Go read." And I add that person is reading that, and I keep that in my directory.

Or if somebody's writing that. I say sure, "I can't give it to read because somebody's writing that." So I can do two things. I can tell that person, saying -- "You have to get it from the person who's writing. So go directly get it from there. And I will mark that now you own it as a read value." Or, I can tell the person who's writing -- "Look, you have to give up your write privilege. If you have modified it, give me the data back." And that person goes back to the read or no privileges with that data. When I get that data, I'll send it back to this person and say -- "Here, you can read." And the same thing if you ask for write permission. If anybody has [UNINTELLIGIBLE] I have to tell everybody -- "Now you can't read it anymore. Go invalidate, because somebody's about to write." Get the invalidate request coming back, and then when you've done that I say, "OK, you can write that." So everybody keeps part of the memory, and then all of that in there. So because of that you can really scale this thing.

So if you look at a bus-based machine. This is the kind of way it looks like. You have a cache in here, microprocessor, central memory, and you have a bus in here. And a lot of small machines, including most people's desktops, basically fit in this category. And you have a snoopy bus in here.

So a little bit of a bigger machine, something like a Sun Starfire. Basically it had four processors in the board, four caches, and had an interconnect that actually has multiple buses going. So it can actually get a little bit of scalability, because here's the bottleneck. The bus becomes the bottleneck. Everybody has to go through the bus. And so you actually get multiple buses to get bottleneck, and it actually had some distributed memory going through a crossbar here. So this cache coherent protocol has to deal with that.

And going to the other extreme, something like SGI Origin. In this machine there are two processors, and it had actually a little bit of processors and a lot of memory dealing with the directory. So you keep the data, and you actually keep all the directory information in there -- in this. And then it goes -- then after that it almost uses a normal message passing type network to communicate with that. And they use the crane to connect networks, so we can have a very large machine built out of that.

So now let's switch to multicore processors. If you look at the way we have been dealing with VLSI, every generation we are getting more and more transistors. So at the beginning when you have enough transistors to deal with, people actually start dealing with bit-level parallelism. So you didn't have -- you can do 16-bit, 32-bit machines. You can do wider machines, because you have enough transistors. Because at the beginning you have like 8-bit processors, 16-bit, 32-bit.

And then at some point that I have still more transistors, I start doing instruction-level parallelism in a die. So even in a bit-level parallelism, in order to get 64-bit you had to actually have multiple chips. So in this regime in order to get parallelism, you need to have multiple processors -- multiprocessors. So in the good old days you actually built a processsor, things like a minicomputer. Basically you had one processor dealing with a 1-bit slice. So in the 4-bit slice, dealing with that amount, you could fit in a chip. And a multichip made a single processor. Here a multichip made a multiprocessor. We are hitting a regime where a multichip -- what [? it ?] will be multiprocessor -- now fits in one piece of silicon, because you have more transistors. So we are going into a time where multicore is basically multiple processors on a die -- on a chip.

So I showed this slide. We are getting there, and it's getting pretty fast. You had something like this, and suddenly we accelerated. We added more and more cores on a die.

So I categorized multicores also the way I categorized them previously. There are shared memory multicores. Here are some examples. Then there are shared network multicores. Cell processor is one, and at MIT we are building also Raw processor. And there is another part, what they call crippled or mini-cores. So the reason in this graph you can have 512, is because it's not Pentium sized things sitting in there. You are putting very simple small cores, and a huge amount of them. So for some class replication, that's also useful.

So if you look at shared memory multicores, basically this is an evolution path for current processors. So if you look at it, what they did was they took their years' worth of and billions of dollars worth of engineering building a single superscalar processor. Then they slapped a few of them on the same die, and said -- "Hey, we've got a multicore." And of course they were always doing shared memory at the network level. They said -- "OK, I'll put the shared memory bus also into the same die, and I got a multicore." So this is basically what all these things are all about. So this is kind of gluing these things together, it's a first generation. However, you didn't build a core completely from scratch. You just kind of integrated what we had in multiple chips into one chip, and basically got that.

