

Microsoft Research Faculty Summit
Chuck Thacker
Redmond, Washington
July 12, 2010

CHUCK THACKER: Well, thank you, Judith. It's a pleasure to be here. Seeing if we can key the slides.

Rethinking architecture, research and education. Why should we do that? Well, I've been worried about the state of architecture for a long time, and it didn't require us to hit the wall that we've now hit for me to be worried about how it was done, because I was not in it for 20 years. I did those networks Judith was talking about. But I'm back now, and so since I was offered this chance to talk to a number of faculty, I thought I would do it. However, I've found that in matters of architecture it's always good to emulate Dave Patterson, because after all, he wrote the book. And Patterson has a rule for projects and things like that: You pick a four-word acronym that starts in R, and both terms in that equation are important.

If you remember the RISK Enterprise, or, for instance, the RAID, or now more recently the RAMP effort, they were all in line with this rule. However, if you think about a few other projects, like NOW, Networks of Workstations, even though that project pretty much defined the way we do datacenters today, nobody remembers it. SPUR (?) and SOAR (?) were another two examples of why that doesn't work.

So, today you're actually going to get an industrial perspective on what we ought to be doing in order to teach architecture, and why should you be interested? You're, after all, academics. Well, I'll give you two answers. One is that we actually hire your product. So, you're producing product, and we are consuming it. We would like to be able to get the best that we can.

The second reason is that sometimes we found it possible to help you in ways that might seem a little odd, but can work very, very effectively, and I'll say some more about those things in a minute.

So, yeah, when Dave was president of the ACM, he wrote columns every month for the communications. This was while the communications was not quite as good as it is today. And two of the columns that influenced me quite a lot were these. The second column, that is, "computer science education in the 21st century," is the content of this talk, and I'll say a little bit more about that in a while, but the first column, "seven reasons to shave your head and three reasons not to, the bald truth," really impressed me a lot, because in that column Dave said that shaving your head makes you look younger and tougher.

And I thought, well, here I am, I could use both of those things. (Laughter.) But, of course, I'm an engineer, and I believe in the power of simulation in order to figure out whether something is going to work. (Laughter.) And so while Dave is almost right, in this case he was just solid, flat-out wrong. (Laughter.)

Okay, so the points that Dave raised in his second ACM column were these four. First was use tools and libraries. That seems, actually, pretty self-evident. I think we do that. But he said something else, which I believe appeared as a side bar, which struck me much more deeply, which is for many CS courses a dramatic change, would simply be if students first wrote a clear specification and then built software using modern tools and software components.

How many of you in your courses teach students to write specifications? One, two, three. I stand confirmed. (Laughter.) This is something we don't really do. And the reason we don't do it, I think, is because it's so easy to write programs these days that the program is the specification. But, of course, that leaves a lot to be desired. Nothing beats a spec, particularly because it unambiguously says what the program is supposed to do, and if it doesn't do it, then that's not a feature, it's a bug. If you only have feature lists for what the program is supposed to do, then the poor person who does the unit testing or the integration testing is left having to interpret the feature list, and so you wind up with a very unpleasant property that your testers are designing your programs for you, which is probably not what you want.

Right, the second point was embrace parallelism. And the reason he wants to do that, of course, is because we're going to have to. We are at an inflection point in computing these days, and we're now at a situation where we're not going to get faster computers. We used to get faster computers every year, and they were cheaper, and they were much more wonderful.

I have a table from another talk that describes 40 years of progress in some of these technologies, and it's really quite amazing. But we're going to have to have parallelism. We're going to have to figure how to use it. That's difficult for students. People do not like to think in parallel. They like to think sequentially. But we're going to need to do it. And maybe some of the things I'll talk about a little later will point at least one direction in how to do that.

The third thing was join the Open Source movement and actually contribute to Open Source projects. That may or may not have pedagogic value. I'm not sure, but Dave is almost always right, so maybe this is a good idea.

But the thing that he said as his fourth point was an interesting thing to me, because what he did was he described the RAMP consortium. And RAMP stands for Research Accelerator for Multiple Processors. And the idea which was put together in ISCA a few years ago—I think 2005—this was before I had anything to do with it—described something that's actually relatively new in the world. It's a consortium of six major universities who are in the architecture biz these days: Berkeley, Stanford, University of Texas, MIT, UW, and CMU, in no particular order. It's a little like herding cats I think.

