SAMAN AMARASINGHE: Today I'm going to talk a little bit about computer architecture. And how computer architecture impacts performance engineering. So the main part of this is going through a long overview of the Pentium, the Nahalem architecture that in the machines that you guys are using, which is hot off the press. Six core processor that's out there now.

And then I will talk a little bit about profiling a program. And then the next lecture, the TAs are going to demonstrate how to use some of those profiling tools that you guys are supposed to use for the project two. And I guess the project two will probably appear, since we had a delay, so probably be up here 24 delayed. We'll release the project, too, up there.

And then at the end of the talk, end of the lecture, I'm going to talk about a little bit of example. Go to some program and show what you can gain by looking at things like profiling, and what information that you gain. Just kind of do a high level view.

So let's start something. So how many of you have had a chip in your hand. Microprocessor in your hand. Some. There's some people who haven't. So I brought some show and tell things. So what I'm going to do is pass them around. And there are two things that are actually very valuable that I want to tell you and then make sure that they don't get damaged. So this is a Pentium 3 that's already in a big heat sink in there.

So you can't see that much, but they're really cute and they created a hologram of the chip and put it there so you can catch, I guess, people who are trying to use counterfeit and stuff like that. So this is MDK 6 with the packaging. I'm going to send it around in there.

And this is a Pentium Pro. Normally what happens is that when the first dye comes out, this is humongous, giganiticized. But after a couple of generations, what you do is you take the same circuit and you shrink it and shrink it. This is probably what two shrinks-- so the actual dye right now is very small because this is copper [UNINTELLIGIBLE PHRASE] later process that things have been shrunk.

This is a Core Two Duo So I think there's a two cores in there. So I didn't want to bring anything newer because those chips are darn expensive, and if you touch it, basically just static will basically destroy the process. So these are something that came out of things that doesn't work.

The next two things. These are-- I really need to get this [UNINTELLIGIBLE] because these are like the only thing that's the one available in this world. Basically there's only one of item. So at MIT, we build a process a couple of years ago called a row processor. So here's a row dye before it's being mounted. So this is what comes out after all the fabrication, before it goes into this mounting with all the pins.

If you look at the top of this one, what do you see? It's all these dots, which are basically where all the pins go. So you don't actually even see inside the dye in this one. However, then you pay a bunch of money to these people who go and grind that chip. You take a real working chip and you grind it, and expose insides of the chip.

So this is a ground chip that's exposed the metal layering. In fact, this you can see real circuits in the chip. So I don't think anybody has the mission that's required to see the transistors or even the circuits. But it's kind of interesting to see. So here are these two things, so some show and tell like that.

Ah-ha. And this. This is entire reason that microprocessor not getting faster anymore. Because this is one humongo heap thing that you put in the modern machines, and there's a huge fan that goes above that. And you can't keep building larger and larger heat things to get the heat out. And so that this why the heat is paramount, and that's why we can't run 4 gigahertz, 10 gigahertz process. We are thick into the current kind of gigahertz train, and then we [UNINTELLIGIBLE] multiple cores. So it's a big, humongo block in there, and then some actually might even have a bigger block.

So while this go around, let's start the lecture.

So computer architecting includes many different things. You are doing instructions, memory, IEO Bus, disk systems, GPU, graphics, all of those things. But we are not getting into beyond instruction and memory system. So we are going to focus on that, but if you really want to understand the full end to end performance, you have to worry about all those other things.

So that said, let's go into instructions and memory. So here is the Nehalem processor. And there's a beautiful picture of the processor in the left. This as kind of the role chip that I'm showing. They have ground out the upper layers and showed some of the layers in there. So that's what you get when you buy the Nehalem. And this very complicated diagram in the right side, it's an abstract notion of what's happening inside.

So what you do is go a little bit into this [UNINTELLIGIBLE] diagram, and trying to understand what impact it would have on you. This is not architecture class, but you, as trying to get performance, has to make sure all these components works pretty well.

That said, it's very hard to understand in a modern microprocessor exactly what's going on. Not even Intel understand what's going on most of the time. And so a lot of things [UNINTELLIGIBLE]. So you can't ask tell me exactly what happened or get there. So you can be fussy.