So to go a little bit beyond, I think you can do better. So for example, this AMD multicore. Basically you have CPUs in there, actually have a full snoopy controller in there, and can have some other interface with that. So you can actually start building more and more uni CPU, thinking that you're building a multicore. Instead of saying, "I had this thing in my shelf, I'm going to plop it here, and then kind of [INAUDIBLE]

And you'll see, I think, a lot of interesting things happening. Because now as they're connected closely in the same die, you can do more things than what you could do in a multiprocessor. So in the last lecture we talked a little bit about what the future could be in this kind of regime.

Come on.

OK. So one thing we have been doing at MIT for -- now this practice is ended, we started about eight years ago -- is to figure out when you have all the silicon, how can you build a multicore if you to start from scratch. So we built this Raw processor where each -- we have 16, these small cores, identical ones in here. And the interesting thing is what we said was, we have all this bandwidth. It's not just going from pins to memory, we have all this bandwidth sitting next to each other. So can we really take advantage of that to do a lot of communication? And also the other thing is that to build something like a bus, you need a lot of long wires. And it's really hard to build long wires. So in Raw processor it's something like each chip, a large amount of part, is into this eight 32-bit buses. So you have a huge amount of communication next to each other. And we don't have any kind of global memory because that requires, right now, either do a directory, which you didn't want to build, or have a bus, which will require long wires. So we did in a way that all wires -- no wires longer than one of the cores. So we can do short wires, but we came up with a lot of communications for each of these, what we called tile those days, are very tightly coupled.

So this is kind of a direction where people perhaps might go, because now we have all this bandwidth in here. And how would you take advantage of that bandwidth? So this is a different way of looking at that. And in some sense the Cell fits somewhere in this regime. Because what Cell did was instead of -- it says, "I'm not building a bus, I am actually building a ring network. I'm keeping distributed memory, and provide to Cell a ring." I'm not going to go through Cell, because actually you had a full lecture the day before yesterday on this. AUDIENCE: Saman, can I ask you a question? Is there a conclusion that I should be reaching in that I look at the multicores you can buy today are still by and large two and four processors. There are people that have done more. The Verano has 16 and the Dell has 8. And the conclusion that I want to reach is that as an engineering tradeoff, if you throw away the shared memory you can add processors. Is that a straightforward tradeoff?

PROFESSOR: I don't think it's a shared memory. You can still have things like directory-based cache coherent things. What's missing right now is what people have done is just basically took parts in their shelves, and kind of put it into the chip. If you look at it, if you put two chips next to each other on a board, there's a certain amount of communication bandwidth going here. And if you put those things into the same die, there's about five orders of magnitude possibility to communicate. We haven't figured out how to take advantage of that. In some sense, we can almost say I want to copy the entire cache from this machine to another machine in the cycle. I don't think you even would want to do that, but you can have that level of huge amount of communication. We are still kind of doing this evolutionary path in there [UNINTELLIGIBLE] but I don't think we know what cool things we can do with that. There's a lot of opportunity in that in some sense.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, because the interesting thing is -- the way I would say it is, in the good old days parallelization sometimes was a scary prospect. Because the minute you distribute data, if you don't do it right it's a lot slower than sequential execution. Because your access time becomes so large, and you're basically dead in water. In this kind of machine you don't have to. There's so much bandwidth in here. Latency was still -- latency would be better than going to the outside memory. And we don't know how to take advantage of that bandwidth yet. And my feeling is as we go about trying to rebuild from scratch multicore processors, we'll try to figure out different ways. So for example, people are coming up with much more rich semantics for speculation and stuff like that, and we can take advantage of that.

So I think there's a lot of interesting hardware, microprocessor, and then kind of programming research now. Because I don't think anybody had anything in there saying , "Here's how we would take it down to this bandwidth." I think that'll happen.

Now the next [? thing ?] is these mini-cores. So for example, this PicoChip has array of 322 processing elements. They have 16-bit RISC, so it's not even a 32-bit. Piddling little things, 3-way issue in. And they had like 240 standard -- basically, nothing more than just a multiplier, and add in there. 64 memory tiles, full control, and 14 some special [UNINTELLIGIBLE] function accelerator.