But I was initially quite skeptical when Dave described the goals of the RAMP program to me as a way to use field programmable logic and Field Programmable Gate Arrays to actually build your own supercomputer. I had used FPGAs for many years, and I said, oh, it's not going to work, Dave, because the FPGAs are not big enough, and the design tools are not good enough to allow a small group of people to actually build a real computer.

I was wrong. I went back and looked, and I decided that Dave was absolutely right, and this would actually be a pretty good thing to do.

And the RAMP group had originally been working with an earlier machine called the BEE2, Berkeley Emulation Engine, Version 2, but they were looking for a more modern platform. The BEE2 was built with a somewhat older FPGA, and it was designed in the university, and so it was maybe not quite as reliable as they would like, and they wanted to build a replacement.

So, they came to MSR looking for support for some graduate students to do this job, and I looked around at the project, and I said, well, hmm, no, we won't do that, because if we give you this money, you'll just spend it on graduate students. And Dave said, well, that's what we do. And I said, well, but in this case, that's not what you need.

And so we made the RAMP consortium a bit of a deal, and we built this thing called the BEE3, which occupied BEE from about 2006 to 2009. The BEE3 is to a computer what AWK is to a compiler, or YACC perhaps. But it's four large FPGAs, lots of memory, lots of interconnect wiring, lots of IO so that you can plug many of them together into a larger system. And you can build anything on a BEE3 that you can describe in a Verilog program provided that it fits. And since the FPGAs are actually very large, most things do.

You can program the FPGAs from your desktop using low-cost commercial tools from Xilinx and download your program into it and run it, and it'll do whatever you want it to do.

It started about 1982. 1982 was probably the last year when a university could reasonably build a complete computer. The last chips that we saw of that ilk were things like the Berkeley RISC and the Stanford MIPS. There have been a few exceptions over the years. MIT has built a few large-scale systems, but they always had industrial partners.

So, until very recently, it's been impossible for a university to do this. But now with the advent of these large-scale FPGAs, it's possible again, and I think that this might represent a nice opportunity to do some more serious research in architecture than the kind of stuff that's been done over the last roughly 25 years.

We see mostly in architecture very incremental papers describing how you've taken somebody's SPARC architecture and twiddled it in the following seven ways, and run it through simulators and gotten plus 5 to minus 4 percent improvement on the following 17 benchmarks. They're all like that.

So, with the BEE3, you can do something quite different. Here's what it looks like inside. There are four large FPGAs. Each one has a PCI Express and two high-speed connectors that can be InfiniBand or 10 gig Ethernet or something like that, two channels of DDR memory, and lots of bandwidth there, and not much else.

It has been actually used. So, the program to put this whole thing together was done in a very odd way. MSR and Berkeley collaborated and did the detailed specification of the boards. We engaged a contract manufacturer, Celestica, for the implementation. And we feel that this is

considerably better than burning out graduate students to do this kind of work, because using graduate students to build something of this complexity and size is not good pedagogy. They learn to run CAD tools, but that's a sort of a fleeting skill. Next year's CAD tool will require more learning. And they avoid getting the kind of things they need to learn as graduate students.

So, we believe that it is much better to use pros for this purpose, and in our case, the pros were accessible to us, although they would not be normally accessible to a university because Celestica does not like to build—to design things with their Design Services Group if they're not going to have a large manufacturing follow-on, which this thing certainly is not.

So, it was fortunate that when I called them up I said, would you like to do this for us? You won't lose money on it, but you won't make a lot. But it's maybe a chance to do some good in the world, and I know it won't make you a lot of manufacturing follow-on, but what do you think?

And they said, well, yeah, sure. And that was a good thing to say, because they actually build about half of the XBox 360 consoles that Microsoft buys. (Laughter.) So that was an example where we could bring to play a resource that you just couldn't have ever possibly had.

So, we also were instrumental in starting up a company, a third-party company called BCube Incorporated, which actually builds, sells, and supports the systems, and they've shipped about 75 to date. And MSR supplied some of the basic IP, intellectual property, like a DRAM controller, but by and large BCube has done a fine job at supporting the users.

The fact that the design was done by us and, in fact, owned by us, although licensed to BCube at no charge, means that both academic and industrial customers can buy them.

