PROFESSOR: OK. So today is the final lecture, and I'm going to talk to about the future. If I can figure out how to get the slides going. Predicting the Future is always not a simple thing.

And especially this day and age we are -- things are appearing on the internet, so this is getting recorded and on the internet. And I'm going to make a few comments that in five years time, I'm going to regret.

What I'll try to do is try to have a systematic process through that. Because I know you guys are great hackers. You guys are great programers. You and any problem, you can solve it. But as researchers, what's hard, and what's very difficult, is to figure out which problem to solve.

Because as Roderick pointed out, when we started on Raw in about '97, there was a raging debate what to build. And there's group of people who say we can build bigger and bigger superscalars. In fact, there was this article, very seminal proceedings that came out, that basically asked many people to contribute, saying what do they think will a billion transistor processor look like?

And about 2/3 of them said, we would have this gigantic, fabulous superscalar. And about three out of about ten, I think about three groups said that, OK, it would be more like tiled architecture. The Stanford people and us are the main people who look at that.

So even that short time, 10 years ago, it was not sure what's going to happen. And so we took a stance and spent 10 years working on that. We were lucky, more than anything else, that ITV had a little bit better insight, hopefully, but also mostly a larger dose of luck we get here.

So part of that is how do you go about doing this? So future ia a combination of evolution and revolutions. Evolution is somewhat easy to predict. Basically, what you can look at is, you can look at the trends going on and you can extrapolate that trend. And you can say, if this trend continue, what will happen? If this trend continue what will happen?

It's very interesting in computer science because one thing I do in 6033 is look at the rate of

evolution that happens in computer science, and trying to predict it with things like the rate of [OBSCURED] and stuff like that and you get ridiculous numbers.

So even if it is evolution, in computer science it's very close to revolution. Because every 18 months things double. That's almost unheard of. In a lot of areas, every 18 months you get 1% improvement and people are very happy. This is doubling. We have that.

THe other part is revolutions. If some completely new technological solution just come about and completely change the world. Change how things happen. These are a lot harder to predict. But still it's critically important because some of these things can have huge impact. And so we will more focus on evolution and if we have at the end we can see if we have some interesting revolutionary ideas.

Paradigm shifts occur in both. You don't have to wait for a revolution to have something different. Because things like 2x improvement type things that happens, actually brings paradigm shifts on a very regular basis. So it's not 1% we are a getting. We are getting huge evolution changes. And that can keep changing things.

So what I'm going to talk to you about, a little bit of trends. Some of it is repetitive. I spent sometime in the first lecture talking about these trends. And then we'll look at two different parts. One is architecture side, what can happen next. And languages, compilers and tools side.

We'll spend a little bit of time on revolution, and just see that we have group brainstorming. And then I have this little bit of a my preview of what people should be doing, very broad level, and I'll finish off with that.

So look at trend. I think there are a lot of interesting trends that we can look at. Moore's law, which is a very trusting trend that keep going for a very long time. Power consumption, wire delay, hardware complexity, parallelizing compilers, program design methodologies. So you have all these trends her to look at.

I have this picture, the graph on the right, which shows what happened at every generation, assume this is a generation. You concentrate on the hardest problem, what's the most taxing thing? And what happens is as you go, at this time, something like this might be the taxing, and then that generation, that's where all the design going on.

Then some slowly growing things start catching up, and then, after some time, that becomes the most important thing in the design. So it's a paradigm shift, because you move that focus on that, that becomes the important thing. And then some fast moving things start catching up and accelerate, and that becomes our most taxing thing.

And then this thing, again, and the different curves start catching up again. So what happens is you have this cyclic phenomenon, too. But also you have the discrete So you might get into a revolutionary thing, and really bring something down. For example, we went from very power-hungry, bipolar devices to SEMOS. And suddenly everything shifted down a lot. And then we are back again, we are those power-hungry days again. So that was kind of a revolutionary change and get evolution.

If you want to do work on something, it's no fun working on something that's very important, hot today. The people who are making the biggest breakthroughs on the things that are hot today, started 10 years ago. So your finally figure out that this is the most important technology today, I'm going to start working on doing my Ph.D. By the time you are done, it'll be passe, most of these cyclic things.

So the key thing is, what is going to be hot in 10 years? That's a hard problem. So this is the thing you can look at these trends, and say, look, these are the kind of trends. This can give you some idea where what might be interesting.

So are you going to become a low power person? For example, six, seven years ago, people who studied low power, they made a lot of significant contributions. People now still can contribute. But people who got in early had an easier time, and they had a lot more impact then people who got in late.

So we look at some of these trends, and see what we can come up with. The first trend, this is a very trust trend so far, is Moore's Law. Basically, most of our basis for things happening is part of Moore's Law. Basically it says, you get twice the number of transistors every 18 months. We have had that trend for a very long time.

What's the impact of function? Nobody cares about processors who can't use transistors, it's some kind of closed, you don't even know many transistors are in the processor. What matters is what it can deliver. So performance is what? Delivery. So what people have realized is if you are in the superscalar world, this has been flattening out recently.

If you look at it, this is kind of getting flat and that was a big problem. So that's kind of the reason for crossover for superscalar, I mean multicore. So we are trying to get back into again a higher growth curve in here.

And then another trend is power. Power had been going in the wrong direction and also power density has been going in the wrong direction. This current keep growing more than this and then people have problems in that.

