MICHAEL PERRONE: So my name's Michael Perrone. I'm at the T.J. Watson Research Center, IBM research. Doing all kinds of things for research, but most recently-- that's not what I want. There we go. Most recently I've been working with the cell processor for the past three years or so. I don't want that. How's that? And because I do have to run out for a flight, I have my e-mail here if you want to ask me questions, feel free to do that. What I'm going to do in this presentation is as Saman suggested, talk in depth about the cell processor, but really it's still going to be just the very surface because you going to have a month to go into a lot more detail. But I want to give you a sense for why it was created, the way it was created, what it's capable of doing, and what are the programming considerations that have to be taken in mind when you program. Here's the agenda just for this section, Mike, of this class. I'll give you some motivation. This is going to be a bit of a repeat, so I'll go through it fairly quickly.

I'll talk about the design concepts, hardware overview, performance characteristics, application affinity-- what good is this device? Talk about the software and this I imagine is one of the areas where you're going to go into a lot of detail in the next month because as you suggested, the software really is the issue and I would actually go a little further and say, why do people drive such large cars in the U.S.? Why do they waste so much energy? The answer is very simple. It's because it's cheap. Even at $3 a gallon, it's cheap compared to say, Europe and other places. The truth is it's the same thing with programmers. Why did programmers program the way they did in the past 10, 20 years? Because cycles were cheap. They knew Moore's law was going to keep going and so you could implement some algorithm, you didn't have to worry about the details, as long as you got the right power law-- if you got your n squared or n cubed or n log n, whatever behavior. The details, if the multiplying factor was 10 or 100 it didn't matter. Eventually Moore's law would solve that problem for you, so you didn't have to be efficient. And I think I've spent the better part of three years trying to fight against that and you're going to learn in this class that, particularly for multicore you have to think very hard about how you're going to get performance.

This is actually the take home message that I want to give. I think it's just one or two slides, but we really need to get to these because that's where I want to get you thinking along the right lines. And then there's a hardware consideration, we can skip that. All right, so where have all the gigahertz gone, right? We saw Moore's law, things getting faster and faster and the answer is I have a different chart that's basically the same thing. You have relative device performance on this axis and you've got the year here. And different technologies were growing, growing, growing, but now you see they're thresholding. And you go to conferences now, architecture

conferences, and people are saying, Moore's law is dead. Now, I don't know if I would go that far and I know there are true believers out there who say, well maybe the silicon on the insulator technology is dead, but they'll be something else. And maybe that's true and maybe that is multicore, but unless we get the right programming models in place it's not going to be multicore.

Here's this power density graph. Here we have the nuclear reactor power up here and you see pentiums going up now. Of course, there's a log plot, so we're far away, but on this axis we're not far away. This is how much we shrink the technology, the size of those transistors. So if we're kind of going down by 2 every 18 months or so, maybe it's 2 years now, we're not so far away from that nuclear reactor output. And that's a problem. And what's really causing that problem? Here's a picture of one of these gates magnified a lot and here's the interface magnified even further and you see here's this dielectric that's insulating between the 2 sides of the gate-- we're reaching a fundamental limit. A few atomic layers. You see here it's like 11 angstroms. What's that? 10, 11 atoms across? If you go back to basic physics you know that quantum mechanical properties like electrons, they tunnel, right? And they tunnel through barriers with kind of an exponential decay. So whenever you shrink this further you get more and more leakage, so the current is leaking through. In this graph, what you see here is that as this size gets smaller, the leakage current is getting equivalent to the active power. So even when it's not doing anything, this 65 nanometer, the technology is leaking as much power as it actually uses. And eventually, as we get smaller, smaller we're going to be using more power, just leaking stuff away and that's really bad because as Saman suggested we have people like Google putting this stuff near the Coulee Dam so that they can get power. I deal with a lot of customers who have tens of thousands of nodes, 50,000 processors, 100,000 processors. They're using 20 gigabytes-- sorry, megahertz. No, megawatts, that's what I want to say. It's too early in the morning. Tens of megawatts to power their installations and they're choosing sites specifically to get that power and they're limited. So they come to me, they come to people at IBM and they say, what can we do about power? Power is a problem. And that's why we're not seeing increasing the gigahertz.

Has this ever happened before? Well, I'm going to go to this quickly, yes. Here we see the power outage output of a steam iron, right there per unit area. And something's messed up here. You see as the technology changed from bipolar to CMOS we were able to improve the performance, but the heat flux got higher again and that begs the question, what's going to happen next? And of course, IBM, Intel, AMD, they're all betting this multicore. And so there's an opportunity from a business point of view. So now, that's the intro. Multicore: how do you deal with it?

Here's a picture of the chip, the cell processor. You can see these 8 little black dots. They're local memory for each one of 8 special purpose processors, as well as a big chunk over here, which is a ninth processor. So this chip has 9 processors on board and the trick is to design it so that it addresses lots of issues that we just

discussed. So let me put this in context, cell was created for the Sony Playstation 3. It started in about 2000 and there's a long development here until it was finally announced over here. Where was it first announced? It was announced several years later and IBM recently announced a cell blade about a year back and we're pushing these blades and we're very much struggling with the programming model. How do you get performance while making something programmable? If you go to customers and they have 4 million lines of code, you can't tell them just port it and it'll be 80 person years to get it ported, 100 person years more. And then you have to optimize it. So there are problems and we'll talk about that. But it was created in this context and because of that, this chip in particular, is a commodity processor. Meaning that it's going to be selling millions and millions. Sony Playstation 2 sold an average of 20 million units each year for 5 years and we expect the same for the Playstation 3. So the cell has a big advantage over other multicore processors like the Intel Woodcrest, which has a street price of about $2000 and the cell around 100. So not only do we have big performance improvements, we have price advantages too because of that commodity market.

All right, let's talk about the design concept. Here's a little bit of a rehash of what we discussed with some interesting words here. We're talking about a power wall, a memory wall and a frequency wall. So we've talked about this frequency wall. We're hitting this wall because of the power really and the power wall people just don't have enough power coming into their buildings to keep these things going. But memory wall, Saman didn't actually use that term, but that's the fact that as the clock frequencies get higher and higher, memory appeared further and further away. The more cycles that I have to go as a processor before the data came in. And so that changes the whole paradigm, how you have to think about it. We have processors with lots of cache, but is cache really what you want? Well, it depends.