So this is kind of what people call heterogeneous systems. Where what this is -- you have all these cores, why do you make everything the same? I can make something that's good doing graphics, something that's good doing networking. So I can kind of customize in these things. Because what we have in excess is silicon. We don't have power in excess. So in the future you can't assume everything is working all the time, because that will still create too much heat. So you kind of say -- the best efficiencies, for each type of computation you have some few special purpose units. So we kind of say if I'm doing graphics, I fit to my graphics optimized code. So I will do that. And the minute I want to do a little bit of arithmetic I'll switch to that. And sometimes I am doing TCP, I'll switch to my TCP offload. Stuff like that. Can you do some kind of mixed in there? The problem there is you need to understand what the mix is. So we need to have a good understanding of what that mix is. The advantage is it will be a lot more memory efficient. So this is kind of going in that direction.

And so in some sense, if you want to communicate you have these special communication elements. You have to go through that. And the processor can do some work, and there are some memory elements. So far and so forth.

So that's one push, people are pushing more for embedded very low power in.

AUDIENCE: Is this starting to look more and more like FPGA, which is [UNINTELLIGIBLE]

PROFESSOR: Yeah, it's a kind of a combination. Because the thing about FPGA is, it's just done 1-bit lot. That doesn't make sense to do any arithmetic. So this is saying -- "Ok, instead of 1 bit I am doing 16 bits." Because then I can very efficiently build [UNINTELLIGIBLE] Because I don't have to build [UNINTELLIGIBLE] out of scratch.

So I think that an interesting convergence is happening. Because what happened, I think, for a long time was things like architecture and programming languages, and stuff like that, kind of got stuck in a rut. Because things there are so very efficiently and incremental -- it's like doing research in airplanes. Things are so efficient, so complex. Here AeroAstro can't build an airplane, because it's a $9 billion job to build a good airplane in there. And it became like that. Universities could not build it because if you want to build a superscalar it's, again, a $9 billion type endeavor to do that -- thousands of people, was very, very customized. But now it's kind of hitting the end of the road. Everbody's going back and saying -- "Jeez, what's the new thing?" And I think there's a lot of opportunity to kind of figure out is there some radically different thing you can do.

So this is what I have for my first lecture. Some conclusions basically. I think for a lot of people who are programmers, there was a time that you never cared about what's under the hood. You knew it was going to go fast, and in the air it will go faster. I think that's kind of coming to an end. And there's a lot of variations/choices in hardware, and I think software people should understand and know what they can choose in here. And many have performance implications. And if you know these things you will be able to get high performance of software built easy. You can't do high performance software without knowing what it's running on.

However, there's a note of caution. If you become too much attached to your hardware, we go back to the old days of assembly language programming. So you say -- "I got every performance out of a -- now the Cell says you have seven SPEs." So in two years, they come with 16 SPEs. And what's going to happen? Your thing is still working on seven SPEs very well, but it might not work on 16 SPEs, even with that. But of course, you really customize for Cell too. And I guarantee it will not run good with the Intel -- probably Quad, Xeon processor -- because it will be doing something very different. And so there's this tension that's coming back again. How to do something that is general, portable, malleable, and at the same time get good performance with hardware being exposed. I don't think there's an answer for that. And in this class we are going to go to one extreme. We are going to go low level and really understand the hardware, and take advantage of that. But at some point we have to probably come out of that and figure out how to be, again, high level. And I think that these are open questions.

AUDIENCE: Do you have any thoughts, and this may be unanswerable, but how could Cell really [INAUDIBLE]. And not Cell only, but some of these other ones that are out there today, given how hard they are to program.

PROFESSOR: So I have this talk that I'm giving at all the places. I said the third software crisis is due to multicore menace. I termed it a menace, because it will create this thing that people will have to change. Something has to change, something has to give. I don't know who's going to give. Either people will say -- "This is too complicated, I am happy with the current performance. I will live for the next 20 years at today's level of performance." I doubt that will happen. The other end is saying -- "Jeez, you know I am going to learn parallel programming, and I will deal with locks and semaphores, and all those things. And I am going to jump in there." That's not going to happen either. So there has to be something in the middle. And the neat thing is, I don't think anybody knows what it is. Being in industry, it makes them terrified, because they have no idea what's happening. But in a university, it's a fun time.

[LAUGHTER]

AUDIENCE: Good question.

PROFESSOR: OK. So we'll take about a five minutes break, and switch gears into concurrent programming.