Something that is really revealing is what's per spec performance. That means how much power you are consuming to do something. And what you see is, we are just starting to -- these days we use very little power to get work done, and now we, what superscalars did was be more and more [OBSCURED]. And this is what, I think, led to a big reason for going into multicores. Because we realized, look, we keep wasting and now the power consumption of computers are non-trivial. You could talk to Google. Their power costs, power budgets, is non-trivial. We can't just keep wasting that resource.

You can keep wasting things when it is not that important, smaller factor, but power becomes a big factor. So how do we get out of the wasteful mode? And I think that -- a lot of people are I'm looking at that. If you talk to people five years ago, people say, this is free. Transistors are free. You can just keep doing things. Just keep them available, [OBSCURED] makes it free. But what people realize is, wire transistors are free, switching them is not free. And switching the request file, so power become an important issue.

At that point I want to point out was this wire delay. As we keep switching them faster and faster, the world looks smaller and smaller and smaller, you can make global decisions. What the implication of that? And we're facing that, I think it's going to get a little bit worse. One saving grace for that is people realize they can keep scaling the clock speeds that much. This chart I did a few years ago, still is a road map, that's made the silicon people think where the silicon is going.

Thought they can get to 10 gigs, then 15 gigs, and stuff like that. And now they [OBSCURED], and go to all of these power requirements. They might not be able to switch that fast. So there are changes to road map because of that. Another interesting thing is --

AUDIENCE: RC line type wire delay, as opposed to -- that's not speed of light delay, that's including scaling of the wires themselves?

PROFESSOR: This is mainly speed of light delay. Because what you do is you just look at clock speeds. I haven't done too much of dialectrical and stuff like that. That's the thing, you can get closer to the speed of light but right now we are only about -- I mean we are not even a factor of, I don't know, 80%, 90% speed of light. So best case we can get 10% better. It doesn't give you that much of room.

And this is a continuous thing, because microprocessors are going much faster than d-RAM. And this is having a problem. There are people says there might be revolutionary things happening here. Things like non-volatile RAM, might come and kind of take over this area. What happening today is d-RAM now looks a lot more like a disc looked like 10 years ago. In may ways.

We have to treat a d-RAM like a disc. That's kind of the way we have of looking at the world.

Here where I look at compilers and parallelizing. There's no nice number here. So in somewhere around the 1970s, people did things like vectorization -- question?

AUDIENCE: I have a question about the last slide. When people do this plot which is performance, it's not exactly performance that or throughput that matters relative to d-RAM, it is the computation lengths that is how fast from when I know I want something from memory to when I really, really need it that matters. Is this going to change as we move to a wider architecture, with more parallelism, in the sense that it will no longer be, the lazy gap won't be increasing anymore and we'll start to hear more about bandwidth?

PROFESSOR: I don't know, because bandwidth is also power. These are a lot of interesting arguments, this is what you don't know. Because the thing is, what superscalar people did was, we reduced the problem with latency by predicting. And doing extra work. We will try as much as possible to predict what you need and do things in advance. What happens is as you keep predicting more and more, you're actually doing, your accuracy of predictions go low.

But you still keep increasing -- the more data you at least get a small percentage increase in useful data. Effecting latency. But accuracy start pretty low and it start becoming not useful. So bandwidth problem has an impact, too.

So unless you're using inside the entire thing, you're predicting a huge amount of things, predicting that you might use it. So if the prediction is wrong, it's a lot of wasteful work. And the key thing is can you pay by power?

I think latency, at some point, if you can really predict, it becomes hard. Unless you have very regular patterns that the predictions working there.

If you look at what happened in compiling and programming technology. In the '70s, you gets vectorization technology. It did well. And then some time in 80's we did things that compile for instruction level parallelism. That got a lot of parallelism in there, very low, low parallelism in there. And somewhere in 90's we did automatic parellelization compilers for FORTRAN.

I think that was today almost the highlight of that. Because what happened was we figured out how to get the program language in FORTRAN and do pretty well automatically extract patterns.

Then things were kind of down in front there because people say FORTRAN is interesting, but we don't program in FORTRAN anymore. We are going to program in things like C, C . That did two things.

First of all, one is a language problem, which is C, C was unfortunately was not a typesafe language. It was not be given up the right scientific application. More like operating system hacking, that required looking at the entire address space. That really hindered the compiler.

Second, programmers start falling in love with complex data structures. The world is not fixed size arrays any more. They want to develop things, have different structures, trees. These things are much harder to analyze. So suddenly you can do it nicer, they said, look, people are doing recursions and these very complex structures and suddenly, the more of the things you can go went down pretty drastically. And you're back to a pretty bad state.

And then we're slowly recovering a little bit because things like JAVA and C Sharp gave a typesafe language, became a little more analyzable. I think automatically what you can do kind of improves. And the hope here is a demand driven by multicores There's no supply technology but there's the demand. You need to do something. And hopefully there'll be interesting things that come about because of the demand. So this is my prediction, where things are. What happened then and what's going to happen next.

Of course, multicores are here. There's no question about that. But I don't think programmers are ready for that. I mean we have established a very nice, solid boundary between hardware and software. Nice abstraction layer. Most programmers don't have to worry about hardware. Programmers didn't have any knowledgeable about the process. In fact, we are moving in that

direction.

JAVA said you don't even have to know about the class of the program. We are running this thing in a nice high-level byte code. Write once, run everywhere. That's a very nice abstraction.