On the other hand, being fussy means there's a level of abstraction that you have to live with. And what that means is when something hit their head very hard, and have a really bad situation you can see that and you can react to that. But there are a lot of small things that happen. These things interact in very complex ways. So you don't understand exactly the minor detail what's going on in these microprocessor for a given application. A lot of different things might work different ways. So that's it.

So if we look at what you learn from 004, so what that means is instruction. You have here two instructions. One after the other. So if the instructions take [UNINTELLIGIBLE] cycle. So we see the non-[UNINTELLIGIBLE] method.

So you look at why it's probably taking five cycles. So that's because it's doing different things. You are doing instruction fetch, instruction decode, after that execute. Then you do some memory [UNINTELLIGIBLE] and then find the right packet in there.

So you do all those things [UNINTELLIGIBLE] instruction, and then you start the next [UNINTELLIGIBLE] cycle. Of course, this is very necessary. So the first two people [UNINTELLIGIBLE]. After I do instruction fetch, that logic that's doing instruction fetch is not doing anything for a while. Why don't I start the next instruction phase immediately because you are doing the same circuit, and then recycle, that's so it can go do the next thing, and next thing, and next thing.

And by doing that, well I can basically recycle, I can get the instruction through the system. So this looks very nice and simple and you can get. So what are they choosing here? Is the world this nice and simple? No. OK, what might happen?

AUDIENCE: [INAUDIBLE PHRASE]

SAMAN AMARASINGHE: You have a lot of issues that normally cause hazards. So there could be three different type of hazard. There's a thing called structural hazard. What you are trying to do is attempt to use the same hardware, do two things, but there's only one hardware. You can't get two things done. One has to be after another.

And there's a thing called data hazards. That means you're trying to run these things, one now another, but in logically. They have to run sequence serially. So that means that's a data dependence, I'll get into, that makes it impossible to make these things run in the pipeline fashion.

And finally, this control has things like branches and things like that, [UNINTELLIGIBLE] interfere with that. So let me get a little bit detail down here.

So the first thing we have is what we call data hazard. I will talk about two different hazards. So before I go there, I will share a little bit about this assembly representation. In two lectures, you're going to get a more deep drilling into the how to go from C to Assembly. But before that, so what this instruction says, this is the normal x86 form. We are doing add long. Add in to the values in this rbx and rax and put the results back into rax. So you are doing rbx plus rax and get the result into rax. You are subjecting rax with rcx and put the results into rcx.

So basically all the rest of the results are in the right-hand side. Two [UNINTELLIGIBLE] are basically the first and second. So the last one gets read and modified. So that's the way it's weighted. So while you have two is-- question?

AUDIENCE: [INAUDIBLE PHRASE]

SAMAN AMARASINGHE: Instruction, yeah. It's different concept. The second instruction is data dependent, of course. What's happening is I am writing this value here, which will be read by this guy here. So I write something and the next instruction is reading that. The problem is it's the right thing must not be available until very late in the pipeline. So they cannot execute simultaneously and all that thing basically of this dependence.

This is called basically read after write, because I'm going to read after it's been written. And if you look at the pipeline here, what happens is the write happens here, and the next feed has to happen somewhere down here. So until this one, this slide [UNINTELLIGIBLE] so this status has to get delayed after this point. This make sense? Because I'm trying to in here read something that's not going to be produced for two more clock cycles. So that's not available and I can't do that. Make sense? OK.

So this next cycle dependence is called name dependence. That's the dependence and anti-dependence. Basically what it's doing is two instructions are using the same register. And because of that I can't start to see the radius there until the other one switches off with the register. So in here, what's happening here is I am subtracting rax and rbx. I'm putting the value in rbx. And here I am adding rcx to rax, and basically start putting the value in rax.

The problem here is I cannot modify rax before this guy has [UNINTELLIGIBLE] the value in here. So that means I'm trying to go modify it, and say no, no, no. You can't touch it because somebody will still want to get the value, and I can't go destroy the value. And I had to [UNINTELLIGIBLE]. That's the first type of what you call anti-dependence. And because of this, you both are using rax. There's no real data movement, but it basically has the same space. I had to wait till the other person can be evicted before I can read that register.