So, in a sense, this was not a direct contribution, this was enlightened self interest, because we had just recently decided to set up architecture research groups in Microsoft, and we wanted these machines. But when I went to the BEE2 people and said what do they cost and who do I write the check to, they said, we don't know.

So, doing it this way means that you've got something quite a bit faster in terms of calendar time, quite a bit more reliable than if it had been the first design done by a graduate student, and it was quite amazing.

As I said in that earlier bullet, the resulting board worked the first time. This is the first time that's ever happened to me in building a piece of hardware. I have never had a computer system or anything, a network, whatever, that's worked the first time. And these guys just did a brilliant job. So, it was great.

There are some downsides. It's pretty expensive. Do any of your schools have BEE3s? I don't see any hands at all. They are expensive. They cost about \$20,000 if you're not an academic, or maybe 25, 15,000 if you are, because Xilinx will give you the chips. But that's still a fairly large amount of money for a school.

So, the other problem is that it is quite big, so it doesn't fit in an office, and it's also quite loud. You wouldn't want it in an office with you. I have one in my office, but the fans are actually turned off. If they weren't, it wouldn't be there.

The other problem is we got to looking around starting about a year ago, and I made the observation that although the RAMP consortium calls itself a group that wants to do research in multi-core computing, they didn't actually have a multi-core computer. What they had is cores that they could get out on preferably the Open Source market, including the Sun OpenSPARC, or something like cores they could get from Xilinx, or cores that they could get from someplace like, say, ARM. All of these people have such cores optimized for FPGAs, but they're not optimized to be done in quantity.

So, I said, well, hmm, we can probably fix this. I know how to design computers. So, we did a thing called Beehive, and Beehive has occupied several people in our laboratory in Palo Alto, or in Mountain View, for the last year or so. It's an FPGA-based mini core system, and it is actually a mini core. It has 13 RISC cores. They're not anybody's architecture particularly, very vanilla. They run at 100 megahertz each. The board on which we run it supplies only 2 gigabytes of DDR memory, and there's only one FGPA on that board. There's a display controller.

The BEE3 does not have a display controller. It has no place to plug in a display, unless you add it. It does have a gigabit Ethernet controller, which the Beehive does, too.

But the nice thing about the Beehive from the standpoint of using it in education is that it's about 6,000 lines of Verilog.

Now, people have built cores in Verilog. I have a student from Berkeley working with me this summer who has built a very nice one, a multi-threaded SPARC, which is 36,000 lines of system Verilog, which is actually quite a bit more expressive than Verilog.

So, this thing is much smaller. Students can actually understand it and modify it using the only the basic Xilinx tools. We did not want to use the expensive CAD tools that the major universities can get their hands on, but which the smaller ones might not be able to. So, we kept it simple.

We've built a software tool chain for it, a C compiler, assemblers, linkers, things like that, and a small but steadily growing set of libraries.

We're actually just beginning to bring up an MSIL compiler, which will allow us to run any .NET language on this machine.

And, of course, we license it for academic use.

This is the board. This is a Xilinx XUPV5 board, which is available for an academic price of \$750 from a company called Digilent. It has a somewhat smaller FPGA than the BEE3 does, and it has only one of them, but actually has the FPGA that was used in the prototype BEE3, so it's pretty big.

Xilinx wants you to use it for running the OpenSPARC design. You can get one of those on this board. We can get 13, but it's not a SPARC, it's something else. So, they use it, one of them, use it for one OpenSPARC core. We use it for Beehive.

The board has actually quite a bit more stuff on it than we use, so there's a lot of things out there on the board that could be used for student projects of one sort or another. We have not done that yet because we've been actually doing some other things, which I'll talk a little more about.

We did have to put heat sink on the FPGA and a small fan, because it does get hot.

So, here's the Beehive core CPU. Now, this looks like something that you would see in architecture books about 30 years ago. It has dual port register file, 32 registers of 32 bits. It has an instruction cache and a data cache, although the data cache is not shown in this picture. Very simple instructions, and there's only slightly wacky thing about this architecture. If you note, there is an address queue and a write queue and a read queue. And the idea is that when the CPU wants to issue a memory operation, data memory operation, it writes an address into the address queue, and then if it's a write, it writes data into the write queue, and it can do those things in either order. And as soon as the queue is non-empty, it begins to do the operation. And if it's a read, then a little bit later the data is deposited in the read queue and the process can access it.