For most types, more slow actually gave you enough performance, that that was good enough. We can deal with that. The nice artifact of that is programmers were oblivious to what happened in processors. A program written in 1970 still works. Still will run 10 times faster. That is really great. If you look at Windows Office that probably [OBSCURED]. We still have code written in 1970, but in 1980. There's still code in there. Every time they tout they have this completely new program. Only probably 10% of new code is there.

But that's not true for the cell phone. Every time they say I have a new cell phone, most of the time actually rewritten completely. So that's why in an industry like that it's very hard to evolve like that because we are just churning too much.

And in soft ware we got really enamored by this ability to reuse stuff. Ability to have this nice abstraction in there. And the problem is there's probably a lot of freedom for programmers, so they push on complex issues. So your question about how large companies want complex things, they way people get their computer advantage is pushing the complexity. So what they did was, instead of pushing the complexity in the processor, they push the complexity in software, features and stuff like that. So we are still on the brink of complexity. But now we are adding another complexity that's very hard to deal with.

So this is where things came about, how things move. Where can we go? I want to talk about two different parts. Architecture, and languages, compilers and tools. Most of the things I talk about, That's no solution. This is what I think that will happen, should happen. I think that can hopefully really inspire people to go do that. And hopefully I don't have to eat what I say in 10 years.

One thing now about novel opportunities in multicores is you don't have to contend with uniprocessors. Most of the time they are a source of big pain when you are doing research in here. Everybody says, yeah, that is very nice, but I just wait two years and this will happen for me automatically. Why do I have to worry about your complex whatever you are suggesting in there.

The other thing is, in the good old days, the people who really wanted performance were these really performance weenies. In fact, you probably got a good talk from a performance weenie last time, who's [OBSCURED]. And we want just performance at all cost. There are people like that.

For them, producing anything it's not easy. They say, I don't want tools. They just suck my performance out. And it was very hard to help them, because they don't want help. But right now what's happening this is becoming everybody's problem. I think that will lead people to want the things to take advantage of parallelism.

The other thing is, the problem can change. Because people worked for 20 years or 40 years on this parallel problem, but that's for multiprocessors. So how does this problem has changed? What's the impact from going from a multiprocessor, which is multiple chips, sitting on a board, with some kind of interconnecting to multiple cores on the same die. OK. I think there are some very substantial things that happen, that can have big impact.

Let's look at two of them. Which is communication bandwidth and communication latency. I think this is where you can have revolutionary kind of things happening. If you look at communication bandwidth, the best you could get, if you put two chips on any kind of a board, it's about 32 gigabits per second. Because you had to go through pins and pins had a lot of issues.

Today if you try to put two different cores next to each other in the same die, the bisection bandwidth is huge. It's four orders of magnitude huge, that's what you can get. That's a huge change. Even for the lowest 2x improvement, this is four orders of magnitude difference you can do.

So why is that? Because what changes number of wires in the die. Because, if you think about it, one name type of architectures is to just just build, to deal with this reduction of communication bandwidth. The closed resistors communicates the most, the local memory next, and then remote memory. And then if you had to go remote to the next guy, network. These things basically reduce the amount of things you can do. That was really true.

Because if you you're inside the chip, you can do a lot more. The minute you go to the board, you can do less. Minute you go to multiple boxes, less. Stuff like that. That is the way the processor would decide. And also the clocks. Being a processor you can do much faster to go pins, not that fast. People are trying to change that, but still that's an issue. And multiplexing,

because pins were inherently limited. If you have thousand pins that's a big processor basically in there.

And within a die, a thousand wires is basically nothing. So we have four orders of magnitude improvement in here. And that's amazing. That can have a huge impact in here. So you can do massive data exchange. On big issue right now in parallelism is the minute you parallelize something, you might slow it down. Because the thing is, you might not get data locality. Because that pipe is going to get clogged up because you've been trying to take too much data in there.

So if you put wrong things, you're going to get clogged by the communication panel. This thing says, I might not have to deal with that. For example, I don't know why you want to do that, but if you want to move your entire cache from one core to another, you can probably do it in a few clock cycles. That is bandwidth.

I don't know why you want to do that, because that's probably not memory efficient, power efficient. But you can do it. We have wireless. So the key thing is how can you take advantage of these wires. And how can you make it in a way you can really use the difficulty in programming, because, as you've figured out, parallel programming is hard. If you make a wrong decision, you're going to pay a huge price.

How can you make that in a way that you can make a few wrong decisions and get everything by taking advantage of that. I think that will be a big opportunity in here.

Another opportunity is latency. This is not as big as bandwidth because latency has the inherent speed of light issue. Again, what happens is now length of wires are a lot shorter. And no multiplexing, so you can get a direct wire from there. And on-chip can be much closer.

And the other interesting things you can also think about. Right now, why is register to register communication just very fast? In a processor if you send something one way this can use it almost instantaneously is much faster. Of course, wires are not that, it's very close. Also there are a lot of other things we do. We need more register, register before we even come to the data.

In a normal pipeline machine you have a pretty long pipeline. The data is externally receivable at the end of the pipeline, but I can speculate the same to the next one when you start working on it if you're wrong, I would yank it out and you have give [OBSCURED]. But right now if you

actually go across processors, we can use speculation. You have to wait until the things are committed to memory, and then you can start using it.

That's even longer than ideally, because that's like 10, 15 cycles I have to wait. Why can't I send something speculative to my neighbor? I have enough bandwidth to deal with it. If I do something wrong, can I speculate across multiple cores. So I can get my data much faster. I save probably an order of magnitude beyond my wire delay because I have no pipeline stages made until [OBSCURED] happens, very long later.

