PROFESSOR RABBAH: OK, so today's the last lecture day we're going to talk about the raw architecture. This is a processor that was built here at MIT and essentially trailblazed a lot of the research in terms of parallel architectures for multicores, compilation for multicores, programming language and so on. So you've heard some things about RAW and the parallelizing technology in terms of StreamIt. So we're going to cover some of that again here today just briefly and give you little bit more insight into what went into the design of the raw architecture.

So these are RAW chips they were delivered in October of 2002. Each one of these has 16 processors on it. I'm going to show you sort of a diagram on the next slide. It's really a tiled microprocessor. We'll get into what that means and how it actually-- what does a tiled microprocessor give you that makes it an attractive design point in the architecture space?

Each of the raw tiles-- you can sort of see the outline here sort of replicates-- is four millimeters. It's four millimeters square. It's a single-issue 8-stage pipeline. It has local memory, so there's a 32K cache. And the unique aspect of the raw processor is that is has a lot of on-chip networks that you could use to orchestrate communication between processors.

So there's two operand networks. I'm going to get into what that means and what they used for. But these eventually allow you to do point-to-point communication between tiles with very low latency. And then there's a network that essentially allows you to handle cache misses and input and output and one for message passings, a more dynamic style of messaging, something similar to what you're accustomed to at the cell, for example, in DMA transfers.

This was built in 180 nanometer ASIC technology by IBM. It's got 100 million transistors. It was designed here by MIT grad students. It's got something like a million gates on it. Three to four years of development time. And what was really interesting here is that this was-- because of the tiled nature of the architecture, you could just design one tile and then once you have one tile, you essentially just plop down more and more and more of them. And so you have one, you scale it out to 16 tiles. And the design sort of came back without any bugs when the first chip was delivered.

The core frequency was expected to run at 425-- I think lower than 425 megahertz.

AUDIENCE: Designed for 250?

PROFESSOR RABBAH: 250 megahertz and came back and it ran 425 megahertz. And it's been clocked as high as 500 megahertz at 2.2 volts.

The chip isn't really designed for low power but the tile abstraction is really nice for power consumption because if you're not using tiles you can essentially just shut them down. So it'll allow you to sort of have power efficient design just by nature of the architecture. But when you're using all the tiles, all the memories, all the networks, in a non-optimized design, you consume about 18 watts of power.

So how do you use this tiled processor? So here's one particular example. The nice thing about tile architecture is that you can let applications consume as many tiles as they need. If you have an application with a lot parallelism then you give it a lot of tiles. If you have an application that doesn't need a lot of parallelism then you don't give it a lot of tiles. So it allows you to really exploit the mapping of your application down to the architecture and gives you ASIC-like behavior-- application specific processing technology.

So one example is you have some video that you're recording and you want to encode it and stream it across the web or display it on your monitor or whatever else. So you can have some logic that you map down. If your chips are here, you do some computation. You have memories sprinkled across the tile that you're going to use for local store. So you can parallelize, for example, the motion estimation for encoding the temporal redundancy in a video stream.

You can have another application completely independent running on other part of the chip. So here's an application that's using four different tiles and it's really isolated. It doesn't affect what's going on in these tiles. You can have another application that's running something like MPI where you're doing dynamic messaging, and httpd server and this tile is maybe not used so it's just sleeping or it's idle. You can have memories connected off the chip, I/O devices. So it's really interesting in the sense that probably the most interesting aspect of it is you just allow the tiles to sort of be used as your fundamental resource. And you can scale them up as your application parallelism scales.

This is a picture of the raw board-- the raw motherboard. Actually you see it in the Stata Center in the Raw Lab. This is the raw chip. A lot of the peripheral device, firmware and interconnect for dealing with a lot of devices off the chip are implemented in these FPGAs, so these are Xilinx chips. There's DRAM. You have connection to a PCI card, USB stick. Network interface so you can actually log into this machine and use it. And there's a real compiler. It can run real applications on it.

There's actually a bigger chip that we built where we take four of these raw chips and sort of scale them up. So rather than having 16 tiles on your motherboard, you can have four raw chips. That gives you 64 tiles. You can scale this up to a thousand tiles or so on. Just because of the tile nature, everything is symmetric, homogeneous, so you can really scale it up really big.

So what is the performance of raw? So looking at the overall application performance, so we've done a lot of benchmarking. So these are numbers from a paper that was published in 2004, where we took a lot of applications-- some are well-known and used in standard benchmark suites-- and compiled them for raw using various raw compiler that we built in-house. And we've compared them against the Pentium 3. So the Pentium 3 is sort of a unique comparison point because it sort of matches raw in terms of the technology that was used to fabricate the two.

And what you're seeing here, this is a lock scale. The speedup of the application running on raw compared to the application running on a P3. So the higher you get, the better the performance is. So these applications are sort of grouped into a few classes. So the first class is what we call ILP applications. So these are applications that have essentially instruction level parallelism. I'm going to talk a little bit more about and sort of explain it. But you've seen this early on in the lecture-- in some of Saman's lectures. So here you're trying to exploit inherent instruction level parallelism in the applications. And if you have lots of ILP then you map it to a lot of tiles and you can get parallelism that way and you get better performance.