If you have a very localized process where you're going to bring something into cache and the data is going to be reused then that's really a good thing to do. But what if you have random gather and scatter of data? You know, you're doing some transactional processing or whatever mathematical function you're calculating is very distributed like an FFT. So you have to do all sorts of accesses through memory and it doesn't fit in that cache. Well, then you can start thrashing cache. You bring in one integer and then you ask the cache for the next thing, it's not there, so it has to go in and so you spend all this time wasting time getting stuff into cache. So what we're pushing for multicore, especially for cell is the notion of a shopping list. And this is where programability comes in and programing models come in. You really need to think ahead of time about what your shopping list is going to be and the analogy that people have been using is you're fixing something in your house, you're pipe breaks. So you go and say, oh, I need a new pipe. So you go the store, you get a pipe. You bring it back and say, oh, I need some putty. So you go to the store, you get some putty. And oh, I need a wrench. Go to the store-- that's what cache is. So you figure out what you need when you need it.

In the cell processor you have to think ahead and make a shopping list. If I'm going to do this calculation I need all these things. I'm going to bring them all in, I'm going to start calculating. When I'm calculating on that, I'm going to get my other shopping list. So that I can have some concurrency of the data load with the computes. I'm going to skip this here. You can read that later, it's not that important.

Cell synergy, now this is kind of you know, apple pie, motherhood kind of thing. The cell processor was specifically designed so that those 9 cores are synergistic. That they interoperate very efficiently. Now I told you we have 8 identical processors, we call those SPEs. In the ninth processor its the PPE. It's been designed so that the PPE is running the OS and it's doing all the transaction file systems and what not so that these SPEs can focus on what they're good at, which is compute. The whole thing is pullled together with an element interconnect bus and we'll talk about that. It's very, very efficient, very high bandwidth bus.

Now we're going to talk about the detail hardware components. And Rodric somewhere, there you are, asked me to actually dig down into more of the hardware. I would love to do that. Honestly, I'm not a hardware person. I'll do the best I can, perhaps at the end of the talk we'll dig down and show me which slides you want. But I've been dealing with this for so long that I can do a decent job.

Here's another picture of the chip. It has lots of transistors. This is the size. We talked about the 9 cores, it has 10 threads because this power processor, the PPE has 2 threads. Each of these are single threaded. And this is the wow factor. We have 200 gigaflops, over 200 gigaflops of single precision performance on these chips. And over 20 gigaflops of double precision and that will be going up to 100 gigaflops by the end of this year. The bandwidth to main memory is 25 gigabytes per second and up to 75 gigabytes per second of I/O bandwidth. Now this chip really has tremendous bandwidth, but what we've seen so far-- particularly with the Sony Playstation and I think you may have lots of them here, the board is not designed to really take advantage of that bandwidth. And even the blades that IBM sells really can't get that type of bandwidth off the blade. And so if you're keeping everything local on the blade or on the Playstation 3 then you have lots of bandwidth internally. But off blade or off board you really have to survive with something like a gigabyte, 2 gigabytes in the future.

And this element interconnect bus I mentioned before has a tremendous bandwidth, over 300 gigabytes per second. The top frequency in the lab was over 4 gigabytes-- gigahertz, sorry. And it's currently running when you buy them at 3.2 gigahertz. And actually the Playstation 3's that you're buying today, I think, they only use 7 out of the 8 SPEs. And that was a design consideration from the hardware point of view because as these chips get bigger and bigger, which is if you can't ratchet up the gigahertz you have to spread out. And so as they get bigger, flaws in the manufacturing process lead to faulty units. So instead of just throwing away things, if one of these SPE is bad we don't use it and we just do 7. As the design process gets better by the end of this year they'll be using 8. The blades that IBM sells, they're all set up for 8

OK, so here's a schematic view of what you just saw on the previous slide. You have these 8 SPEs. You have the PPE here with this L1 and L2 cache. You have the element interconnect bus connecting all of these pieces together to a memory interface controller and a bus interface controller. And so this MIC is what has the 25.6 gigabytes per second and this BIC has potentially 75 going out here. Each of these SPEs has its own local store. Those are those little black dots that you saw, those 8 black dots. It's not very large, it's a quarter of a megabyte, but it's very fast to this SXU, this processing unit. It's only 6 cycles away from that unit. And it's a fully pipelined 6 so that if you feed that pipeline you can get data every cycle.

And here, the thing that you can't read because it's probably too dark is the DMA engine. So one of the interesting things about this is that each one of these is a full fledged processor. It can access main memory independent of this PPE. So you can have 9 processes or 10 if you're running 2 threads here, all going simultaneously, all independent of one another. And that allows for a tremendous amount of flexibility in the types of algorithms you can implement. And because of this bus here you can see it's 96 bytes per cycle and we're at 3.2 gigahertz. I think that's 288 gigabytes per second. These guys can communicate to one another across this bus without ever going out to main memory and so they can get much faster access to their local memories. So if you're doing lots of computes internally here you can scream on this processing; really, really go fast. And you can do the same if you're going out to the memory interface controller here to main memory, if you sufficiently hide that memory access. So we'll talk about that.

All right, this is the PPE that I mentioned before. It's based on the IBM power family of processors, it's a watered down version to reduce the power consumption. So it doesn't have the horse power that you see in say a Pentium 4 or even-- actually, I don't have an exact comparison point for this processor, but if you take the code that runs today on your Intel or AMD, whatever your power and you recompile it on cell it'll run today-- maybe you have to change the library or two, but it'll run today here, no problem. But it'll be about 60% slower, 50% slower and so people say, oh my god this cell processor's terrible. But that's because you're only using that one piece. So let's look at the other-- OK, so now we go into details of the PPE. Half a megabyte of L2 cache here, coherent load stores. It does have a VMX unit, so you can do some SIMD operations, single instruction multiple data instructions. Two-way hardware multithreaded here. Then there's an EIB that goes around here. It's composed of four 16 byte data rings. And you can have multiple, simultaneous transfers per ring for a total of over 100 outstanding requests simultaneously. But this slide doesn't-- this kind of hides it under the rug. There's a certain topology here. And so these things are going to be connected to those 8 SPEs. And depending on which way you send things, you'll have better or worse performance. So some of these buses are going around this way and some are going counterclockwise. And because of that you have to know who you're communicating if you want have real high efficiency. I haven't seen personally cases where it made a really big difference, but I do know that

there's some people who found, if I'm going from here to here I want to make sure I'm sending things the right way because of that connectivity. Or else I could be sending things all the way around and waiting.

AUDIENCE: Just a quick question.

MICHAEL PERRONE: Yes.

AUDIENCE: Just like you said you could complie anything on the power processor would be slower, but you can. Now you also said the cell processor is in itself a [INAUDIBLE] processor. Can I compile it in a C code just for that as well.