So these are called write after read hazard. The other type of dependence is called output dependence. So I am updating rax twice, and of course in this simple example we can probably even drop this instruction because it doesn't matter because I'm re-writing it. But if there's something in between also reading, what happens is I can't do this in wrong order. The last value that updated has to be this instruction. So we had to make sure this instruction happens after this instruction. So we have a dependence, so there's some ordering in here that has to be maintained. And this is call write after write hazard.

So instructions that have medium dependence, we can actually get rid of the dependent by what we call register renaming. So everybody uses rax, so that's kind of artifact of good old inter [UNINTELLIGIBLE] architecture sometimes [UNINTELLIGIBLE] when register, so you have to use the same register for many, many things.

So in a modern-- inside the hardware there's what we call a register renaming. You have lots more slots for the same registers, and even when you're using rax, this is an old one, this is a new one, use a different location for the new one. So I can do renaming. And then basically hardware can get rid of that.

The next interesting thing is control hazard. So here what we see, if you had this kind of a loop, s1, it's control dependent on p1. So we can't do s1 until p1 is done. Or s2 until p2 is done. So we are [UNINTELLIGIBLE] the p1 condition is

[UNINTELLIGIBLE] before you can do s1 and [UNINTELLIGIBLE] for the next one.

So the interesting thing is control dependence also we can get rid of it by doing speculation. So the idea there what hardware does is it will say wait a minute. I know I have to wait till p1 is [UNINTELLIGIBLE] to do s1. But I'm going to do s1 anyway. I will just go to s1, speculatively. And then at the end of doing that, at some time when p1 is calculated, and I know what the p1, I said did I do it right? If I did it right, I'm good. If I did something wrong, I have done some [UNINTELLIGIBLE] that is not useful for that.

Question?

AUDIENCE: [INAUDIBLE PHRASE]

SAMAN AMARASINGHE: So what happens is there's this thing called a write buffer. The complex thing is at the end of the day, the instructions have to be get committed in the order they arrive. So before committing, you keep the state in buffers without [UNINTELLIGIBLE] into the main one.

AUDIENCE: [INAUDIBLE PHRASE]

SAMAN AMARASINGHE: So right back into memory doesn't happen until the commit point. But when you read something, before reading the memory, say OK, I have updated something, is it in the right buffer? So I read things from the right buffer in there. So you had to go in order commitment. Because you can't do out of order commitment, that can do crazy things. You do in order commitment, but inside the metrics things can go easier. I'm just kind of jumping the gun a little bit.

So in a modern Nehalem processor, these are kind of Intel [UNINTELLIGIBLE], so you don't know exactly what it is. In some place it says it has 16 clock cycles. So it says it might be 16 stages of pipeline. And another place it says 20 to 24 stages of pipeline. OK, if you work for Intel and sign the NDA you will get the exact number. But it doesn't matter. It's a lot of pipeline stages, so that's all you need to know.

So what happens is if you are doing this in sort of five stage, like what you did in probably beta. You do instruction decode, instruction queue, pre-record queue, decode, register rename, and allocate registers. And then there's thing called reservation and stations that wait for when data's available to execute that instruction. And then execute, and then basically go there. This is what normally instruction life would be, if everything goes one after another after another.

So what to get out of this is the pipelines are long. So pipeline stores and stuff can be expensive.

So the other thing you can do in abstract way is do multiple issue. So if you do pipeline, what happens is you do one at a time so you can use the pipeline stages nicely. How about instead of having one unit, if you have integer unit and a floating point unit. And every clock cycle you can basically abstract. You could say look, I'm taking interger instruction, I'm taking floating point instruction, and take them both together [UNINTELLIGIBLE]. And then whala, at the end of the day in one clock cycle I can [UNINTELLIGIBLE] two instructions, one integer, one floating point, if you have two different sets in there.