These applications here-- what we call the streaming applications. So you saw some of these in the StreamIt lecture and the StreamIt parallelizer compiler. Some of those numbers were generated on a raw-like architecture. And then you have the server or sort of more traditional applications that you expect to run in a server style or throughput-oriented. And then finally you have bit-level applications. So doing things at the very lowest level of computation where you're doing a lot of bit manipulation.

So what's interesting here to note is that as you get into more applications that have a lot of inherent parallelism in them, where you want explicit-- where you can extract a lot of parallelism because of the explicit nature of the applications-- you can really map those really well through the architecture. And because of the communication nature-- because of communication capabilities of the architecture, being able to stream data from one tile to another really fast, you can get really high on-chip bandwidth and that gives you really high performance, especially for these kinds of applications.

There are other applications that we've done. Some of the students have worked on in the raw group. So an MPEG-2 encoder where you're essentially trying to do real-time encoding of a video screen at different resolutions. So 350 by 240 or 720 by 480 where you're compiling down to a number of tiles. One, 4, 8 sixteen, 16-- 1 and 16 are somehow missing, I'm not sure why. And what you're looking for here? Sort of scalability of algorithm. As you add more tiles, are you getting more and more performance or are you getting better and better throughput? So you could encode more frames per second for example. So if you're doing HDTV, it's 1080p, then you really want to sort of get-- there's a lot of compute power that you need. And so as you add more frames, maybe you can get to sort of the throughput that you need for HDTV.

So this is something that might be interesting for some of your projects as well. And we've talked about this before. On the cell, as you're using more and more SPEs, can you accelerate the performance of your application? Can you sort of show that if you're doing some visual aspect? And you can sort of demonstrate it. So there's a demo that is set up and in the lab where you can sort of crank up number of tiles that you're using and you get better performance from the MPEG encoder. And just looking at number of frames per second that you can get, with 64 tiles-- so the raw chip is 16 tiles, but you can scale it up by having more chips-- so you can get about 51 frames. These numbers have been improved and there are different ways of optimizing these performances. At 352 by 4 240, the estimated data rate-- estimated throughput-- of 160 frames per second almost. So this is really high bandwidth.

Another interesting thing that we've done with the raw chip is taking a look at graphics pipelines and looking at is there anything we can do to exploit the inherent tiled architecture of the raw chip. So here's a screenshot from *Counterstrike* and a simplified graphics pipeline where you have some input to the screen you want to render. You do some vertex shading. So these are triangles that you want to figure out what colors to make-- what colors to paint

them. The triangle's set up for pixel stage. And in this screen you'll notice that there are two different things that you're rendering. There's essentially this part of the screen which has a lot of triangles that span a relatively not-so-complex image.

And then you have these guys here that have fewer triangle span a smaller region of the frame. And what you might want to do is allocate more computer power to the pixel stage and less compute power to the vertex stage. So that's analogous to saying, I want more tiles for one stage of the pipeline and fewer tiles for another. Or maybe I want to be able to dynamically change how many tiles I'm allocating at different stages of the pipeline. So that as your screens that you're rendering change in terms of their complexity, you can maintain the good visual illusions transparently without compromising the utilization of the chip.

So some demos that were done with the graphics group it at MIT-- Fredo Durand's group-- phong shading. You have 132 vertices with 1 light source. So this is what you're trying to shade. You have a lot of regions black. So if you're looking at a fixed pipeline where the vertex shader is taking six tiles-- this is on a 64-tile chip-- the rasterizer is taking 15 tiles, the pixel processor has 15 tiles, the alpha buffer operations has 15 tiles, then you might not get the best utilization because for that entire region that you're rendering where it's black there's nothing really interesting happening there. You want to shift those tiles to another processor, to another stage of pipeline. Or, if you can't really utilize them, then you're just wasting power, wasting energy, and so you might just want to shut them and not use them at all. So with a fixed pipeline versus a reconfigurable pipeline where I can change the number of tiles allocated to different stages of the pipeline, I can get better utilization. And, in some cases, better performance. So here, fuller bars, and you're finishing faster in time.

So this is indicative also of what's going on in the graphics industry. So the graphics card used to be very-- well, it had fixed resources allocated to different stage, which is essentially what we're trying model in this part of the experiment, where more and more now you have unified shaders that you can use for the pixel shading and the vertex shading. So you're getting into more of that programmable aspect. Precisely because you want to be able to do this kind of load balancing and exploit dynamisms that you see in different things that you're trying to render.

Another example: shadow volumes. You have 4 triangles, one light source. And this was rendered in three passes. So pass 1, pass 2, pass 3, would essentially take the same amount of time because you're doing the same computation map to a fixed number of resources. But if

I can change the number of resources that I need for different passes-- so the rasterizer, for example, and the alpha buffer operations, is really where you need a lot of power. So if you go from 15 tiles for each to 20 tiles for each, you get better execution time because you were able to exploit parallelism or match parallelism better to the application. And so you get 40% percent faster in this particular case.