MICHAEL PERRONE: C code would compile. There's issues with libraries because the libraries wouldn't be ported to the SPE necessarily. If it had been then yes. This is actually a very good question. It opens up lots of things. I don't know if I should take that later.

PROFESSOR: Take it later.

MICHAEL PERRONE: Bottom line is this chip has two different processors and therefore you need two different compilers and it generates two different source codes. In principle, SPEs can run a full OS, but they're not designed to do that and no one's ever actually tried. So you could imagine having 8 or 9 OSes running on this processor if you wanted. Terrible waste from my perspective, but OK, so let's talk about these a little bit. Each of these SPEs has, like I mentioned this memory flow controller here, an atomic update unit, the local store, and the SPU, which is actually the processing unit. Each SPU has a register file with 128 registers. Each register is 128 bits. So they're native SIMD, there are no scalar registers here for the user to play with. If you want to do scalar ops they'll be running in those full vector registers, but you'll just be wasting some portion of that register. It has IEEE double precision floating point, but it doesn't have IEEE single precision floating point. It's curiosity, but that was again, came from the history. The processor was designed for the gaming industry and the gamers, they didn't care if it had IEEE. Who cares IEEE? What I want is to have good monsters right on the screen.

And so those SIMD registers can operate bitwise on bytes, on shorts, on four words at a time or two doubles at a time. The DMA engines here, each DMA engine can have up to 16 outstanding requests in its queue before it stalls. So you can imagine you're writing something, some code and you're sending things out to the DMA and then all of a sudden you see really bad performance, it could be that your DMA egine has stalled the entire processor. If you try to write to that thing and then that queue is full, it just waits until the next open slot is available. So those are kind considerations.

AUDIENCE: You mean [UNINTELLIGIBLE PHRASE]

MICHAEL PERRONE: Yes.

AUDIENCE: It's not the global one?

MICHAEL PERRONE: Right. That's correct. But there is a global address space.

AUDIENCE: 16 slots each in each SPU.

MICHAEL PERRONE: Right. Exactly. Each MFC has its own 16 slots. And they all address the same memory. They can have a transparent memory space or they can have a partitioned memory space depending on how you set it up.

AUDIENCE: Each SPU doesn't have its own-- the DMA goes onto the bus, [UNINTELLIGIBLE] that goes to a connection to the [UNINTELLIGIBLE].

PROFESSOR: You can add this data in the SPUs too. You don't have to always go to outside memory. You can do SPU to SPU communication basically.

MICHAEL PERRONE: Right. So I can do a DMA that transfers memory from this local store to this one if I wanted to and vice versa. And I can pull stuff in through the-- yeah, I mentioned this stuff. Now this broadband interface controller, the BIC, this is how you get off the blade or off the board. It has 20 gigabytes per second here on I/O IF. In 10 over here--I'm sorry, 5 over here. I'm trying to remember how we get up to 70. This is actually two-way and one is 25 and the other one's 30. That gets you to 55. This should be 10 and now, what's going on here? It adds up to 75, I'm sure. I'm sure about that. I don't know why that says that. But the interesting thing about this over here, this I/O IF zero is that you can use it to connect two cell processors together. So this is why I know it's really 25.6 because it's matched to this one. So you have 25.6 going out to main memory, but this one can go to another processor, so now you have these two processors side-by-side connected at 25.6 gigabytes per second. And now I can do a memory access through here to the memory that's on this processor and vice versa.

However, If I'm going straight out to my memory it's going to be faster than if I go out to this memory. So you have a slight NUMA architecture and nonuniform memory access. And you can hide that with sufficient multibuffering. So I know that this is 25 and I know the other one's 30. I don't know why it's written as 20 there.

AUDIENCE: Can the SPUs write to the [UNINTELLIGIBLE PHRASE]?

MICHAEL PERRONE: Yes, they can read from it. I don't know if they can write to it. In fact, that leads to a bottleneck occurring. So I happily start a process on my PPE and then I tell all my SPEs, start doing some number crunching. So they do that. They get access to memory, but they find the memory is in L2. So they start pulling

from L2, but now all 8 are pulling from L2 and it's only 7 gigabytes per second instead of 25 and so you get a bottleneck. And so what I tell everybody is if you're going to initialize data with that PPE make sure you flush your cache before you start the SPEs. And then you don't want to be touching that memory because you really want to keep things-- stuff that the SPEs are dealing with-- you want to keep it out of L2 cache.

Here there's an interrupt controller. An I/O bus master translation unit. And you know, these allow for messaging and message passing and interrupts and things of that nature. So that's the hardware overview. Any questions before I move on?

So why's the cell processor so fast? Well, 3.2 gigahertz, that's one. But there's also the fact that we have 8 SPEs. Each 8 SPEs have SIMD units, registers that are running so they can do this parallel processing on a chip. We have 8 SPEs and each one are doing up to 8 ops per cycle if you're doing a mul-add. So you have four mul-adds for single precision. So you've got 8, that's 64 ops per cycle times 3.2. You get up to 200 gigaflops per cycle, 204.8. So that's really the main reason. We've talked about this stuff here. This is an image of why it's faster. Instead of staging and bringing the data through the L2, which is kind of what we were just discussing and having this PU, this processing unit, the PPE manage the data coming in, each one can do it themselves and bypass this bottleneck. So that's something you have to keep in the back of your mind when you're programming. You really want to make sure that you get this processor out of there. You don't want it in your way. Let these guys do as much of their own work as they can.

Here's a comparison of theorectical peak performance of cell versus freescale, AMD, Intel over here. Very nice. That's the wow chart. The theoretical peak, this is in practice, what did we see? I don't know if you can read these numbers but what you really want to focus on is the first and last columns. This is the type of calculation, high performance computing like matrix multiplication, bioinformatics, graphics, security, it was really designed for graphics. Security, communication, video processing and over here you see the advantage against an IA 32, a G5 processor. And you see 8x, 12x, 15, 10, 18x. Very considerable improvement in performance. In the back-- question?

AUDIENCE: [UNINTELLIGIBLE] previous slide, how did it compare to high [UNINTELLIGIBLE PHRASE]?

MICHAEL PERRONE: All right, so you're thinking like a peak stream or something like that?

AUDIENCE: Any particular [UNINTELLIGIBLE PHRASE]. The design of the SPUs is very reminiscent of [UNINTELLIGIBLE PHRASE].