Now, this is called a de-coupled architecture, and it was under investigation in about the early '80s. I heard about this idea from Bill Wulf, who designed a machine actually called the WM, whose purpose in life was to do a good job at running ADA programs, because Bill was at that time at Tartan Labs, which had a strong economic interest in doing exactly that.

The nice thing about this is that if you have a compiler than can hoist the issuing of the address well in advance of the need for the data, then you can overcome an awful lot of the memory's latency, and memory latency is, of course, getting to be the problem in designing computers these days, or one of the problems. The other one is heat.

So, when Bill told me about this, I said, well, I'll put it in. The pipeline is a three-stage pipe. It's very, very simple: instruction fetch and read the registers, execute the instruction, and write the results back; the sort of thing you saw in the first version of Hennessy and Patterson. There is a local IO system. You see here those queues connecting to a number of external sometimes co-processors and sometimes caches. There's a multiplier, which uses a digital signal processor that is very efficient in the Xilinx architecture. So, it can do a 32 by 32 signed integer multiply in 10 cycles. That's fast enough that you can actually do integer division by Newton-Raphson's method, Newton-Raphson approximation, and do about twice as fast as you can do the one bit at a time sort of thing that you would otherwise do. There's an RS-232 interface, which is switched between the cores. There's the data cache. Instruction cache is actually also in there, too, but it was shown on the other slide.

There are two interesting things. There is a message unit which allows you to send intercore messages between the Beehive cores without going through memory at all. And there's also a lock unit. The lock unit supports 64 global binary semaphores, also without using memory at all.

Here's the instruction format, an A and B field for read addresses for the register file, a write address. There's a count field because the thing has a very comprehensive barrel shifter that can do all kinds of shifts and operations. And if you recall, the shifter and the adder are in series. There's a function field and an operation field.

So, normal instruction is take something from the register address by RA, do a function on it against the contents of RB, and then shift it by some amount.

There are variants for jumps and memory accesses, and there's a lot of support for constants, because constants seem to be important these days.

The other thing that's interesting about this architecture is if you're familiar with FPGAs, you know that you cannot build something that has 13 CPUs, because you can't wire it up to the memory. And the reason for that is that in an FPGA the critical resource is long wires. And so what we did was instead of using a lot of long wires, we made all that wiring local by making the cores talk to one another through this ring interconnect, and that means that we can actually build it. The ring interconnect passes through each core, the display controller, the Ethernet controller, and the DRAM controller. And the way to think about it is a train. So, at the beginning of time, or when it can, the memory controller issues a token slot on this ring, and as the ring goes around, if a particular core, when it sees the token, if it wants to send something on the train, it adds the number of slots that it is going to add to the train to the thing that's going around with the token. And then when the original number of slots has gone by, it does its addition. And so the ring contents can grow as it passes between the cores, and then when it returns to the memory controller, those things are actually put into some queues that actually access the DDR memory.

Everything happens in order in this system. There's no out of order anything. And that means that a lot of these things can be very simple queues rather than more complex structures.

There's a separate bus for delivering read data back to the cores, because that allows us to double the bandwidth, and it was easy, and we could wire it. There's also a separate path to the display controller, because the display controller can actually consume more bandwidth than the ring can provide, so we put in a back door path to make that happen with no interference to the ring. That seems to have worked out fairly well.

So, the curiosities for this design are these. There is no coherent memory. There is no byte addressing, although we fudge this a bit. There is no protection at all. We may add this, but we may not. If you're programming in a language like the MSIL languages, protection is kind of overrated. There's no virtual memory. There's no kernel mode.

So, how do you build a system with a thing like this? Oh, and there's one thing I didn't mention. There are no interrupts. Interrupts are actually considered a bad idea these days. I was surprised at ISCA to discover that a lot of people are beginning to say this, and I said it as well. If you have a large number of cores, it seems very prudent to just give threads to cores, and then they can just poll. And if there's no work to do, then the core just stops, goes to sleep. Now, 30 years ago when there was one CPU and there was a high motivation to keep it busy all the time, interrupts

were necessary, because you needed to be able to time slice. But today that's much, much less clear, so we left them out. And we haven't actually missed them.