And another interesting application: this is the largest in the world microphone array. It's actually in the *Guinness Book of Records.* It was build in the lab. And what it essentially has-- each of these little boards has two microphones on it. And so what you can use this for is eavesdropping for example. Or you can carry this around if you want. Pack it in the car and do some spying. But somewhat more interesting demos that were done with this in smaller scales was that in a noisy room, for example, if you want the sort of hone in. Let's say everybody here was speaking, but for the camera they want to record only my voice. They can have a microphone array in the back that focuses on just my voice. And the way it's done is you can measure the distance from the time it takes for the sound wave to reach each of these different microphones and you can focus in on a particular source of noise and be able to just highlight that.

So there's this demo where's it's a noisy room-- I probably should have had these in here in retrospect-- there's a noisy room, lots of people are talking, then you turn on the microphone array and you can hear that one particular source and it's a lot clearer. You can also have applications where you're tracking a person in a room with videos as well, so you can sort of follow him around. So it's a very interesting application. An now I regret not having the video demo in here. Actually, should I do it? It's on the Web. OK.

So a case study using the beamformer. So what's being done in the microphone array is you're doing beamforming. So you're trying to figure out what are the different beams that are reaching the microphone. You want to be able to amplify one of them. So looking at the application written natively in C running on a 1 gigahertz Pentium , what is the operation throughput? So you're getting about 240 MegaFLOPS. And if you go down to an optimized-- same code but running on single tile raw chip, you get about 19 MegaFLOPS. So a not very good performance. But here, what you really want to do, is you have al lot of parallelism. Because each of those beams that's reaching individual microphones can be done in parallel. So you have a lot of parallelism in that application.

So taking the C program, reimplementing it in StreamIt that you've seen in previous lectures,

and not really optimizing it in terms of doing a lot of the optimizations you saw in the parallelizing compiler talk, you get about 640 MegaFLOPS. So already you're beating the C program running on a pretty fast superscalar machine. And if you really optimize the StreamIt code in terms of doing the fission and fusion, increasing the parallelism, doing better load balancing automatically, you can get up to 1.4 GigaFLOPS. So really good performance and really matching the inherent parallelism to the architecture.

So it was just a big overview of the raw chip and what we've done with it in lab. There's more in here than I've talked about. But what I'm going to do next is give you some insights as to what is the design philosophy that went into raw architecture, why was it designed the way it was. And then I'm going to talk a little bit about the raw parallelizing compiler. And while the StreamIt language and compiler also has a back end for the raw architecture, we've sort of seen that in previous lectures so I'm not going to talk about that here. So I'm just going to focus on the first two bullets.

And a few years ago when the project got started, sort of the insight in the wide issue processors and the design philosophy that was being followed in industry for building wider superscalars, faster superscalars, was really going to come to a halt largely because you have scalability issues. So if you look at sort of a simplified illustration of a wide issue microprocessor, you have your program counter such as instructions. Goes into some control logic. Control logic is then going to run. You're going to read some variables from the register file. You'll have a big crossbar in the middle that routes to operands like ALUs. Yell And then you operate on those and you have to send it back to the register file.

Plus you have this really big problem with the network. So if you're doing some computation-- sorry, I rearranged these slides. So what you have if you have n ALUs, then the complexity of your crossbar increases as n squared, because you essentially have to have everybody talking to each other. And in terms of the number of wires that you need out of the register file to support everybody being able to sort of talk to anybody else very efficiently, the number of ports, the number of wires increases n cubed. So that's a problem because you can't clock all those wires fast enough. The frequency becomes sort of limited. It grows even less than linearly. And this is a problem because operational routing-- operand routing, is global. So if I have- I'm doing some operations and it's an add, the results of this add is fed to another operation to shift, and these are going to execute on two different ALUs.

So what's going to happen? I do the add operation. It's going to produce a result. But there's

no direct path for this ALU to send this result to this ALU. So instead what has happened is the operand has to travel all the way back around through the crossbar and then back to this ALU. So that's really just going to take a long time and not necessarily very efficient. And if you're doing this for a lot of ALU operations, you have a lot of parallelism in your application level, instructional level parallelism, and that's just creating a lot of communication. But you're not really exploiting the locality of the computation. If 2 instructions are really close together, you want to be able to just have a point-to-point path, for example, or a shorter path that allows you to exploit where was instructions are in space.

And so this was the driving insight for the architecture in that you want to make operand routing local. So an idea is to essentially exploit this locality by distributing the ALUs. And rather than having that massive crossbar, what you want to do is have an on-chip mesh network. So rather than have one big crossbar, you have lots of smaller ones. So these become switch processors. So I can put value from this ALU here and then have that value routed to any other ALU. Maybe that just cost me more in terms of instructions that says where this operand is going. We'll get into that.

But here, what this allows me to do is exploit that locality better. Same instruction chain, I can put the first operation on one ALU, I can put the other operation on the second ALU. And here, rather than putting it for example here, which would send the operand really far across chip, what I want to do is recognize that there's a producer/consumer relationship here. I want to exploit that locality and have them close in spaces so that the routes remain fairly short.