MICHAEL PERRONE: So I believe, and I'm not well versed in all of the processors that are out there. I think that we still have a performance advantage in that space. You know, I don't know about Xilinx and those kind of things-

- FPGAs I don't know, but what I tell people this is there's a spectrum. And at one end you have your general purpose processors. You've got your Intel, you've got your Opteron whatever, your power processor. And then at the other and you've got your FPGAs and DSPs and then maybe over here, somewhere in the middle you've got graphical processing units. Like Nvidia kind of things. And then somewhere between those graphics processing processors and the general purpose processors you've got cell. You get a significant improvement in performance, but you have to pay some pain in programming. But not nearly as much as you have to do with the graphics processors and no where near the FPGAs, which are just every time you write something you have to rewrite everything. Question?

AUDIENCE: Somewhat related to the previous question, but with a different angle. I always figured anyone could do a [INAUDIBLE], so that's why I ask about FFTs. Are they captured on the front or otherwise [UNINTELLIGIBLE]

MICHAEL PERRONE: Yeah, so this is actually one of the things I spent a lot of time on for FFTs. I spent a lot of time with the petroleum industry. They take these enormous boats, they have these arrays that go 5 kilometers back and 1 kilometer wide, they drag them over the ocean, and they make these noises and they record the echo. And they have to do this enormous FFT and it takes them 6 months. Depending on the size of the FFT it can be anywhere from a week to 6 months, literally.

AUDIENCE: [UNINTELLIGIBLE].

MICHAEL PERRONE: Sorry?

AUDIENCE: Is this a PD FFT?

MICHAEL PERRONE: Sometimes I do too, but they do both. I've become somewhat of an expert on these FFTs. For cell the best performance number I know of is about 90 gigaflops of FFT performance. You know, that's very good. Yeah, it's like 50% of peak performance. You know, it's easy to get 98% with [? lynpacker ?] or [? djem ?] on a processor like this and we have. We get 97% of peak performance, but it's a lot harder to get FFTs up to that.

AUDIENCE: Well, then I'll [INAUDIBLE] the next questions then which is somehow or another you get the FFT performance, you've got to get the data at the right place at the time. [UNINTELLIGIBLE] So you've personally done that or been involved with that?

MICHAEL PERRONE: Right, so we do a lot of tricks. I can show you another slide or another presentation that we talk about this, but typically the FFTs that we work with are somewhere from a 1024 to 2048, that's square. And so it's possible to take say, the top 4 rows-- in the case of 1024, four rows complex, single precision I think is 16 kilobytes. That fits into the local store very nicely. So you can stop multibuffering. You bring in one, you start computing on it. While you're computing on those 4 in a SIMD fashion across the SIMD registers you're bringing in

the next one. And then when that one's finished you're writing that one out while your computing on the one that arrived and while you're getting the next one. And since you can get the entire 1024 or 2000 into local store, you're only 6 cycles away from any element in it. So it's much, much faster. We also did the 16 million element FFT. 1D, yeah and we did some tricks there to make it efficient, but it was a lot slower.

AUDIENCE: [UNINTELLIGIBLE PHRASE] would have to be a lot slower by the need for the problem. [UNINTELLIGIBLE PHRASE]

MICHAEL PERRONE: What I remember it was fifteen times faster than a power 5. It might have been a power 4, I don't remember, sorry.

I might want to skip this one. I think I'm going to skip this one.

AUDIENCE: [UNINTELLIGIBLE PHRASE]

MICHAEL PERRONE: Right. Let's talk about what is the cell good for. You kind of have a sense of the architecture and how it all fits together. You may have some sense of the gotchas and the problems that might be there, but what did we actually applied to 2? I mean you saw some of that here. Here's a list of things that either we've already proven to ourself that it works well or we're very confident that it works well or we're working to demonstrate that it works well. Signal processing, image processing, audio resampling, noise generation. I mean, you can read through this list, there's a long list. And I guess there are a few characteristics that really make it suitable for cell. Things that are in single precision because you've got 200 gigaflops single and only 20 of double, but that will change as I mentioned. Things that are streaming, streaming through and so single processing is ideal where the data comes through and you do your compute and then you throw it away or you give out your results and you throw it away. Those are good. And things that are compute intensive, where you bring the data in and you're going to crunch on it for a long time, so things like cryptography where you're either generating something from a key and there's virtually no input. You're just generating streams of random numbers that's very well suited for this thing. You see FFTs listed here. TCPIP off load. I didn't put that there. There's actually a problem with cell today that we're working to fix that the TCPIP performance is not very good. And so what I tell people to use is open NPI. You know, so that over InfiniBand. The PPE processor really doesn't have the horse power to drive a full TCPIP sack. I'm not sure it has the horse power to do a full NPI stack either, but at least you have more control in that case.

The game physics, physical simulations-- I can show you a demo, but I don't know that we'll have time where a company called Rapid Mind, which is developing software to ease programmability for cell. Basically you take your existing scalar code and you instrument it with C++ classes that are kind of SPE aware. And by doing that, just

write your scalar code and you get the SPE performance advantage. They have this wonderful demo of these chickens. They've got 16,000 chickens in a chicken yard. You know, the chicken yard has varying topologies and the chickens move around and all 16,000 are being processed in real time with a single cell processor. In fact, the Nvidia card that was used to render that couldn't keep up with what was coming out of the SPEs. We we're impressed with that. We're happy with that. We showed it around at the game conferences and the gamers saw all these chickens and were like, this is really cool. How do I shoot them? So we said, you can't. But maybe in the next version.

But the idea that we've designed this so that it can do physical simulations, and this is maybe an entree for some of you people when you're doing your stuff. I don't know what kinds of things you want to try to do on cell, but I've seen people do lots of things that really have no business doing well on cell and they did very, very well. Like pointer chasing. I'm trying to remember. There are two pieces of work. One done by PNNL Fabritzio Petrini and he did a graph traversal algorithm. It was very much random access and he was able to parallelize that very nicely on Cell. And then there was another guy at Georgia Tech who did something similar for linked lists. And you know, I expect things to work well on cell if they're streaming and they have very compute intensive kernels that are working on things, but those are two examples where they're very not very compute intensive and not very streaming. They're kind of random access and they work very well.

Over here, target applications. There are lots of areas where we're trying to push cell forward. Clearly it works in the gaming industry, but where else can it work? So medical imaging, there's a lot of success there. The sysmic imaging for petroleum, aerospace and defense for radar and sonar-- these are all signal processing apps. We're also looking at digital content creation for computer animation. Very well suited for cell. This is kind just what I just said. Did I leave out anything?