Things like that, can I do in modern process? Because no, I'm putting these darn things next to each other, what can I do with that? And you can do ultrafast [OBSCURED] real time and they can use -- can I do things a lot fine-grain across and take advantage of that?

So the way to look at that a traditional microprocessor basically had this structure. Cache memory and processor cache memory. And this is there because to deal with a small amount of wireless power and stuff like that. Today what we have done is basically taken this structure, and kind of put it in the same die. OK. We haven't thought beyond that.

And we get a few order, a few factors better. Because, no, I don't have to go through the pin because now I can have instead of say 128 pins, I can have 512 going in, or 1,024 going in there. But still, I have 4 orders of magnitude freedom in here. I am not using any of those. So one thing Raw tried to do is have a much more tightly coupled, a lot more communication going on there. So a lot of people say that today. The biggest problem in multicores is the network. Because how will I build that scale of a network? My feeling is that's baloney. Network is not an issue. In fact, in Raw, a lot of our experiments we could never saturate a very, very well done network.

Because in a network I have so many pins, I can keep adding bandwidth like crazy. The problem the way people design processor code today is to minimize communication going out of it. It has a very thin memory bus going in and out of it. That memory bus can never saturate any kind of network you can build on a microprocessor.

It matches well if you have to go through the pins, because it partially measures that. But within the die you have so much more bandwidth.

So I think something interesting would be a fundamental sign of what the microprocessor looks like. The core looks like. It's not taking a Pentium and kind of plopping it once, or plumping it

four times. A Pentium is not any kind of a nice processor, it evolved over the years to basically deal with this very thin memory bandwidth.

And suddenly you've got four [OBSCURED] and that problem goes away. And we're still saying, OK yeah, I have this entire thing in there, but I only want four wires out of you, or 1,024 wires when you can give me a hundred million wires. That is the question. So I think they'll be some interesting things that the people can make a big impact trying to figure out how to take advantage of that.

And by doing that, really reducing the burden for the programmer. Right now, you say, ok look, I am taking advantage of that, but I'm passing on the buck to the programmers. They're dealing with all the small interconnectary. Then we say, you can't just that because programming is hard.

How can I reduce the programmer burden by taking advantage of that. I think this is a very interesting to looked at this problem. If you're doing any architecture kind of research or are interested in that, here is an open problem. I have this huge bandwidth out there, and I have evolved something that is completely in the way dealing with very limited amount of bandwidth, but that's why they live for this long. And the minute you open it up, how can I do with that? I don't think we have a model. And I think a lot of interesting can come out there.

OK. Let's switch to language, compilers and tools. The way you look at it, what's the last big thing that happened in languages? I think one that happens is object oriented revolution. The interesting thing about object oriented revolution is it didn't happen in a vacuum. It happened because a lot of interesting research in academia and outside went into it. People didn't come up and say, wow, it's a great I'm writing Java. There were hundreds of languages that developed and explored these concepts.

And the interesting thing about programming is, a lot of things that looks very neat and interesting, when you look at it first time when you start using it in a media-line program, that has been evolve over 10 years, you realize that's a pretty bad concept. Can anybody think about a concept that looks very interesting at the beginning, but actually very hard to deal with.

AUDIENCE: My last program.

PROFESSOR: I haven't heard that analogy before but what else?

AUDIENCE: Exceptions.

PROFESSOR: Exceptions, I think people still haven't figured out. I think one thing people have made assumptions think of multiple integrators. With multiple integrators, the first step, wow this is the best thing since sliced bread, because the thing object oriented programming does is it forces this one hierarchy, and the world is objects, but there's no one hierarchy in the world. World has these multiple interconnections in there and so people say, this is great. This basically now give you objects and you can get a lot of different relationships.

But, after people trying to deal with multiple integrators they realize this is a pain. It's very hard to understand it, it's very hard to right compilers, there's so many ambiguities around it. And then at some point you just say, this is too much.

And that's why you went to, like Java went multiple interfaces light by having, you can have multiple interfaces. Here's a concept that looks very nice, but you had to work -- it took five, 10 years to realize that it's actually not usable.

So if you to look at object oriented languages, there have been many languages. This is an interesting thing, because in parallel community we haven't had this kind of explosion out there. Another interesting thing about languages is language has lot of relationship and evolving. So if you loot at from FORTRAN that's started. You can see the FORTRAN, so this is FORTRAN in here.

To figure out what's happening here, you can kind of trace over the years how new languages academically and industrially came about. Got influence from others, and at then end up with things like at the end of the graph there's something you can never read it. But the key thing is, there's all these influences going across in there. And cross fertilization.

At the end, you end up with some TCL TK, Python -- I can't even read it carefully. Java, C sharp, PHP, and stuff like that. There are a lot of different languages trying to get influence in there. The key thing is you need to feed this beast. If you want to have a lot of evolution or even evolutionary changes.

What that means is, people say, yeah, we have Java, C and C , C sharp, we are happy with languages. No. I think in the future to deal with that you need to have kind of revolution in there.

So if you look at something like C , almost every language started with FORTRAN. And then

there had been lot of different influences that happened to that. And I'm going to just go, that's for example, Java, it's acknowledge that a lot of these different languages influenced ideas in there.

So why do we need new languages in there? I think we have paradigm shifting architecture. Because sequential to multicore will require a new way of thinking, and we kind of need a new common machine language for these kind of machines. And the new application domains. Because streaming is becoming very interesting. Scripting. People are using things like Python and PEARL and stuff like that in a much regular basis to do much bigger things.