So this is also very nice. You can get a lot of what we call super scaler performance. However, it's called instruction level parallelism. There's a lot of problems, also things that you worry. Well first of all, you have to have enough instruction in level parallel. Because the instruction you get is the sequence history, and you see one after another. I mean you write your program and compile. It looks like you run one instruction, finish it, run another, run another. So in order to form parallely, you're to find things. You can actually do parallel. That takes time.

And of course, between hardware and software you are to maintain that preserve order of instructions, because you want to make sure that it looks like and it feels like you ran one after another after another. You don't want things to run in a haphazard way and create arbitrary result. And so you have to, again, things like this, data dependences, control dependences have to be satisfied when you're getting this parallelism.

So data dependency, there's a hazard, and determining which order you can run things. So for example, output dependence. Say look, I can't just run them parallel. I have to make one after another after another. You're to get all those things done. And also if you're lot of dependency kind of gives you bounds, how much parallelism you can get. If things are all one dependent against one after another, you can't run them parallel. You have to wait to run things to another one.

So by looking at data dependence, as you can figure out, OK look, did I get good performance, good ILT or not? So what we want to do is exploit this parallelism like we see in the program order. And basically make sure that you always get the same result on that.

One way of getting parallelism that is in modern process is called multimedia instruction. It's called SIMD in academic circle, something like Single Instruction Multiple Data, and it's called data level parallelism, and Intel, of course, has to give the [UNINTELLIGIBLE] name they call SSE, they just call it MMX. So the idea there is look, they are building this wide measure. They can easily build 128 bit wide register. But you most probably don't need 128 bit data, because that's too big. But they can build this large.

And most will tell you about happy with about 32 bit data. So what? [UNINTELLIGIBLE] wait a minute. You build this wide thing. But you can chop it into four pieces or eight pieces or two pieces, what you want. So here we chop that large thing into four pieces.

And then what you can do is in a single instruction, you can instead of doing one large add, the same type of adder, you can add four separate things. Four small parts. So you're assuring add, but instead of adding 228 bit data, what you are doing is adding here, four 32 bit data.

The entire architecture looks very identical to an 228 bit data. It's basically to remember your double 004, it's like a carry. Then just have to come the carry. You don't carry from here to here, and do that. So this is you're loading this large chunk, and you are just working on these things. And whala, by doing that, having the same kind of circuitry, you get much better parallelism. And if you have like 8 bit data, boy, you can get lots of parallelism.

So there's all these multiple instructions in there. And so what you can do is you can have a loop like this. So you can, in every situation, you can add the I to BI, and create pretty good [UNINTELLIGIBLE]. So you can do one at a time.

But assume you have 32 bit values. Instead of doing-- single instruction you can load 0 to 8 3 in one go. B0 to BC in one go. Just do the add of all four in one, go [UNINTELLIGIBLE]. So in single time you get four things done. So this is really cool. And if you remember my first lecture, we showed this matrix multiplier. If you get [UNINTELLIGIBLE] in there, you get a lot less instructions executed. We got a nice [UNINTELLIGIBLE].

So I'm going down, so the top path I'm showing, the C code-- for that loop, and then the bottom half in there, I'm showing the normal assembly that we [UNINTELLIGIBLE], what you see. And once you start understanding assembly a little bit more, you can go ahead and read this and see what's going on. And then if you [UNINTELLIGIBLE], you get a humongous beast like this. This has unrolled [UNINTELLIGIBLE], whatever. And this is the kind of code that we generated by [UNINTELLIGIBLE]. And that's just much, much faster than the one on the left. Any questions so far? OK.

So when we have this superscalar [UNINTELLIGIBLE], with multiple execution, so what you can do is in a clock cycle, you can do a lot of things. So in the [UNINTELLIGIBLE] processor, you have basically six things, six execution things that can happen in one clock cycle. It's not the same thing, so you can do a data store. You can store address, load address from there, and then you can do three different type of executions. There are three units in here. In here you can do either integer or SSE, add or move instruction, this unit can do this kind of instruction, you can have a floating point add-in around here. This one needs to also have this bunch of different things you can do. So this complex hardware, we'll kind of figure out, OK, if you have these instructions, figure out what of these [UNINTELLIGIBLE] are available. If the instruction is able to execute, it will put it into one of these and het the results out.