Finance-- once we have double precision we'll be doing things with finance. We actually demonstrated that things work very well. You know, metropolis algorithms, Monte Carlo, Black shoals algorithms if you're familiar with these kind of things from finance. They tell us they need double precision and we're like, you don't really need double precision, come on. I mean, what you have is some mathematical calculation that you're doing and you're doing it over and over and over. And Monte Carlo there's so much noise, we say to these people, why do you need double precision? It turns out with decimal notation you can only go up to like a billion or something in single precision. So they have more dollars than that, so they need double, for that reason alone. But this gets back to the sloppiness of programmers. And I'm guilty of this myself. They said, oh we have double. Let's use double. They didn't need to, but they did it anyway. And now their legacy code is stuck with double. They could convert it all to single, but it's too painful. Down on Wall Street to build a new data center is like $100 million proposition. And they do it regularly, all of the banks. They'll be generating a new data center every year, sometimes multiple times

a year and they just don't have time or the resources to go through and redo all their code to make it run or something like cell. So we're making double precision cell. That's the short of it.

All right, now software environment. This is stuff that you can find on the web and actually, it's changing a lot lately because we just released the 2.0 SDK. And so the stuff that's in the slide might not actually be the latest and greatest, but it's going to be epsilon away, so don't worry about it too much. But you really shouldn't trust these slides, you should go to the website and the website you want to go to is www.ibm.com/alphaworks.

PROFESSOR: Tomorrow we are going to have a recitation session talking about the environment that we have created. I think we just got, probably just set up the latest environment and then we increase it through the three weeks we've got. This is changing faster than a three week cycle. So [UNINTELLIGIBLE PHRASE] So this will give you a preview of what's going to be.

MICHAEL PERRONE: Then you go to alphaworks, you go to search on alphaworks for cell and you get more information then you could ever possibly read. We have a programmer's manual that's 900 pages long, it's really good reading. Actually there's one thing in that 800, 900 hundred pages that you really should read. It's called the cell programming tips chapter. It's a really nice chapter. But there are many, many publications and things like that, more than just the SDK in the OS and whatnot, so I encourage you to look at that.

All right, so this is kind of the pyramid, the cell software pyramid. We've got the standards under here, the application binary interface, language extensions. And over here we have development tools and we'll talk about each of these pieces briefly. These specifications define what's actually the reference implementation for the cell. C++ and C, they have language extensions in the similar way to the extensions for VMX for SSE on Intel. You have C extensions for cell that allow you to use intrinsics that actually run as SIMD instructions on cell.

For example, you can say SPU underscore mul-add, and it's going to do a vector mul-add. So you can get assembly language level control over your code without having to use any assembly language. And then there's that. There is a full system simulator. The simulator is very, very accurate for things that do not run out to main memory. They've been working to improve this so I don't know if recently they have made it more accurate, but if you're doing compute intensive stuff, if you're compute bound the simulator can give you accuracies within 99%. You know, within 1% of the real value. I've only seen one thing on the simulator more than 1% off and that was 4%, so the simulator is very-- excuse me-- very reliable. And I encourage you to use it if you can't get access to hardware. What else?

The simulator has all kinds of tools in there. And I'm not going to go through the software stack in simulation. This gives you a sense for-- you've got your hardware running here. You can run this on any one of these platforms. Power PC, Intel with these OS's. The whole thing is written in TCL, the simulator. And it has all these kind of

simulators. It's simulating the DMAs, it's simulating the caches and then you get a graphical user interface and a command line interface to that simulator. THe graphical user interface is convenient, but the command line gives you much more control. You can treat parameters. This gives you a view of what the graphical userface looks like. It says mambo zebra because that was a different project, but now it'd probably say system sim or something like that. And you'll see the PPC-- this is the PPE I don't know why they changed it. And then you have SP of zero, SP of 1 going down and it gives you some access to these parameters. The model here, it says pipeline and then there's I think, functional mode or pipeline mode. Pipeline mode is where it's really simulating everything and it's much slower. But it's accurate. And then the other is functional mode just to test the code actually works as it's supposed to.

PROFESSOR: I guess one point in the class what we'll try and do is since each group has access to the the hardware, you can do most of the things in the real hardware and use the debugger in the hardware that's probably been talked about. But if things gets really bad and you can't understand use simulator as a very accurate debugger only when it's needs needed because there you can look at every little detail inside. This is kind of a thing, a last resort type thing.

MICHAEL PERRONE: Yeah, I agree. Like I said, I've been doing this for three years. Three years ago we didn't even have hardware. So the simulator was all we had, so we relied on it a lot. But I think that usage of it makes a lot of sense. This is the graphical interface. You know, it's just a Tickle interface. I'm going to skip through these things. It just shows you how you can look at memory with this more memory access. You get some graphical representation of various pieces. You know, how many stalls? How many loads? How many DMA transactions? So you can see what's going on at that level. And all of this can be pulled together into this UART window here.

OK, so the Linux, it's pretty standard Linux, but it has some extensions. Let's see. Provided as a patch, yeah. That might be wrong. I don't know where we are currently. You have this SPE thread API for creating threads from the PPEs. Let's see. What do I want to tell you here? There's a better slide for this kind of information. They share the memory space, we talked about that. There's error event and signal handling. So there are multiple ways you communicate.

You can communicate with the interrupts and the event and signaling that way or you can use these mailboxes. So each SPE has its own mailbox and inbox and an outbox so you can post something to your outbox and then the PPE will read it when it's ready. Or you can read from your inbox waiting on the PPE to write something. You have to be careful because you can stall there. If the PPE hasn't written you will stall waiting for something to fill up. So you can do a check. There are ways to get around that, but these are kind of common gotchas that you have to watch out for. Then you have the mailboxes, you have the interrupts, you also have DMAs. You can do communication with DMAs so you have at least three different ways that you communicate between the SPEs on

cell. And which one is going to be best really depends on the algorithm you're running.

So these are the extensions to Linux. This is going to show you a bunch of things that you probably won't be able to read, but there's something called SPUFS, the file system that has a bunch of open, read, write, and close functionality. And then we also have this signaling and the mailboxes that I mentioned to you previously. And this you can't even read. I can't even read this one. What is it? Ah, this is perhaps the most important one. It says SPU create thread. So the SPEs from the Linux point of view are just threads that are running. The Linux doesn't really know that they're special purpose hardware, it just knows it's a thread and you can do things like spawn a thread, kill a thread, wait on a thread-- all the usual things that you can do with threads. So it's a lot like P threads, but it's not actually P threads. So here you could see these things are more useful. This is SPE create groups. So you can create a thread and thread group so that threads that are part of the same group know about one another. So you can partition your system and have three SPEs doing one thing and five doing another. So that you can split it up however you like. You have get and set affinity so that you can choose which SPEs are running which tasks, so that you can get more efficient use of that element interconnect bus. Kill and waits, open, close, writing signals, the usual. Let me check my time here.