So those are cobbled together languages. But can you do better in those things?

Real time. Things like cell phones and stuff become more important how we deal with those issues. And also new hardware features. Right now good language is tied very well with the one [OBSCURED] hardware features in there. So things like, for example, black when I tell you that we have all this wires available, if you figure out some new way of taking advantage of those wires.

That hardware feature. What's the software that can really utilize that? So you can just build a hardware feature, but it's not sufficient. You have to have a software that do that, too. So what's the coupling? A lot of things that has to be that nice interconnection there.

And I think there are new people who want mobile devices. A great program. If we would look at parallelizing compilers or parallelization as I said before, was always geared to this high performance weenies, who just wanted to get their global simulation going as fast as possible. Who has a multi-million dollar supercomputer waiting.

It's very different having things like great programmers who want these things. How do you cater to them? They are very different. The key thing is how can we achieve parallelism without burdening the programmer? Because we don't, in this class we barely burden you guys and then you realize how hard it is. And how much complexity it adds on top of the algorithm, just to get it working parallel.

So how do you reduce some of the burden? Perhaps not completely, but reduce that burden. I'm going to skip some of this. I'll make it available on the site. I want to have some time for discussion.

In streaming, we talked about it, we tried to do some of that. Tried to come up with the

programming model where it will give you some additional benefits for the program, at the same time you can actually get multicore performers. And, of course, we saw that in compilers we can actually get good performance in then.

One thing that I did a long time ago, is this SUIF parallelizing compiler. What we did was we can automatically parallelize FORTRAN programs all the way. This kind of almost worked in the 90's.

We, at some point, spec bench marks were the way to measure how good your world is, and every company had 20 engineers trying to get 5% of spec. At the time our spec performance was about 200. Wire registered, we did this compiler, that we could expect bench mark performance at 800, by just parallelizing. And in fact we made this t-shirt that said, spec red and the number. That's all the t-shirt said. And, in fact, I was working in a few places, including the grocery store at Palo Alto, and people stopped me and said, how did you do that?

It's only on this kind of weenie things. And I had in the grocery story, I had half an hour conversation with [OBSCURED] of how -- he was, like, hmm. The only thing, spec 823. That's the only two things the t-shirt said, people knew what it is.

But, there's problems. The compiler was not robust. The compiler can be used by the people who wrote the compiler, because we knew all the things in there. The thing about that is it worked in a way that if program worked, you get 8x improvement. You change one line of a code somewhere, you went from 8x to 1x. That, I think, stopped parallelizing around the slow [OBSCURED].

So now, normal programmer can now understand what happens. We go and analyze it and afterward say aha, this is how to change the program, how to change the compiler. So we just kind of developing program and compiler at the same time, and we could use this thing. You cannot ignore the fact that this is happening. Because there's so much of a difference in performance. That makes it even a curse.

I mean, if you only get 4x improvement probably things would have been better. We should have probably capped the performance or something.

And clients, in those days, were impossible to with. You saw a very good example in last lecture. OK. Performance at any cost. I don't want any of these tools if it slows down at any time. So how we deal with those kind of people. It's hard.

In multiprocessor, communication was expensive. If you make a wrong decision, you pay a huge price. Basically slow the program down. You go from being 1x, you actually go from 0.5x. And worse. And you had to always deal with Moore's law.

Why do I need to go all these complicated compiling? Can I just wait six months and won't I get it? They're probably right at that time. And another problem, what you call a dogfooding problem. The reason that object oriented languages is very successful is everybody who did the object oriented language, in order to get accepted as a reasonable language, had to write their own compiler in that language.

And the compiler is a very good way of really pushing object oriented concepts. And so almost every object orientated language wrote their compiler. That's a really big thing you have to write. And by the time you write the compiler you realize all the problems of the language, and you feel equal, because you write a few hundred thousand line programming of language. Except all the people who were doing high performance, we were catering for somebody else.

What that means is we can deal with this weirdness, and say, yeah it's no problem, nobody -- but those are the things that really kill the users in there. We didn't have the users perspective.

And another thing that today is these things are really different. Because you have complex data structures, complex control flow, complex build processes, aliasing type unsafe languages. All this cool things we used to do, just became a hundred times harder, because of all those. People fell in love with data structures. And they say, oh, they were so nice. We knew exactly the same size thing, it doesn't change, I could analyze it. But malloc data structures, it's just and very complex trees and doubly linked lists was just impossible to deal with.

We want to go in the days where everything is a nice simple array But I guess the paths for the compiler became much harder in those days. So I think compilers are a critical thing to deal with. I think if you have to improve in multicores we had to deal with compilers.

And the sad thing is some of those things people are trying to do it today by hand, we could do it with the compilers 15 years ago. We kind of lost that technology. And people are doing this by hand again today. And go back and dust off this technology and bring it back in there.

Best case, we might automate everything. And say, don't worry, do what you are doing today

and just keep doing it and we'll get multicore performance. I don't think that's going to be that realistic. But in worse case, at least we can help the darn programmers. Tell them what's good, what to do when they do something wrong. How can you build those kinds of things. I think that's very important.

I think in tool side, we need tools. I mean you realize after programming on [OBSCURED] that was really nice to have your tools. Everybody can probably come up with some tools additions. And I think tools, on this, is pretty ancient. How do you come up with really good tools that the normal programmer can use them? Figure out problems, figure out debugging issues and stuff like that, and get through together.