So if you're lucky, what you can see is, you might get about six instructions going each cycle. So if you look at the mesh here, what we call clock [UNINTELLIGIBLE] instruction, it could be 1/6. If you're lucky. If you can run the mesh into the metal, that means everything is working all the time. It's very hard to find that kind of program, but [UNINTELLIGIBLE] can get something like that. That's really cool.

So what happens is to get this thing, they do things called out-of-order instruction. So what that means is, issuing this varying number of instructions in the clock, in here, in the Nehalem processor, you can store 128 instructions in this core instruction buffer. [INAUDIBLE] better name than instruction buffer. If you can see that, [INAUDIBLE] very far. But [INAUDIBLE] up there, sorry. Here.

It's called reservation stations. So when you put it in, all those [UNINTELLIGIBLE], reformation [UNINTELLIGIBLE], ready to get, and the minute it becomes available, that means all the data is ready for it to run, you can basically issue that and run.

So what they're doing is, in these hundreds of instructions, you're figuring out the patterns you can run. And a minute some part is available, the data becomes available, you can run that. And so that means the program might have instructions in certain order. It gets executed in very different orders than what the program has. So that's what call you can give things like [UNINTELLIGIBLE] register, [UNINTELLIGIBLE] dependencies, [UNINTELLIGIBLE] execution. And sometimes you might run speculator. You might not know. You might have the data, but there might be controlled dependence. You might not know that it's actually is supposed to happen, but it says wait a minute, I have execution available. It's not doing anything. OK, let's run it and keep the results, and if it is actually useful, we'll use it. Otherwise, we'll throw it away. You can do things like that. And that's a speculation.

So another type of speculation-- speculation mostly means, you want to do something, but you don't know whether you can do it. So how do you go about doing that? The way that interprocess and modern processor do, is they do a lot of predictions. It has like this is [UNINTELLIGIBLE] say, look at the crystal ball and say, aha, this is the future. And it does that [UNINTELLIGIBLE] really many things. So it does things like branch prediction, value prediction, prefetching.

So it says, aha, I don't know where you're going, but I think you are going home after the class. And then probably if I said something like this, it's probably right for most of the people. And actually, no, in this class, most of you, again, will probably go to sleep after the class, because of the project. So if I do that probably I do a very good prediction. How many of you are going to go to sleep after the class? Huh, my prediction is not that good. OK. We'll see.

So you can go all these kinds of things, identify normal behavior programs to deal with that. Ad what prefetching says is, if I look at what you have been taking from the money, there are some patterns, and I'm expecting you to [UNINTELLIGIBLE] that pattern, and I will basically get the data, even though you don't ask for it, because I'm expecting you to be doing that. It's like a very good butler who can really think, you ask something, have that ready for you, basically.

So by doing that, you can get much better parallelism, because now you're not waiting for that to be sure that you can do it. You're just kind of doing this, assuming you might be using it, and if you do it really well, you're actually going to use it. And that will be useful.

So for example, even things like value prediction. Value prediction says, [UNINTELLIGIBLE] to get out, you know when you do this complex calculation, most of the time, data is zero. You multiply and add a lot of zeroes, and if you're doing spreadsheets, all it is [UNINTELLIGIBLE] calculations. So if I don't know the value, let me assume it's 0. And that actually works most of the time. Or you look at the last couple of times, what has happened to that [UNINTELLIGIBLE], and there, let me predict something. And people to find these kind of things that are a lot of patterns in [UNINTELLIGIBLE] program. And if it works, that's really great.

So in speculation, what you do is first issue and execute. As if the branch was-- I already knew where it is going, just by using the prediction. And if I just know dynamic scheduling, I am only doing [UNINTELLIGIBLE] issues, I am not going to execute until I know everything is [UNINTELLIGIBLE]. So it's speculation goes one step beyond.

It's what we call a data flow execution model, the dynamic schedule. And that means, the minute the data is ready and everything is ready, you can execute, even though it might not mean the same mode of the program. It might be much later instruction. But implicitly, you can execute that. So again, the [UNINTELLIGIBLE] pipeline, you can do 20 to 24 stage or 16, whatever is correct.