You know what I can also do is sort of pipeline this network so that I can have the hardware essential match computation flow. If one ALU is producing a lot of results at a lot faster rate than for example this instruction can consume them, then the hardware can take care of, for example, blocking or stalling the producing processor so it doesn't get too far ahead. It gives you a nature mechanism for regulating the flow data on the chip.

Well, this is better than what we saw before because with the crossbar you're not really getting any scalability in terms of your latency transport operands from one ALU to another. Whereas with on-chip network, if you've taken routing classes, you know that there exists an algorithm that sort of allows it to route things at least the square root of n, where n is the number of things that are communicating in your network.

But if you're doing locality driven placement then it's essentially costing time. And in a raw chip,

it's in fact three cycles. So you can send one operand from one tile to another in three cycles. And we'll get into how that number comes about. So this is much better. But what it does is increase the complexity on the compiler. It says, this is my computation, how do you map it efficiently so that things are clustered in space well so that I don't have these really long routes for communication?

But then we can look at what else can we distribute. Well, we have the register file. We can distribute that across all the ALUs. And that essentially decreases that n cubed relationships between ALUs and register file ports to something that's a lot more tractable. Where it's one small register per ALU. And this is better in terms of scalability, but we haven't solved the entire problem in that we still have one global program counter, we have one global instruction fetch unit, one global control unified load/store queue for communicating with memory. And those all have scalability problems. So whereas we fixed the problem with the crossbar-- that becomes more scalable-- we haven't really fix the problems with the others.

So what's the natural solution to do here? Well, we'll just distribute everything else. And so you start off with each ALU here now having it's own program counter, its own instruction cache, it's own data cache. And it has its register file ALU and everybody-- that same sort of design pattern is repeated for each one of those ALUs. So now it looks like it's a lot scalable. I don't have any global wires. There's no global centralized data structure. And all of that means I can do things more-- I can do things faster and more efficiently. And what you start seeing here is this sort of tile processor coming about all. So each one of those things was exactly the same.

And what was done in the raw processor is that none of those tiles was longer than you can communicate in one clock cycle. So this solved essentially a wire delay problem as well. So if this is the distance that a wire-- that a signal can travel in one clock cycle, the tile is smaller. It can fit within this circle. So that means that you're guaranteed-- you have better scalability problems. You're solving the issues that people are facing with wire delay.

And in terms of the tile processor abstraction, Michael Taylor was is a PhD student in the raw group, his thesis sort of identified the tile processor approach and this aspect of the tile processor approach that makes it more attractive, the SON. Which is the scalar operand network. And the next two slides, the next part of the lecture, is going to really focus on what that means. He argues why the tile processor approach is scalable. And it's scalable for the same reasons as multicores. You just add more and more cores on a chip. But the intrinsic difference between the multicore that you see today and the raw architecture is the scalar

operand network.

So I'm going to ask you questions about this in a few slides. But really what you're getting here is the ability to communicate from one processor to another very efficiently. And the way you do this on raw is you have your instruction fetch d code, register file read stage, WALU-- your competition pipeline. But part of the registers-- the new register file-- so 24 through 27 are network mapped. So what that means is, if I write-- if one of the operations that I have in my computation has a destination register that's 24, 25, 26 or 27, that value automatically get sent to the output network. And if I have a value-- if one of my source operands is registered at 24, 25, 26 or 27, implicitly that means get that value off the network.

And so I can have add 25-- added to register 25-- so this is one of the network map ports, sum two operands. So this is a picture of the raw chip. This is one tile. This is the other tile. So you can sort of see the computation and the network switch processor here. So the operand flows into the network and then gets transported across from one tile to the other. And then gets injected into the other tiles compute networks. And here this instruction has sort a source operand that that's register map operand. So it knows where to get its value from. And then you can do the computation.

An interesting aspect here is that while you've seen instructions like this, just normal instructions, here you also have explicit routing instructions that are executed on the switch processor. So the switch processor here says take the value that's coming from my processor and send it east. So each processor can send values east, west, north or south. So it can go to the tile above it, the tile below it, the tile to the left of it or tile to the right of it.

And so sending it east sends it along this wire here. And then this particular switch processor says get a value from the west port and send it to my processor. Now you could have had here, this process could say, this value is not for me, so I want to just pass through to some other processor. So you can pass it from the west port to the south port or to the north port or just pass it through laterally to the other east port.

So it just allows you to essentially just have an on-chip network and not operand-- you can imagine having an operand that has a data packet and header that's says, I'm going to tile 10 and the switches know which way to send it. But the interesting aspect here is that the compiler actually orchestrates the communication, so you don't need that extra header that says, I'm going to tile 10. You just have to generate a schedule of how to write that data

through. So we'll get into what that means for the compiler in terms of that added complexity.

So communication on multicores is expensive for the following reasons. And this is really sort of going contrast or going to put the scalar operand network into slightly more perspective. But first, so how do you communicate between multicores on the cell? You have the DMA transfers from one SPE to another. You can't really ship an operand single value. So if I write the value x, and I want to send x from one SPE to another, I can't really do that very efficiently, right? So this is essentially the contrasting thing between multicore processors that largely exist today and the raw processor. So I've shown you an empirical-- a quantitative-- an analytical model for communication costs before in earlier slides.