So, what are we going to use it for? Well, there are two things. One is to use it for education, and the idea here is architecture lab courses. The boards are sufficiently inexpensive that every student can have one. And, in fact, if you ask them nicely, Xilinx might even give them to you. The Verilog is simple enough for students to make changes and try out new things. And this is very much like—how many of you use coursework that is part of the NetFPGA project at Stanford in your departments? No? Well, that's a little odd, because they have actually distributed over a thousand of these boards and many, many copies of their courseware to a large number of universities. I guess the—hmm, very strange.

PARTICIPANT: (Off mike).

CHUCK THACKER: Sorry?

PARTICIPANT: (Off mike).

CHUCK THACKER: I did actually.

PARTICIPANT: They didn't.

CHUCK THACKER: Oh, they didn't. Well, maybe you should get in touch with them. It's Nick McKeown.

And they've basically revolutionized the way a lot of people teach networking. And the kids basically get a starting board and a simple Ethernet switch, and they wind up building a router. The tool chains and libraries are very familiar, GCC and MAKE. Everybody knows how to use that. And initial results in using this thing in that kind of a context are pretty promising. At MIT Franz Kaashoek and others taught a two-week IAP course in January, and they will be teaching a full semester junior-level course based on the Beehive in fall of this year. Xilinx donated 20 boards for this thing. Here are some of them lined up in their machine room with their fan. (Laughter.)

And amazingly enough, I was a little bit surprised, because the course duration was quite small, just two weeks. Actually, it was a week with a weekend separating it out in the middle, so they actually got a few more days.

But students were actually able to modify the Verilog and successfully test their changes. And, in general, they were improvements, because I cut corners in a lot of places, so there are a lot of things that can be improved.

Another use for the BEE3 is research, as a research platform. Now, that's what we're using it for now. It's mostly done. We have something that we can distribute to people. And so we're beginning to use it.

A couple of ideas. One is forget shared memory, okay? Shared memory has been with us for a long time, shared coherent memory, but it's increasingly difficult as you scale up the number of cores to make something that works. And message passing and shared memory are actually duals of one another. They'll do the same thing. But 25, 30 years ago we went down the shared memory path rather than the message passing path, and I think that probably happened because we didn't understand how hard it would be to build a shared memory system at large scale. And if we'd thought about it a little more carefully, we might have really looked much more carefully at message passing.

We're actually using it to try out transactional memory and figure out whether it's a good idea. My friend, Richard—I beg your pardon—at Cambridge told me that there have been 500 papers written on transactional memory, since he just finished the new bibliography and described all of them.

It's all been software systems, software transactional memory. The problem with software transactional memory is it's not fast enough. So, it will never be commercially useful, because nobody will tolerate a slowdown of a factor of two or three just to write a correct program, correct concurrent program. So, we need to use hardware to do it, and if we could get a hardware system that would do it, we could allow things like apples-to-apples comparisons with monitors and condition variables, which are the other way to do concurrency.

The nice thing about the system that we've built, which we stole many ideas from the Stanford Transactional Consistency and Coherency System, except that ours is not quite so bounded as theirs was, it gets you memory coherence where you need it, which is your shared mutable data. It doesn't pay the overhead if you don't need it.

So, one of the things that we're beginning to think about is for the computers of the 21st century looking at some of these decisions that we've made and asking, were they really a good idea, things like shared memory, things like interrupts, things like virtual memory, even an operating system. Maybe you don't need operating systems for a lot of things.

So, non-goals for the Beehive: Well, we did not want to emulate an existing instruction set architecture, because existing modern instruction set architectures are not simple. They are very complicated. They are very difficult to understand and probably for students and so on impossible to modify, except very good students.

So, the problem with that and using the Beehive for education is that you can't do direct comparisons with stuff you have today. You can only do A/B comparisons of the form; if I have this feature versus not having it, how does my program's behavior change? Those you can do, and you can do them very effectively because you can instrument the thing six ways from Sunday, because you can build the instrumentation into your hardware. It's surprising the amount of payback that idea has given us just in debugging the basic system.

We didn't want to run an OS initially. We discovered that, in fact, our friends in the Cambridge lab who had been building an operating system called Barrelfish in conjunction with ETH in Zurich have actually looked at the Beehive and decided that it might be actually a pretty

reasonable machine, because their system is heavily based on message passing and doesn't use coherent memory. So, they're porting.