The other thing it does is this crazy thing called micro-ops that puts everybody out to a loop. So what happened was-- this is kind of historical-- Intel, a long time ago, came up with the x86 architecture. They came up with this instruction set. That instruction set is called a CISC instruction set. It's a complex instruction set architecture. What that means is there are these instructions there are that are really, really complex. You can move an entire string from one place to another in one instruction. The problem is, most instructions cannot be done in one cycle.

So what modern [UNINTELLIGIBLE] did was, they built a [UNINTELLIGIBLE] compiler into the hardware. So what happens inside this process is, take this CISC instruction and compile them down to these particular micro-operations, that each operation is small can be done in basically one cycle. So it's doing this mapping in there. And then inside the computer, it's basically dealing with micro-ops. So up to here, it's dealing with these long instructions, and here it's precoding and decoding into micro-ops. So after that, so what that means, every cycle, it can issue 6 micro-ops. So there might be 6 instructions you know, 3, or even 1, depending on how many micro-ops [UNINTELLIGIBLE].

So if you look at instruction manual in the [UNINTELLIGIBLE], you can see how many micro-op [UNINTELLIGIBLE] instruction it's going into, and if it's large, that means those instructions are slow. And you can have this 120 micro-op waiting to be executed, sitting in this reservation station. Basically, they are waiting there. The minute the data is available, and then a slot available, it will go down, get executed, and come back.

So branch prediction, basically-- the problem with branches is if you have this very nice pipeline, what happens is, branch target is not known until this target detection time. Where is my mouse? OK. Until this point, I don't know whether the branch, where I am supposed to go. And so how do I go ahead and get the instruction, because I don't know where I'm going until this point. Even worse, I might not even know address. Some [UNINTELLIGIBLE] you are going AOB, you are taken or not taken. But sometimes there might be [UNINTELLIGIBLE] branches. I don't even, if I'm returning the return address in the stack. Until I go fit the stack and load it and all those things, I don't know where I'm going, so I had to wait until this point.

So if I do it, basically I have to wait until this pipeline. All these things are going to stalled. Nothing happens in here. This is first stage. You can have a interested pipeline. You are waiting 20 stages, just stalled, doing nothing. And then there's superscalar plus 20. So that means 20 cycles without the [? 6 issues ?] you can do. So 6 plus 20 is 120 possible instructions wasted, and it's a lot of waste. So you don't want to do that.

So what you do is, you build a predictor to figure out which direction the brand is going. And depending on what the predictor do, and the neat thing is, these days, these predictions are really, really good. They can predict up to 99 points plus [UNINTELLIGIBLE] where you're going. So it's like me telling you, you guys are all going to sleep afterwards, and this is going to sleep on me.

So what you can predict, I will say, aha! I tell you, you are going in that direction. This is like going to [UNINTELLIGIBLE]. I think you're going that direction, and then you just go there. And 99% of the time, you're OK, and when [UNINTELLIGIBLE] finally decided, you can say, oh, that was slight. If you're wrong, you squash everything and restart.

And so modern predictors are this very complicated beast. It looks at things like what happened previously. It looks at [UNINTELLIGIBLE] call stack where are the call stack, what went into the call stack. And then when you're returning, you can predict where we might be returning. A lot of different things go in there, and in fact, we see the core microarchitecture that came before Nehalem's predictor, and Nehalem [UNINTELLIGIBLE], oh, it even has more things, and so it's even more different.

So these guys are doing these complex things. What that means is, sometimes it affects program in a way that you think, oh, huh, this is something that I don't know what might happen. [UNINTELLIGIBLE] this condition, I might not know what might happen. But model prediction sometimes surprises you. Aha, I know there's a pattern, I recognize that pattern, and I go about that. And every architecture, I try different kinds of patterns, and I find the architectures get better and better.