This is an illustration of that concept. So if I have a processor that's talking to another, that value has to travel across some network and there's some transport costs associated with that. But there's also some added complexities. So there were lots of terms, if you remember, in that really big equation I've shown before. You have some overhead in terms of packaging the data. And you have some overhead in terms of unpacking the data. So what does that look?

Well, there are two components we're going to break this down to: the send occupancy and send latency. And I'm going to talk about each of those. And similarly on the receive side, you have the receive latency and the receive occupancy. So bear in mind, this lifetime of a message essentially has to flow through these five components. It has to go through the occupancy stage, then there's the send latency, transport, receive latency and receive occupancy before you can actually use it to compute on.

So what are some things that you do here? Well, it's things that you've done on cell for getting VME transfers to work. You have to figure who the destination is, what is the value, maybe you have an idea associated with it, a tag, things of that sort. And you have to essentially inject that message into the network. So there's some latency associated with that. Maybe your-- on cell you have a DMA engine which essentially hides this latency for you. Because you can essentially just send the message to the DMA, right into its queue. And you can especially forget about it unless it stalls because the DMA list is full.

On the receive side, you sort of have a similar thing. You have to get the network to inject that value into the processor and then you have to depackage it, demultiplex it and put it into some form that you can actually use to operate on it.

So this 5-tuple is gives us a way of sort of characterizing communication patterns on different architectures. So I can contrast, for example, raw versus the traditional microprocessor. So this is a traditional superscalar. A traditional superscalar essentially has all the sophisticated circuitry that allows you to essentially bypass network. You can have an operand directly flowing to another ALU through all the n squared wires in the crossbar. And a lot of dynamic scheduling is going on. So it really has no occupancy, latency, you're not really doing any packaging of the operands. Your transport cost is essentially completely hidden. You have no complexity on the receive side. So it's really efficient. So this is essentially what you want to get to go: this kind of 5-tuple. But as we saw before, it's really not scalable because the wire complexity woes-- whether it's n squared or n cubed, that's not good from an energy efficient point of view.

Scalable multiprocessors-- these are on-chip multiprocessors more indicative of things that you have today-- have this kind of 5-tuple where you have about 16 cycles just to get a message out, know roughly 3 cycles are so to transport message. So maybe this is being done through a shared cache. Which is how a lot of architecture communicates between processors today. And you have to sort of demultiplex the message on the receive side. So that adds some latency.

In raw, because you have these net memory map registers on the input side and the output side, you really can knock down the complexity from the send side in terms of the occupancy and latency to zero. And you just write the values to the register. And it looks like a normal register, right? But it just magically appears on the network. And then from one tile to another, it's one cycle to ship the value across that one link from one switch processor to the other, as long as it's a near neighbor. And then two cycles to inject the network into the tile processor. And then you're ready to use it. So in this space, where would you put cell is the question? Anybody have any ideas? What would the communication panel look like on cell?

So you have to do explicit sends and receives. So let's look at this. So can we get rid of this stage on cell which is essentially saying packaging up my message, is it's no, right? Because you have to essentially say where that DMA transfer is going to go to-- which region of memory? So you're buildings these control blocks.

And then the send latency here is roughly zero, because you have the DMA processor which allows that kind of concurrency between communication and computation, so you can hide essentially that part of the transport-- that part of communication costs. Your transport costs

here, you have this really massive bandwidth, this really high bandwidth interconnect on the chip. So this makes it reasonably fast, but it's still a few cycles. There's no near neighbor? Yeah, a hundred cycles to go near neighbor communication. Because you're still-- you don't have that fast mechanism of being able to send things points to point. You're putting things on the bus and there's some complexity there.

On the receive, you have the same kind of complexity that you had on the send side. You have to know that the message is coming, that can be done in different ways. And then you have to take that message and write it into your local store. Which also adds some overhead in terms of the communication cost. So the cell would probably be somewhere up here, I would imagine. I didn't have a chance to get the numbers. If I do, I'll update the slide later on.

OK, so that's essentially a brief insight into the raw-- yeah?

AUDIENCE: Where did you get the scalable processor?

PROFESSOR RABBAH: So these are from Michael Taylor's thesis. So I believe what he's done here is just looked at some existing microprocessor and essentially benchmarked communication latency from one processor to another.

AUDIENCE: So this is like going through the cache on the [OBSCURED]?

PROFESSOR RABBAH: That's in fact how you-- a lot of these multiprocessors today have shared caches, either L-1 and more so now it's L-2. So if you have-- L-1s are dedicated to different processors. But you still have to go the memory to communicate. So the raw parallelizing compiler-- yeah? Another question?

AUDIENCE: You might want to postpone this question. Two related questions: so raw has-- I guess raw has pretty well optimized nearest neighbor communication. But we know from, for example, Red's Rule in heuristic and intellectual engineering about the number of wires needed for a given area. Is that in between-- as I recall, it's the minimum for a good sized circuit is proportional to the perimeter, or roughly the square root of the area. And it ranges from there to-- not proportional to the area. There's something in between. Something with 3 in it. Like to the 3/2 power I think, perhaps. No, something like 2/3rds, something like-- yeah, 2/3rds power. So the area to the 1/2 power or area to the 2/3rds power. So Red's Rule says the number of wires you need is roughly in that area.