We need Eclipse type thing for platform for multicores. We have a lot of nice plug-ins come about that actually help you to do that. Another interesting thing, this is about this dogfooding problem. Normally what you come up with language compiler tool, you do implementation, you do evaluation of that and you cycle through that.

And evaluation says, ok, you develop a program, you debug the program for functionality. Performance debugging, evaluate, and everything kind of go around that. I mean there's a process here. And you'd rather realize somethings are very hard to do, somethings are easy. And you need that to basically evolve that.

And the problem is -- I'm contrasting two things. If you look at something like CAD tools. CAD tools were developed by a bunch of guys sitting in places, used by some very different group of people. Because of that, today's CAD tools are horrendous to use. Everyone hates them.

Because the people who developed CAD tools never thought about users. They were indifferent. And they just have to get the functionality out and they were happy after that. Whereas, object oriented languages were more thought about as the how the best way to use it. Very much of a programmer-centric phase. And in that end, you develop a system that's a lot easier to use, a lot of people use it, more acceptance in there.

The problem with high performance languages is they really sit in the CAD tool way. Because the people who write high performance languages never really use it. People who write the compilers and languages are not the ones who are developing the high performance programs. No high performance language ever wrote the compiler in that language. They say, oh, that's different domain, we don't know how to do that.

By doing that, you really never expose the problems in there, and I think the key thing is how do you go about doing that? This is a hard problem. Because how do get two different groups together? Perhaps multicores, what you to do is you probably everybody who write the compiling tool, actually have to write it in the thing that you are doing, because you need the parallelism. At that point you realize all the problems with the tool.

So I'm going to the skip on this side.

Another interesting, very interesting problem, that people don't pay attention is the migration of the dusty deck. Millions if not billions of lines of code out there, written in old styles way. You've can't just say, look, Microsoft, just go rewrite everything new with my spiffy new system. That doesn't work. How to get these people to migrate out? That might not mean doing everything automatically, but how can you even help -- the people who wrote the code is probably gone, left the company, probably dead. How do you help somebody move some of these things to you.

That is the rate of evolution. You don't have to rewrite everything from scratch.

What kind of tools that you can use? And these applications are still in use. Sometimes source code is even available, but programs are gone. But the interesting thing is, some of those things, applications have bugs that have become features.

So I'll give you a very good example. And this happened to Microsoft. So at Microsoft at some point -- if you know Word, Word does algorithmic pagination. That means how to lay out the pages, and how to break pages and stuff like that. So they had the algorithm in there. And they were switching versions, and they said look, this algorithm is now too old. It's been there for five, six years. Rewrite it.

And of course you can assume what the spec is. There's a nice spec to that, a few pages. They said, go rewrite that. And someone wrote a very nice pagination algorithm. But the problem is, when they use that algorithm, a lot of old documents broke. That means they didn't paginate right. So the old document, when you open it, looks very different. And they said, darn, what's going on? What they realized is, the old implementation had a lot of bugs. Subtle things. The guide didn't really conform to the standard. So it didn't break the word exactly right, if at some point it has a break and space, it would probably put it on the next line. A few things that didn't really --

And what they had to do is, they had to spend a lot of time discovering bugs and adding them to the spec. So they did that. Because otherwise, all the previous program doesn't work, the previous files will not look right. And they went through this entire process of trying to discover and trying to figure out what [OBSCURED] instead of what the text of the specs said. They're out of sync.

And so a lot of times, there's that problem. So you see we have something old, and we want to get it new. It doesn't mean that you know the functionality. The functionality that was implemented is the spec, or the functionality that you wrote about. Because that's a little bit subtly different. So how do you go and discover that. I think that's a very interesting and important problem.

And I think there are many reasons that run such things like creating test cases, extracting invariants, things like failure oblivious computing type, if you had listened to professor Rabbah's talk . Things like that can help take these old things and migrate. Can use tools to do that. I did a lot of interesting research on that.

OK, so that's my take on languages and compilers, where interesting things have happened. I want to talk about revolution, but I want to skip that first and go into this crossing the abstraction boundaries part, and then come back to revolution.

So the way the world works is, you have some kind of class of computing you want to do, and you have some atoms that need to run that computation. That's basically the end to end of any process. But if I said, here's some silicon, and here's my algorithm, go do that, no single person can basically build that. OK? You can't take silicon, and go through a semiconductor process, build a process, design all those things, language -- it doesn't happen. Someday probably it was possible to have one person know everything, but you can't.

So what we have done deal with build abstraction boundaries. We have compilers, languages, instruction, microarchitecture, layout, design rules, process, materials science -- basically, everything is separated out. Every time one gets too complicated, we kind of divide it in half, and keep adding and adding. And we have this big stack of things.

So at the beginning, then you break up, people kind of knew both sides, and they went into one side and kind of broke that up. And so people knew a lot of things. But the next generation comes, they only know that. The guy, someone who's going to be expert in microarchitecture. You only know microarchitecture. He has no idea what the languages are, because that world

of his is complicated. What we have is a compartmentalized world that was broken up as we went in the last 40 years, as things progressed, the right thing at that time, they figure out what layers to do.

What we have done is create the domain expert. There's one guy who probably knows about two things. He might know a little bit of design rules, and probably process a little bit. Another person will know a little bit of design rules, layout, and microarchitecture. And there's someone who'll do microarchitecture and ISA. And somebody knows compilers and ISA, but has no clue what's happening. So what this has done is kind of entrenched some layering that really made sense about 30 years ago.