So for example, at some point, if you go odd-even, odd-even branches, if you assume I have a loop, I say, if i is odd, going one direction, otherwise not. Pentium 3 type thing, OK. It just screw up [UNINTELLIGIBLE] with every time it's doing something different. I don't know what's going on. [UNINTELLIGIBLE] Pentium I'd probably say aha, there's a pattern, odd-even, odd-even, it figure out. I core 2 figures out things like every third, every fourth type pattern it figures out. And then things get better and better. There's very complicated patterns you do, these guys manage to figure out. Which is kind of fun, to see what it can figure out, what it can't. By writing a very small program, you can get that. I show some of that.

So the next thing that you have to worry about is the memory system. So the memory system, if you want to be the computer, what you want to do is have a lot of very fast memory very close to you. You can do that. It's going to be very expensive. Sometimes through SQL you can't even attain that, because by the time you build a lot of things, it's actually already far away from you, because we are running very fast here.

So the way you deal with that is building caches. That means you keep a small amount of things close to you in there, and you put things in that cache in a way that hopefully, those are the things you will be using anyways. So it works very nicely.

So the reason that can be done is in programming, when you run a program, there are two types of behaviors that it [UNINTELLIGIBLE] to take advantage of. One thing is called temporal locality. What that means is, most programs, if you use you some data, there's a good chance you are going to use the same data very soon. Because normally, there's a thing called a working set. I am working with a certain set of data. I'm basically touching these things again and again. So if I use that, I'm going to use that data again. So that means I want to keep the data that I used close by.

Other one is called special locality. That means, if I use some data, I have a very good chance that I might be using a neighboring data item. Because if I am accessing a structure or an array, if I'm accessing something, I might be accessing a [UNINTELLIGIBLE] neighbor thing next. So there's what's called partial locality. And taking advantage of these two, you can build this very fast memory system that feels like you have a huge amount of memory very close to you.

Of course, the opposite is true. If your program doesn't behave like that, it looks like you have lots of memory very far from you, and the program becomes very, very slow. Because memories didn't speed up as fast as processors. So what that means is, every generation, the memory [UNINTELLIGIBLE] further and further away, slower and slower and slower. And this is how we manage to keep the machines running fast.

So if you look at what's going on every level, I just gave you a diagram. I won't go through detail. You can look at it later. There are caches, and higher levels you can only keep very small. And one like [UNINTELLIGIBLE], you can only keep a couple of hundred bytes in the registers. But very fast exercise. Then you go to cache. I see on-chip stuff. We can keep a little bit more, access time go down, and then finally when you go to main memory, you can keep a huge amount, slow access. And even at the end, something like tape, you can keep a huge amount of stuff. Almost infinite amount of stuff. But it takes hours to get the tape access is not [UNINTELLIGIBLE]. So there's this hierarchy, and there are a certain amount things you want. You are using [UNINTELLIGIBLE] very fast. You don't want to put it in tape. But you know something that you might access in a year, that's probably a good place to put it. So there's this hierarchy there.

And then you have cache, what happens is, you want to give the illusion everything is there, but when you go to access it, the data might not be there. There are many reasons it might not be there, so let's go through some of the reasons.

So one thing is called cold miss. That means it's the first time I've seen that data. So it cannot be in the cache. If it's the first time I was asking the disk, it's probably sitting very back in main memory, or somewhere in the disk, and I have to go get it. One way to get around this problem is prefetching. So if I keep accessing this I say, aha. You're accessing this in this pattern. And there's a good probability the next pattern is here, and OK, go get it very fast, because you're coming in that sort of direction. So by doing prefetching, you might be able to get rid of [UNINTELLIGIBLE] is happening.

Then there's a thing called capacity miss. What that means is, you're accessing a lot of data. At some point, you have accessed enough data, everything you accessed cannot be fit in the cache, and you have to throw out some of the data to put the next one. So most caches use the a policy called least recently used. That means the data that you unpacked for the longest time gets thrown out of the cache. But the problem is, at some point, you're going to come back to it. And if you come back to it after a long time, that thing has been out of that level of cache. This happened at every level. So what that means is, you have a thing called a working set, that you keep working again and using. If the working set is a little bit larger than the cache, when you come back to that, the data is gone. It's not in the cache. So you want to kind of create a working set that fits in the cache nicely, at ever level, basically. You can do something like pre-fetching and gets around it, but if you can avoid [UNINTELLIGIBLE], that's really nice.