And so that sort of pushes that-- so the minimum you need is the nearest communication. And often you need more than that. We know from the FPGA experience that nearest neighbor communication is not-- or, at least, it's good to have move than nearest neighbor, and that often long wires followed across the chip, in extremely high--

PROFESSOR RABBAH: So I'm going to actually show you an example where nearest neighbor is good but you might also want some global mechanism for control orchestration for example.

AUDIENCE: Not just for con-- not surely just for control but for broadcast, for arbitrary for the computation to use, not just for the chip to use. Like why are you scaling out two hops, four hops, fewer and fewer wire--

PROFESSOR RABBAH: Yes, in fact what I think is going to happen is a lot of these chip designs are going to be heirarchical. You have some really global type communication at the highest level. And then as you get within each one of the processors, then you see things at the lowest level, something that looked like raw. So you can build sort of a hierarchy of communication stages that allow you to sort of solve that problem. But all of that adds complexity, right? First you have to solve the problem of how do you parallelize for just a fixed number of cores and then figure out the communications. Once we understand how to do that well with a nice programming model then you can build heirarchically on that.

AUDIENCE: On the other hand, it might make the compiler's job easier because it's not as constrained.

PROFESSOR RABBAH: It might give you a nice fall back rate. It might save you in cases where there are things that are hard to do. There are some issues in the last two-- the second to the last three slides. We'll talk about an example of where that might be the case.

AUDIENCE: Another question which [OBSCURED] so raw, I guess, being simpled and tiled, I guess one of the selling points I think was that it really cuts down on the engineering effort.

PROFESSOR RABBAH: Oh, absolutely. This was done a million gates in-house for [OBSCURED]

AUDIENCE: So a company like Intel has a ridiculous number of engineers. And to get a competitive edge, they something they want to apply more engineering to it. And so the question is, where might you apply more engineering to try to squeeze more--

PROFESSOR That's the million dollar question that everybody's looking at. Because if somehow Intel

AMARASINGHE: thought they could add more and more engineering. And then build this very complex full-scale [OBSCURED] But separate vessels. And so I think there's still a lot of things that is wrong. Meaning it's [OBSCURED] so at Intel basically they will let you do something like that. They will put a lot of engineers doing each of these components, finding very few, and they can get a lot more performance, a lot less power and stuff like that. So depending on what you want, science is not everything. There are a lot of other things [OBSCURED] So while it makes it easier? [OBSCURED] And the key thing is, you start something simple and as you go on, you can add more and more complexity. Just, as there's more things to do.

PROFESSOR RABBAH: Part of the complexity might be going to-- not making all those [OBSCURED]. OK, so raw pushes a lot of the complexity into the compiler in that the compiler now has to do at least two things. It has to distribute the instructions. You have a single program and you have to figure out how to parallelize it across multiple cores. But not only that, because you have the scalar operand network, you have to figure out how the different cores have to talk to each other. So you have to essentially generate schedule for the switch processors as well.

So I'm going to talk a little bit about the raw paralyzing compiler. And this is different from a StreamIT parallelizing compiler which really talks about a different program as an input, using a different language. This is work again done here at MIT by Walter Lee who graduated two years ago. We have a sequential program. You inject it into raw C seed, raw C compiler, and you get fine-grained Orchestrated Parallel execution. And what the compiler does is worry about data distribution just like you have to do on cell in terms of which memory goes into which local store. which competition operates on-- the raw compiler has to worry about which computation operates on which data element and can you put that data in the right caches for each of the different tiles.

Instruction distribution: so the way this compiler essentially get parallelism, it's going to look at instruction level parallelism in your application. And it's going to divide that up among the different cores. And then the last step is the coordination of communication in control flow. So I'm just going to briefly step through each one of those.

So the data distribution really has essentially trying to solve the problem of locality. You have two instructions. A load into r1 from some address and then you're adding r1. You're incrementing that value. And you might write it back for later on.

So where would you put these two instructions? So to exploit the locality, then you want the

data-- if the data is here, then you want these two instructions to be on this tile. If the data is here, then you want these two instructions to be on this file. Because it doesn't help you to have the data here and the instructions here. Because what do you have to do in that case? You have to send a message that says, send me this data. And then you have to wait for it to come in and then you have to operate on it. And then maybe you have to write it back.

So the compiler sort of worries about the data distribution. It applies some data analysis. A lot of a thing that you saw in Saman's lecture on classic parallelization technology. Sort of figure out the interdependencies and then they can figure out how to split up the data across the different cores. And there's some other work done by other students in the group that tried to address this problem.

The instruction distribution is perhaps as complicated and interesting. In here, what's going on is-- let's say you have a base block. So you take your sequential program. You figure out what are the different basic blocks of computation that you have and within the basic block you have lots of instructions. So each one of these green boxes is a particular instruction. And what you're seeing-- these arrows here that connect the edges-- are operands that you have to exchange. So you might have-- this is an add instruction. It requires a value coming from here. Multiply-- subtract instruction requires values coming in from different areas. So how would you distribute this across a number of cores-- or across a number of tiles? Any ideas here?