I really don't have a lot more time, so I'm going to say that we have this thread management library. It has the functionality that I just mentioned. In the next month or so you're going to go through that in a lot more detail. The SPE comes with a lot of sample libraries. These are not necessarily the very best implementation of these libraries and they're not even fully functional libraries, but they're suggestive of first of all, how things can be written to cell, how to use cell, and in some cases how to optimize cell. Like the basic matrix operations, there's some optimization. The FFTs are very tightly optimized, so if you want to take a look at that and understand how to do that type of memory manipulation. So there are samples codes out there that can be very useful. We'll skip that.

Oh, this is that FFT 16 million. There's an example, it's on the SDK. Actually, I don't know if you've got PS3's if all these things can run. They should run. Yeah, they should run. There may be some memory issues out to main memory that I'm not aware of. There are all kinds of demos there that you can play with, which are good for learning how to spawn threads and things like that. You have your basic GNU binutils tools. There's GCC out there. There's also XLC. You can download XLC. In some cases, one will be better than the other, but I think in most cases XLC's a little better. Or in some cases, actually a lot better. So you can get that. I'd recommend that. There's a debugger which provides application source level debugging. PPE multithreading, SPE multithreading, the interaction between these guys. There are three modes for the debugger: stand alone and then attached to SPE threads. Sounds like two. That's problematic.

There's this nice static analysis tool. This is good for looking for really tightly, optimizing your code. You have to be able to read assembly, but it shows you graphically-- kind of-- where the stalls are happening and you can try and

reorganize your code. And then like Saman suggested, the dynamic analysis using the simulator is a good way to really get cycle by cycle stepping through the code. And someone was very excited when they made this chart because they put these big explosions here. You've got some compiler here that's going to be generating two pieces of code, the PPE binary and the SPE binary. When you go through the cell tutorials for training on how to program cell you'll see that this code is actually plugged into-- linked into the PPE code. And when the PPE code spawns a thread it's going to take a pointer to this code and basically DMA that code into the SPE and tell the SPE to start running. Once it's done that, that thread is independent. The PPE could kill it, but it could just let it run to its natural termination or this thing could terminate itself or it could be interrupted by some other communication. But that's the basic process, you have these two pieces of code.

OK, so now this is really what I wanted to get to. So I want lots of questions here. There are 4 levels of parallelism in cell. On the cell blade, the IBM blade you have two cell processors per blade. So that's one level of parallelism. At chip level we know there are 9 cores and they're all running independently. That's another level of parallelism. On the instruction level each of the SPEs has two instruction pipelines, so it's an odd and an even pipeline. One pipeline is doing things-- the odd pipeline is doing loads and stores, DMA transactions, interrupts, branches and it's doing something called shuffle byte or the shuffle operation. So shuffle operation's a very, very useful operation that allows you to take two registers as data, a third register as a pattern register, and the fourth register as output. It then, from this pattern, will choose arbitrarily the bytes that are in these two and reconstitute them into this fourth register. It's wonderful for doing manipulations and shuffling things around. Like shuffling a deck of cards, you could take all of these and ignore this or you could take the first one here, replicate it 16 times or you could take a random sampling from these, put into that register.

AUDIENCE: Do you use that specifically for the [UNINTELLIGIBLE]?

MICHAEL PERRONE: We do use it, yeah. Yeah, you take a look, you'll see we use shuffle a lot. It's surprising how valuable shuffle can be. However, then you have to worry now, you've got the shuffle here, if you're doing like matrix transpose, it's all shuffles. But what's a date matrix transpose? It's really bandwidth bound, right? Because you're pulling data in, shuffling it around and sending it out. Well, where's the reads and writes? They're on the odd pipeline. Where are the shuffles? They're on the odd pipeline. So now you can have a situation where it's all shuffle, shuffle, shuffle, shuffle and then the instruction pre-fetch buffer gets starved and so it stalls for 15, 17 cycles while I have to load. Basically, it's a tiny little loop. But you get stalls and you get really bad performance. So then you have to tell the compiler-- actually, the compiler is getting better at these things. Much better than it used to be or by hand you can force it to leave a slot for the pre-fetch. These are gotchas that programmers have to be aware of.

On the other pipeline you have all your normal operations. So you have your mul-adds, your bit operations, all the

shift and things like that, they're all over there. There is one other operation on the odd pipeline and I think it's a quad word rotate or something, but I don't remember. So that's instruction level dual issue parallelism.

AUDIENCE: [UNINTELLIGIBLE PHRASE]

MICHAEL PERRONE: Everything is in order on this processor, yeah. And that was done for power reasons, right? Get rid of all the space and all the transistors that are doing all this fancy, out of order processing to save power.

AUDIENCE: [UNINTELLIGIBLE PHRASE]

MICHAEL PERRONE: That's a really good point. When you're doing scalar processing you think well, you're thinking I'm going to-- kind of conceptually, you want to have all the things that are doing the same thing together right. That's how I used to program. You put all this stuff here then you do maybe all your reads or whatever and then you do all your computes and you can't do it that way. You have to really think about how are you going to interlead these things. Now the compiler will help you, but to get really high performance you have to have better tools and we don't have those tools yet. And so I'm hoping that you guys are the ones that are going to come up with the new tools, the new ideas that are going to really help people improve programmability in cell.

Then at the lowest level you have the register level parallelism where you can have four single precision float ops going simultaneously. So when you're programming cell you have to keep all of these levels of hierarchy in your head. It's not straight scalar programming anymore. And if you think of it that way you're just not going to get the performance that you're looking for period. Another consideration is this local store. Each little store is 256 kilobytes. That's not a lot of space. You have to think about how are you going to bring the data in so that the chunks are big enough, but not too big because if they're too big thing then you won't be able to get multibuffering. Let's back up a little bit more. The local store holds the data, but it also holds the code that you're running. So if you have 200 kilobytes of code then you only have 56 kilobytes of data space. And if you want to have double buffering that means you only have 25 kilobytes and then as Saman correctly points out there's a problem with stack. So if you're going to have recursion in your code or something nasty like that, you're going to start pushing stack variables off the register file. So where do they go? They go in the local store. What prevents the stack them overwriting your data? Nothing. Nothing at all and that's a big gotcha. I've seen over the past three years maybe 30 separate algorithms implemented on cell and I know of only one that was definitely doing that. But you know, if there are 30 in this class maybe you're going to be the one that that happens to. So you have to be aware of that and you have to deal with it. So what you can do, is in the local store put some dead beef thing in there so that you can look for an overwrite and that will let you know that either you have to make you code smalller or your data smaller or get rid of recursion. On SPEs, recursion is kind of anathema. Inlining is good. Inlining really can accelerate your codes performance. Oh yeah, it says stack right there. You're reading ahead on

me here. Yes, so all three are in there and you have to be aware of that. Now there is a memory management library, very lightweight library on the SPE and it's going to prevent your data from overwriting your code because once the code's loaded that memory management library knows where it is and it will stop. The date you from allocating, doing a [? mul-add. ?] over this code. But the stack's up for grabs. And that was again done because of power considerations and real estate on the chip. It you want to have a chip that's this big you can have anything you want, but manufacturing it's impossible. So things were removed and that was one of the things that's removed and that's one of the things you have to watch out for.