The problem now is, this is creating a lot of issues in here. So what you kind of really need is a way to break some of this layering. How do you do that? You need people who can cross multiple of these disciplines. So this is a thing for you. I mean, you might think you are an architect. But hey, you're in this class, you're learning a little bit of programming. Go learn a little bit of physics, what's in there. Just have a little bit of broadening in there. Then a lot of people, there are some people who are a lot more in the process side, but know a little bit about microarchitecture and compilers, somewhere in the middle. They are expert in the middle, but they are not just sitting happily in their own domain. Just trying to look at cross-cutting there.

And keep going up, in some sense. I kind of put myself in there, saying OK, I am in compilers, but I know a little bit of language, and I know a little -- I try to go down and down as much as possible. What you actually need is someone who knows everything really well. That is hard, but at least someone who can know the top and the bottom to redo the middle. Because my feeling is the way the middle is done is probably now too old. So I don't know that this is possible. It's not me. But if someone really understands the top and the bottom, and say OK, I can probably throw the middle out and redo the middle. And I think that might be an interesting revolution that might happen.

It's hard. There's so much information there. Some of the information, there are the layers out there, it doesn't make sense, today, the way things are. I mean for example, a lot of these layers had issues with wire delay. Because when wire delay was not an issue, these layers made perfect sense. But as wire delay came about, a lot of changes had to propagate across layers, and it was very hard, because people were domain experts.

So I think this is where the revolution is going to come. Somebody's going to [OBSCURED] the way you do compilers, the way you do architecture, the way you decide everything is wrong. I know the algorithms you want to run. I know what's available in very low [OBSCURED]. Let me see you put that together. A person, a group, whatever. Might have a beginning.

And so with that, I will go to the revolution side. So the way you look at revolutions, what are the far-out technologies? And sometimes revolutions just come from wishful thinking. I wish we can do this, and then if you can push hard, you might be able to get there.

Anybody wants to -- so in this talk I talk way too much. How about, come in? What do you think is going to happen? What do you want to work on? You guys are the ones who have to make this decision. Not me. I've already made a lot of decisions myself, 10, 15 years ago, what I'm going to do. You are in the process, you have a lot more choices to make. You haven't narrowed your horizon yet. So this affects a lot more on you than me at this point.

What do you think is going to change the world? What do you want to work on? I mean, you should work on something, I guess, that you think is going to have a huge impact. Have you thought about that? A lot of silence.

AUDIENCE: [OBSCURED]

PROFESSOR: So faster means very little latency. Latency is a hard problem. A lot of people who are trying to do latency [OBSCURED] And the revolution side people are looking at quantum can kind of do deal with latency.

AUDIENCE: Yeah.

PROFESSOR: Yes, I think there are a lot of revolutionary things you can do there. Especially taking advantage of the fact that you have a lot of wires in there. And I think people haven't really pursued that yet. And this is where the complexity will come in. Trying to basically deal with these, take advantage of these kind of things. I think there are a lot of opportunities there. And then that's the kind of evolutionary part. And then some people are looking at revolution, with quantum.

What else? I expected people to have a lot more interesting insights.

AUDIENCE: [OBSCURED]

PROFESSOR: I think the world is almost bifurcated. There are things that you can have with power plugs, and things that don't have power plugs. And I think people who have power plugs still have some power issues, like what Google is finding out. You can't just keep wasting power. But as things start becoming more mobile, going into smaller and smaller devices, power and size becomes a much bigger thing than what more you can do. You don't become performance, I mean, you want some level of performance; you need to get a video. But the minute you get the video, you don't want to have HD TV. You just say, OK, get the media onto my cellphone screen. I'll be happy with that speed. Just do it. As low power as possible, as [OBSCURED] possible. So I think that's pushing there.

So for example, in for a long time what has happened is there's a trickle down economy. Things that start happening in a very high end system and kind of trickle down to the low end. But there might be a diversion here. There might be a different part coming. Because that technology might be fundamentally different.

AUDIENCE: So [OBSCURED] also changing from the software side is this notion of perpetual data. I've been using this mash-up code, you put it up on the web, lots of people use it. So it's pushing toward scripting light languages or whatever.

PROFESSOR: So this is an interesting obsession I have. So [OBSCURED]

And then a few years ago I had a start up, and I just again had the time to try it, not large , some large piece of code. What I realized was, what programming had become, is the biggest, best tool for programming at that time was Google. Because I was trying a bunch of things with Windows. And everything I want, there was a function. And if I can find that function, my work is a factor of 100 smaller. So I spend more time than actually thinking about algorithms, how the lists should be, or how my data structures should be, trying to Google it. I said, darn, I want to do this. Where's that function? Where can I find that function in this entire big library.

So what that means is there's kind of this missing layer in there. Because what has happened is, our language is a level of abstraction. And then we have built these libraries with no kind of support at all. I mean, the compilers' languages don't really support the libraries. Because it's just built out of basic blocks. And keeping it, can you have better basic blocks, can you have more nice abstraction? Instead of saying, OK, here's 5,000 libraries with 250,000 different functions that you can use for compiling. Can you build it in a nicer way?

AUDIENCE: [OBSCURED] interface, your input, your output. And then you publish your code and then

Google indexes it for you. And then your program is just of Google searches.