So you can look for, for example, some chains that are not interconnected. So you can look for clusters that you can use. And say, OK, well I see no edges here so maybe I can put this on one tile. And then maybe I can put some of these instructions on another tile. Because sort of the communication flow is local. So maybe one strategy might be, look for the longest single chains so you can keep the communication flow. And then you apply and make and algorithm, come up with a number of clusters.

Something like that does happen. And keep in mind from the lectures we talked about the parallelizing compiler, you have to worry about parallelism versus communication. Some the more you distribute things, the more communication you have to get right. So here we're showing-- what I'm showing is color mapping from the original instructions in the base block to the same instructions, but now each color essential represents a different cluster or essentially code that would map a different thread. So blue is one thread, yellow is another, green is another, red, purple, and so on. But I have to worry about communication between the different colors because they're essentially two different threads. They're going to run on two

different processors or two different tiles. So those arrows that are highlighted in dark black are communication edges. They have to explicitly send the operands around. Right?

So then I might look at the granularity. What is my communication cost? What is my computation cost? And I want to worry about load balancing. As we saw, load balancing can give you how it can better make use of your architecture and give you better utilization, better throughput. So you might essentially say, it doesn't-- it's not worthwhile to have these running on a different tile because there's a lot of communication going on. So maybe I'd want to fuse those together. Keep the communication local. And essentially eliminate costly communication. So there are different heuristics that you can apply. You can use that 5-tuple. You can use heuristic space on the 5-tuple to determine when it's profitable to break things up and when it's not.

And then you have to worry about placement. So you don't quite have this on cell in that you create these SPE threads and they can run on any SPE in the raw compiler. You can actually exploit the spacial characteristics of the chip in the point-to-point communication network to say, I want to put these two threads on tile 1 and tile 2, where tile 1 and tile 2 are adjacent to each other. Because I have a well-defined communication pattern that I'm going to use. And map to the communication network on the chip to get really fast, really low latency.

So you can take each one of these colors, place it on a different tile. And now you have these wires that are going across these tiles which essentially represent communication. But now the tile has to worry about, oh, I have to essentially send these on fixed routes. There's no arbitrary communication mechanism. So if there's data going from this tile to this tile, it actually has to be routed through a network. And that might mean getting routing through somebody else's tile.

So the next stage would be communication coordination. You have to figure out which switch you need to go to and what do you do to get that operand to the right switch which then gets it to the right processor. So here, I believe the heuristic is to do dimension order routing so you send along the x-dimension and then the y-dimension. I might have those reversed. I don't know.

And then finally, now you've figured out your communication patterns, you've figured out your instructions, you do some instructions scheduling. And what you can do here, because the communication patterns are static, you've split up the instructions so you know when you need

to ship data around and how. You can guarantee deadlock freedom by carefully ordering your send and receive pairs. So what you see here, every time you see an instruction that needs to ship an operand around, there's the equivalent of a route instruction that has route east, west, north, south. There's an equivalent route instruction on the other processors. And that allows you to essentially analyze code and say, OK, I've laid these things out carefully, I've orchestrated my send and receive pairs so I can guarantee, for example, there are no overlapping routes. Or that there are no deadlocks because one is trying to shift the other while the other is also trying to ship, and they both block on the shared network link. And finally, you have the code representation. So this is where you package things up into object files, into essentially things like threads. And then you can compile them and run them.

Now the question that was posed earlier is, well there's one thing we haven't talked about and that's branching. This is a sequential program, it executes branches. And now I have this loop that I've split up across a number of tiles, how do I know who's going to do the branch? And if one tile is doing the branch, how does it communicate with everybody else? Or if I'm going to repeat the branch on every file, does that mean I'm redoing too much computation on every other tile? So control coordination is actually quite an interesting aspect of-- adds another interesting aspect to the parallelization for raw.

So what you have to do is figure out-- there are two different ways you can do it. Because you have no mechanism for a global message on raw, you can't say, I've taken a branch, everybody go to this program counter. You essentially have to send either the branch result so one tile can do the comparison, it calculates the condition, and then it has to communicate x to each of the different branches-- to each of the different tiles. Or every tiles has to essentially just replicate the control and redo the computations. So every tile figures out what is the condition, what are the conditions for the branch. They redundantly do that computation and then they can all merge at the same time-- at different times.

So that gives you two ways of doing the branching. If each tile's doing its own control flow calculation, then they can essentially branch at different times. But if they're all going to wait for the result to compare, then it essentially gives you points where you have to synchronize. Everybody's going to wait for the result of the branch. But the latency could be different. Because if I'm sending the branch condition to one tile versus another file, and if one's closer than the other. Then the branch that's closer to me-- the tile that's closer to me will take that branch earlier in time. So you get sort of the effective of a global asynchronous branching in

either case. Does that make sense? So, in summary, the raw architecture is really a tile microprocessor. It incorporates the best elements from superscalars in terms of a really low latency communication network between tiles which really cuts down on the communication costs. And as we saw, and as probably you've been learning, communication is really an expensive part of parallelization on existing multicore chips. And it's also getting the scalability of multicores in terms of explicit parallelism but also gives you implicit parallelism because the networks are pipelined and they can give you full control.