And communication, we've talked about this quite a bit. I didn't mention this: the DMA transactions-- oh, question in the back?

AUDIENCE: Is there any reasonable possibility of doing things dynamically? Is it at all conceivable to have [? bunks ?] that fetch in new code or an allocator that shuffles somehow? Or is it basically as soon as you get to that point your performance is going to go to hell.

MICHAEL PERRONE: Yes, well if you don't do anything about it, yes your performance will go to hell. So there are two ways. In research we came up with an overlay mechanism. So this is what people used to do 20 years ago when processors were simple. Well, these processors are simple, so going back to the old technologies is actually a good thing to do. So we had a video processing algorithm where we took video images, we had to decode them with one SPE, we had to do some background subtraction to the next SPE. We had to do some edge detection. And so each SPE was doing a different thing, but even then the code was very big, the chunks of code were large. And we were spending 27% of the time swapping code out and bringing in new code. Bad, very bad. Oh, and I should tell you, spawning SPE threads is very painful. 500,000 cycles, a million cycles-- I don't know. It varies depending on how the SPE feels that particular day. And it's something to avoid. You really want to spawn a thread and keep it running for a long time. So context switching is painful on cell. Using an overlay we got that 27% overhead down to 1%. So yes, you can do that. That tool is not in the SDK. It's on my to-do list to put it in the SDK, but the compiler team at IBM tells me that the XLC compiler now does overlays. But it only does overlays at the function level, so if the function still doesn't fit in the SPE you're dead in the water. And I think the compiler will say, when it compiles it it'll say this doesn't fit quietly and you'll never see that until you run and it doesn't load and you don't know what's going on. So read your compiler outputs.

The DMA granularity is 128 bytes. This is the same, the data transactions for Intel, for AMD they're all 128 byte data envelopes. So if you're doing a memory access that's 4 bytes you're still using 128 bytes of bandwidth. So this comes back to this notion of getting a shopping list. You really want to think ahead what you want to get, bring it in, then use it so that you don't waste bandwidth; if you're bandwidth bound. If you're not than you can be a little more wasteful. But there's a guy, Mike Acton-- you can find his website, I think he has a website called

www.cellperformance.org? Net? Com? I don't know.

AUDIENCE: Just a quick comment [UNINTELLIGIBLE PHRASE].

MICHAEL PERRONE: Oh, he's good. He's much better than me. You're really going to like him. His belief, and I believe him wholeheartedly, is it's all about the data. We're coming to a point in computer science where the code doesn't matter as much as getting the data where you need it. This is because of the latency out to main memory. Memory's getting so far away that having all these cycles is not that useful anymore if you can't get the data. So he always pushes this point, you have to get the data. You have to think about the data, good code starts with the data, good code ends with the data, good data structure start with the data. You have to think data, data, data. And I can't emphasize that enough because it's really very, very true for this processor and I believe, for all the multicore processors you're going to be seeing.

The DMAs that you issue can be 128 bytes or multiples of 128 bytes, up to 16 kilobytes per single DMA. There's also something called a DMA list, which is a list of DMAs in local store and you tell the DMA queue OK, here are these 100 DMAs, spawn them off. That only takes one slot in the DMA queue so it's an efficient way of loading the queue without overloading the queue. Traffic controls, this is perhaps one of the trickier things with cell because the simulator doesn't help very much and the tools don't help very much. Thinking about synchronization, DMA latency handling-- all those things are important.

OK, so this is the last slide that I'm going to do and then I have to run off. I want to give you a sense for the process by which people-- my group in particular went through, especially when we didn't even have hardware and we didn't have compilers that worked nearly as well as they do now and it's really very ugly knifes and stones and sticks. You know, just kind of stone knifes. That's what I'm thinking, very primitive. But this way of thinking is still very much true. You have to think about your code this way. You want to start, you have your application, whatever it happens to be; you want to do an algorithmic complexity study. Is this order n squared, is this log n? Where are the bottlenecks? What do I expect to be bottlenecks? Then I want to do data layout/locality. Now this is the data, data, data approach of Mike Acton. You want to think about the data. Where is it? How can you structure your data so that it's going to be efficiently positioned for when you need it? And then you start with an experimental petitioning of the algorithm.

You want to break it up between the pieces that you believe are scalar and remain scalar, leave those on the SPE and the ones that can be paralellized. Those are the ones that are going to go on the SPE. You have the think conceptually about partitioning that out. And then run it on the PPE anyway. You want to have a baseline there. Then you have this PPE scalar code and PPE control code. This PPE scalar code you want to then push down to the SPEs. So now you're going to add stuff for communication, synchronization, and latency handling. So you

have the spawn threads. The [? RAIDs ?] have to be told where the data is, they have to get their code, they have to run their code, they have to then start pulling in the data, synchronize with the other SPEs and then latency handling with multibuffering of the data so that you can be doing computing and reading data simultaneously. Then you have your first parallel code that's running. Now the compiler, the XLC compiler, GCC compiler-- well, the XLC compiler I know for certain will do some automatic SIMDization. if you put the auto SIMD flag on. Does GCC compiler do that?

PROFESSOR: [UNINTELLIGIBLE PHRASE]

MICHAEL PERRONE: OK, so I don't know if the GCC compiler does that. So that can be done by hand, but sometimes that works, sometimes it doesn't. And it really depends on how complex the algorithm. If it's a very regular code, like a matrix-matrix multiply. You'll see that the compiler can do fairly well if the block sizes are right and all. But if you have something that's more irregular then you may find that doing it by hand is really required. And so this step here could be done with the compiler initially to see if you're getting the performance that you think you should be getting from that algorithmic complexity study. You should see that type of scaling. You can look at the CPI and see how many cycles per instruction you're getting. Each SPE should be getting 0.5. You should be able to get two instructions per cycle. Very few codes actually get exactly-- you can get down to 5.8 or something like that, but I think if you can get to 1 you're doing well. If you get to 2 there's probably more you can be doing and if you're above 2 there's something wrong with your code. It may be the algorithm. It may be just a poorly chosen algorithm. But that's where you can talk to me. I want to make myself available to everyone in the class or in my department as well. We're very enthusiastic about working with research groups in universities to develop new tools, new methods and if you can help me, I can help you. I think it works very well.