PROFESSOR: I think, if you think about what all the Windows source base is, it's like that. I mean, in the company we were doing, during the [OBSCURED], there was this group of people who understood every line of code they wrote, because they were writing a virtualization system that runs on the Windows. So for them, every line was probably -- Interesting thing is, when that group started, that entire piece of code was about 50,000 lines. Now it's about 65,000 lines. But we had about 10 people working on that. That means these things get revised. You don't keep at it.

There's another group of people who's doing user interface and stuff like that. they have alphabet soup of different libraries. They're using Ajax, this, that -- I don't even know what they are using. The amount of code they write is immense. The amount of things they have to know is immense. Each of them are not that completed. But if you use a bad thing, you have a bad looking interface, clunky, slow, whatever, if you figure out the right three function calls, things look much better. That's a very different type of programming in there.

So for them, it's basically knowledge. How much do you know. Just like, it's almost like I remember my college days, friends who were doing things like that in medical school. So you need to know this entire thing about all the different muscles in the body. And there's this big thick book you just study. It's almost like that. You just take this big Windows manual that are a few hundred pounds heavy, and you look at every page, and figure out what you know. That's a really strange way of coding. But, I mean, algorithmically, I think what they were doing is pretty simple. I don't think they every designed a data structure. Every data structure they used came from something. But they had to figure out the right data structure. AUDIENCE:

This is going in a totally different direction, but in terms of graphics in video games, perhaps getting artists who are aware of both the hardware and the software, and so can design more for the capabilities of the machine, and can make more visually appealing --

PROFESSOR: I think that's the are that has a very different set of communities. Artists [OBSCURED]. Not even trained engineers. And they're sitting in one, and they have this neat feature, they want fire to look natural, or whatever it is. And then there's the programmers and hardware people, that who knows what the capability is. I think that's what we're trying to push in Mike Eckham's talk, and say how do you go about it? How do you know when a new thing comes, what is doable?

Like for example, what ATI does, ATI has a group called the research group. What they do is, every time a new chip comes, they actually go and figure out all the things you can do with that. OK, how can you do rain in this one? So they write a lot of -- those of a little bit of an artistic bent, but very hard-core programmer who kind of figure out what's doable. And they make that available and say here, this is what you can do. And a lot of people kind of emulate that. They say, wait a minute, how did they do that? Then kind of figure out, OK, now I have this new hardware, and I'm able to do something completely new.

I think that's, pushing there. I mean, I think this is where interesting revolutions come about. One of my firm beliefs is that a lot of good research is not in one area, as I pointed out. It's kind of looking at two distinct things and kind of bringing them together. Bringing knowledge from one to another and mixing them together. I think that can have huge impact. And that's a place that I think can have huge impact.

What other things that will have big visual impact? That the next hardware can enable? You can keep adding more big servers and stuff like that, but it's another thing. What other things can can have big impact? AUDIENCE:

[OBSCURED] The same is true for biologists, physicists, chemists, end users. So maybe the thing to do is work on language generators. And easy way for me as an end user to express language [OBSCURED] that I want and give me a way to generate it

PROFESSOR: So that's a lot of things. For example, I was talking to this computational biologist. And what he said was, for computer science, the best thing that happened from a biology point of view is Excel only had 32,000 cells. Because the biologists keep using Excel, and they say that --

AUDIENCE: [OBSCURED]

PROFESSOR: 65,000 cells. And they ran into this and said, jeez, I can't do anything more in that. Now what do I do? And then suddenly they had to start looking at programming and stuff like that, because they can't have an Excel spreadsheet doing that. So that's a lot of biologists actually had to switch.

So I think, interesting thing about computer sciences, there are a lot of fundamental sciences still. But I think we have grown and ran so fast developing that, what's lagging behind that is applications. How to apply that to biology, gaming, stuff like that. And I think a lot of interesting things, as computer scientists, we can do is to bring that knowledge in an applied community.

Really, you can have a huge impact and fundamentally change things if you apply that knowledge. Both theoretical things you have to get out, as well as using the powers of the computers we've got in the right way. I think a lot of times, people are taking a few days to do two times better, five times better than kind of seeing fundamentals [OBSCURED].

I think this is an interesting topic. I think I'm going to stop soon. And I think it's interesting, especially for you guys. Because you are probably going into an area, looking at a field, trying to get expert at something.

There's a lot of times you get seduced to become an expert in what's hot today. But the interesting thing is, especially if you are doing a PhD, or if you do something that takes four or five years for you to mature in that area, is it going to be the hardest thing in four or five years? So multicores are hard today. So should I recommend that everybody jump and become expert and try to do a PhD in multicores? Probably not. Because five or six years down the line, some of these things might be solved. they'd better get solved, or else we're in big trouble.

But the thing is, what's going to be the problem there? I see one thing that's really hard, it's almost art, is to kind of picture your [OBSCURED] and then if your lucky And if you're lucky and you do a good job in that prediction, you've come about, and by the time you get to a point that you become expert, you've become expert in the thing that's most important. It's not an easy thing to do. And a lot of times, nobody can claim that they have a technique to do that, too. A lot of people who claim, they discount how lucky they were. There's a large amount of luck associated with it.

But a key thing is to kind of follow that trend, identify, and do that. That's the technique that I think every one of us, every one of you should do a lot. OK. With that, do you have something about the --

SPEAKER: Yes. A couple of things about -- Anybody here want direct access to a Playstation 3? Just one? Two? So come by, actually if you have time now, we'll talk about that and figure out a way of getting you access to it. The final competition's going to be this Thursday.