But the other non-goal is high performance. And the reason for that is you can't get that with an FPGA anyway, unless you're doing something very special that can leverage the built-in things like the digital signal processors that the architecture provides. You're just not going to get it. So, we figured that you only needed to build things that were fast enough to run real programs for substantial periods of time so that we could understand them. That means maybe 1,000 times faster than a simulator, something on that order. We could do that. Actually, a 13-core, 100 megahertz machine is not all that shabby. That's a machine of about 10 to 12 years ago if it's a PC.

So, what are the next steps? Well, the first thing to do is to port it back to the BEE3, because it stopped running on the BEE3 quite a while ago because we got much more interested in making it widely available, and so we ported it to this Xilinx development board. We want to put it back so that we can build bigger systems, use it in our own research. I mentioned the transactional memory system that we have built, which now seems to work. We're starting to run transactional benchmarks that people always run, STAMP and IGN bench (ph) and things like that, and we'll try to see whether transactional memory is a good idea.

The Barrelfish operating system, I mentioned that's MSR Cambridge and ETH.

And finally, we want to make it more widely available for academic use. And if you think this might be interesting for your courses, send me some e-mail. Name is on the first slide.

So, that's really all I have to say today. It seems to me that we have both an opportunity and a need to do a lot of reassessing about how we do architecture, both in the wild, that is, build computers, and how we educate people to do that. And I think that these field programmable components can actually help quite a bit in that effort. As I say, I was initially skeptical. I'm now a convert. So, perhaps what you're hearing is what you always hear from converts, extreme conviction.

So, I thank you very much, and hope you enjoy the lake trip tonight. (Applause.)

MODERATOR: (Off mike). Questions?

CHUCK THACKER: Yeah, sure.

QUESTION: (Off mike).

CHUCK THACKER: It's GCC.

PARTICIPANT: It's not that simple. It's not (off mike)—

CHUCK THACKER: No, it is definitely not that simple, but it's probably the simplest thing we could get.

So, it depends on the complexity of the change. Most of the things that we wanted to do with it did not need to have the ISA changed. People make ISA changes in order to get differing levels of performance. We're interested in things that make qualitative differences in the way you build a computer, not quantitative. So, ISA changes were not all that interesting. We did want to run MSIL though.

QUESTION: (Off mike). I actually—oh, there we go—do multi-core researching using FTPAs. One of the questions I had for you was, how much of the chip are you using, because it greatly affects what else I can put on there? So, as someone who uses the boards you're talking about to do multi-core research, I'm curious with your Beehive how much of the device you're using.

CHUCK THACKER: Yeah, very good question. Right now at 13 cores it uses about 50 percent of the available logic. This means that it's actually quite easy for the automatic tools to place and route the design without spending hours and hours and hours, although working with FPGAs and programming them with Verilog is a little like programming a computer with punched cards and batch queues, if any of you can remember those days. I can. It's the same thing. The turnaround is about three quarters of an hour for a one-line change. So, you're pretty careful when you make those changes.

MODERATOR: Great. Question? Up there? Okay, number two.

QUESTION: I just wonder what's your opinion about the future microprocessor design, and because some people are optimistic, and they're saying that if we can—but based on Moore's Law, we are—we won't be doubling the number of transistors, we'll be doubling the number of cores from now on, and we'll soon have a thousand-core microprocessor. And some people are not that optimistic, and they're saying that we maybe have eight cores and 10 cores, and we maybe have some difficulty to keep up. So, what your opinion on that?

CHUCK THACKER: Well, the problem with having a large number of cores is getting the memory bandwidth to feed them. If you can use a very high core count system to do a specialized function where the data does not have to leave the chip, except maybe as it comes in and when the final result comes out, that seems plausible for a large, high count system. But if you have a system that accesses memory a lot, you're going to be stuck. And maybe the number is not eight, maybe it's 32 or something like that, but it's not hundreds. So, I think it will depend a lot on the type of problem and whether we can get our heads around how to decompose problems into parallel computations.

MODERATOR: Right. Well, on that topic, we can thank Chuck very, very much, and for a splendid, splendid—(applause)—and see you on the boat.

END