Then once you've done this, you may find that what you originally thought for the complexity or the layout wasn't quite accurate, so you need to then go do some additional rebalancing. Maybe change your block sizes. You know, maybe you had 64 by 64 blocks, now you need 32 by 64 or 48 by whatever-- some readjustment to match what you have, And then you may want to reevaluate the data movement. And then you know, in many cases you'll be done, but you're looking at your cycles per instruction or your speed up and you're not seeing exactly what you expected, so you can start looking at other optimization considerations. Like using the vector unit, the VMX unit on the cell processor, on the PPE. Looking for system bottlenecks and this is actually, I have found the biggest problem. Trying to identify where the DMA bottlenecks are happening is kind of devilishly hard. We don't have good tools for that, so you really have to think hard and come up with interesting kind of experiments for your code to track down these bottlenecks.

And then load balancing. If you look at these SPEs, I told you they're completely independent. You can have them all running the same code or they could be running all different code. They could be daisy chained so that this one

feeds, this one feeds that one, feeds that one. If you do that daisy chaining you may find out there's a bottleneck. That this SPE takes three times as long as any of the others. So make that use 3 SPEs and have this SPE feed these 3. So you have to do some load balancing and thinking about how many SPEs really need to be dedicated to each of the tasks. Now that's the end of my talk. I think that gives you a good sense of where we have been, where we are now, and where we're going. And I hope that if was good, educational, and I'll make myself available to you guys in the future. And if you have questions--

PROFESSOR: Thank you. I know you have to catch a flight. How much time do have for questions?

MICHAEL PERRONE: Not much. I leave at 1:10. So I should be there by 12:00.

PROFESSOR: OK. So [UNINTELLIGIBLE] at some time.

MICHAEL PERRONE: My car is out--

PROFESSOR: OK, so we'll have about 5 minutes questions. OK, so I know this talk is early. We haven't gotten a lot of basics so there might be a lot of things kind of going above your head, but we'll slowly get back to it. So questions?

AUDIENCE: You mentioned that SPEs would be able to run kernel. Is there a microkernel that you could install on them so that you could begin experimenting with MPI type structures?

MICHAEL PERRONE: Not that I'm aware of. We did look at something called MicroMPI, where we were using kind of a very watered down MPI implementation for the SPEs in the transactions. I don't recommend it. What I recommend is you have a cluster say, a thousand node cluster and the code today, the legacy code that's out there runs some process on this node. Take that process, don't try to push MPI further down, but just try to subpartition that process and let the PPE handle all the communication off board, off node. That's my recommendation.

AUDIENCE: So MPI is running on [UNINTELLIGIBLE]?

MICHAEL PERRONE: Yeah, Open MPI. It's an open source MPI. It's just a recompile and it hasn't been tuned or optimized. And it doesn't know anything about the SPEs. You know, you let the PPE do all the communication or handle the communications. When it finishes the task at hand then it can issue its MPI process.

AUDIENCE: [UNINTELLIGIBLE PHRASE]

MICHAEL PERRONE: Open NP is the methodology where you take existing scalar code and you insert compiler pragmas to say this for loop can be parallelized. And you know, this data structures are disjoint, so we don't have

to worry about any kind of interference, side effects of the data manipulation. The compiler, the XLC compiler implements open MP. There's several components that are required. One was a software cache where they implemented a little cache on the local store. And if it misses in that local cache it goes and gets it. I don't know how well that performs yet, but it exists. There's the SIMDization. For a while, Open NP wasn't working with auto SIMDization but now it does. So it's getting there, for so C it's there. I don't know what type of performance hit you take for that.

AUDIENCE: Probably runs [UNINTELLIGIBLE PHRASE]

MICHAEL PERRONE: It's

XLC version that does that. I don't know if GCC does it. But my recommendation is if you want to use open NP, go ahead, take your scalar code, implement it with those pragmas, see what type of improvement you get. Play around with it a little. If you find something that you expect should be 10x better and it's only 3x take that bottleneck and implement it by hand.

AUDIENCE: [UNINTELLIGIBLE PHRASE] with the memory models and such that the SPEs certainly went back a couple of generations to a simpler [INAUDIBLE]. How come you went so far back rather to just say, segmentation.

MICHAEL PERRONE: I don't know the answer. I'm sorry. I suspect and most of these answers come down to the same thing, it comes back to Sony. Sony contracted with IBM, gave us a lot of money to make this thing. And they said we need a Playstation 3. We need this, this, this, this. And so IBM was very focused on providing those things. Now that that is delivered, Playstation 3 is being sold we're looking at other options. And if that's something that you're interested in pursuing you should talk to me.

AUDIENCE: Among other things it seems to me that the lightweight mechanism for keeping the stack from stomping on other things --

PROFESSOR: I think that this is very new area. Before you put things in hardware, you need to have some kind of consensus, what's the right way to do it? This is a bare metal that gives you huge amount of opportunity but you give enough rope to hang yourself. And the key thing is you can get all this performance and what will happen perhaps, in the next few years is people come up to consensus saying, look, everybody has to do this. Everybody needs MPI, everybody needs this cache. And slowly, some of those features will do a little bit of a feature creep, so you're going to have they little bit of overhead, little bit less power efficient. But it will be much easier to program. But this is kind of the bare metal thing that to get and in some sense, it's a nice time because I think in 5 years if you look at cell you won't have this level of access. You'll have all this nice build on top up in doing this so, this is a unique positioning there. It's very hard to deal with, but also on the other hand you get to see underneath.

You get to see without any kind of these sort of things in there. So my feeling is in a few years you'll get all those things put back. When and if we figure out how to deal with things like segmentation on the multicore with very fine grain communication and there's a lot of issues here that you need to figure out. But right now all those issues are [INAUDIBLE]. It's like OK, we don't know how to do it. Well, you go figure it out OK?

MICHAEL PERRONE: Thank you very much.

PROFESSOR: Thank you. I don't have that much more material. So I have about 10, 15 minutes. Do you guys need a break or should we just go directly to the end? How many people say we want a break?