So you're trying to get to the point where you have a tile processor with scalar operand network that allows you to do communication with a very low cost. And it might be the case in the future that these chips will especially be-- more complex architectures will sit on top of these so you'll use these as fundamental building blocks. And there was the 80 chip multicore from Intel: there have been rumors that that might actually be something like a graphics processor that has something like a scalar operand network because you could communicate with a very fast-- with very low latency between tiles.

And in that article which came out a few months ago was the first time I think that I had seen tile architectures used in literature or in publications. So I think you'll see more of these kinds of designs pattern appear as people scale out to more than 2 cores, 4 cores, 8 cores and so on, where you could still communication reasonably well with caches.

And that's all I prepared for today. Any other questions? And this is a list of people who contributed to the raw project. A lot of students who are led by Anant and Saman.

PROFESSOR AMARASINGHE: [OBSCURED] view of what happened in our groups and then how it relates to necessary to what you need. But this is trying to take it to a much finer grain. Whereas in Cell, of course, the message has to be large, you can do a lot of coarse grain stuff. But in raw, you try to do much more fine grain stuff. But we're going to talk about it the next lecture on the future. [OBSCURED]

AUDIENCE: [OBSCURED] Don't you need long wires for the clock.

**PROFESSOR
RABBAH:**   There's no global clock.

AUDIENCE: So you have this network that seems to -- So that the network actually requires handshaking?
Or--

PROFESSOR AMARASINGHE: The way you can do is, you can in modern processors, [OBSCURED] so since there's no long wire, you can actually carry the clock with the data. So in the global world, the switching here would happen when the switching here. But since there's no big wire connecting, then that's OK. So you can deal with clock ticking.

AUDIENCE: So this is not going to be not clock drift because--

PROFESSOR AMARASINGHE: Yeah, that's clock drift. One end of the process clock is happening at the global instant time at the other end of the processor. And since the wires also kind of go in the tree, you can deal with that.

AUDIENCE: Drift meaning ticking at different rates, not just--

PROFESSOR AMARASINGHE: Yeah, I know. Basically I don't think I can go back to it. It has a skew. There's a clock skew going in between those.

AUDIENCE: So you don't need synchronizers between the different tiles?

PROFESSOR AMARASINGHE: No, we don't need synchronizers because tiles are local. The clock would bring those tiles. The clock would bring two things that communicate close enough that it fits it in the cycle. But for example, if you get it two very far away branches of a tree and then if you try to communicate with them then you have a problem. Another thing is when the tree goes here, you want to use two different branches it's similar to going down. So you can compress the process. So there are all these things. I mean, modern processors really really destable. The problem occurs when you try to connect directly from the far end of the branch to something that gets clocked there to something that clocks at a very early end of the branch. If you're trying to connect those two, then the skew might be too long. Then you can get into clock trouble.

AUDIENCE: [OBSCURED] I was just worried about this local network. [OBSCURED]

AUDIENCE: Another question I had was in the mesh, obviously the processors in the middle have further to get to the I/O devices or to the main memory. What do you see happening as you get to larger and larger processors? Are they going to just put more and more local memory on the tile and [OBSCURED] it, or are they going to add extra memory buses on it?

PROFESSOR RABBAH: It could be a combination of both. So it's not just memory, I/O devices. If you're doing I/O then you might to be placed at a part of the chip that has direct access to an I/O device or very

close. It also comes up in the case of the communication orchestration. So if this guy is doing the branch then you want him essentially centrally located. So the best patterns for allocating things is essentially across. It's like a plus sign where it branches in the middle.

PROFESSOR AMARASINGHE: But that's not [OBSCURED]. You can make them uniform by everybody equally there. And a lot of times people have done that simple model with everybody equally there Or you try to take advantage of closeness and stuff like that. So you can't have both ways. So anytime you try to make me [OBSCURED] very, very close and fast access, you're are doing it by basically making the other parts to have less resources and less access. On the other hand, there are a lot of people working on [INAUDIBLE] things that, for example, there's a thing called tree space laser. So what that does is you put a mirror on top of the tile, on top of the processor. And each of these-- you can embed a small LED transmitter into the chip. So basically if you want to communicate with someone, you just bounce that laser on top of that and get it to the right guy.

So there are a lot of exotic things that might be able to solve this thing, technological problem. But in some case, speed of light-- I don't think an engineer has figured out how to break speed of light. Unless, of course, people go with quantum computing and stuff like that.

So, I mean the key thing is, you have resources, you have certain data and you just have to deal with it. Getting nice uniformity has a cost.

PROFESSOR RABBAH: Yeah, I mean, on the [OBSCURED] that are groups here at MIT who are working on optical networks in the third dimension. So you have a tile chip plus an optical network in the third dimension which allows you to do things like broadcast much more efficiently. OK?

PROFESSOR AMARASINGHE: I guess we'll take a break here and take a small, three-minute break and then we can go on to the next topic.