Then there's a thing called a conflict miss. One way of cache work-- it doesn't let you store everything in every location. Sometimes there are some locations in cache [UNINTELLIGIBLE] associated with it. Did you get [UNINTELLIGIBLE] associated in 004?? Associativity? OK. Again, I will talk about cache associated [UNINTELLIGIBLE], but I'm going to talk about memory system later and [UNINTELLIGIBLE] there.

And what that means is, this is a really bad behavior, because you are only accessing a small amount of things, but all of the data, all the aggregate can't fit into the cache. The pattern makes it only fit into a small part of the cache, so you had to throw the data out. I'll get back to this. I'm just going to put it there.

And then in a multiprocess system, [UNINTELLIGIBLE] multicourse, there's a thing called true sharing. That means I [UNINTELLIGIBLE] data [UNINTELLIGIBLE], and the next time [UNINTELLIGIBLE] want to trust the data. So I have to give the data to you, so you're getting cache. And then when I want to use my data, it's [UNINTELLIGIBLE]. I have to get it back to me. So I'm kind of using data back and forth, back and forth. So that's called true sharing.

And a more hideous form of that is called false sharing. That means most of the data is in a cache line. When you move data you move the cache line. So what I'm doing, is I'm using my data myself, and you are using some other data, but unfortunately, they're in the same cache line. So when I say, OK, I want my data, I get the entire cache line, including your data, and then you want to use your data. You say, oops, it's not with me. And I then that means you [UNINTELLIGIBLE] go to you, and at that point, I don't have my data in my cache. So this data is going back and forth. Even though I never touch your data, we have two separate data, but it's going back. I will get this things in a lot more detail in later lectures, so this is kind of giving [INAUDIBLE]. So there are all of these different ways that the data can be not in the cache.

So here is what the processor you are working with looks like. There are 6 cores, and there are L1 separate instruction and data caches, and then there's L2 unified cache, [UNINTELLIGIBLE] L3 cache or [UNINTELLIGIBLE], and then there's main memory. So it's even this deep, deep, deep cache hierarchy going on then. And then right-hand side, I kind of showed the difference. And the interesting thing to realize is, the L1 cache delays about 4 nanoseconds. It gets a little bit, 2 1/2 times slower when you go to L2 cache, and [? 12 ?] times slower when you go to L3 cache, and even more slow when you go to main memory. So every time you go down, it gets slower and slower and slower. That's basically the gist of it in here. Question?

AUDIENCE: Each core has two L1 cache for instructions and [INAUDIBLE]?

SAMAN AMARASINGHE: And instruction data. So instruction goes one direction, data goes in one direction.

So next I want to talk a little bit about profiling. So predictor who is going to be-- the first part of the predictor is all about profiling. Profiling is very important, because you run a program. It doesn't do well. So how do you know what's going on? First of all, you do know, even if it's not doing well. And if it's not doing well, what's the reason? So just having one number, that then saying it's [UNINTELLIGIBLE] in 10 minutes, [UNINTELLIGIBLE] 5 minutes, doesn't tell you too much. You know what to look if you have a large program. The profiling means going, looking at what the program is doing to get an understanding in there.

So what you want to do was collect this performance data while running the application, and then if you're a large application, you want to organize and display data in that a variety of ways, and a lot of times, relate that data to source code, and see what that means in the [UNINTELLIGIBLE] code. And hopefully by looking at this one, you can identify, ha, there's a problem here.

So there are a bunch of tools. Intel Vtune, gprof, oprofile, perf. So you guys are going to use mainly perf, and we will probably talk a little bit about gprof. And next time when you come, next lecture we'll talk about this a lot more.

So profiling is mainly to find where in an application or a system there is a significant amount of activity. And when the significant amount of activity you want to know whether those activity can be avoided, or there's a problem here. So there are some [UNINTELLIGIBLE] significant to what's there, it could be anywhere. It might be addressed in the memory, something might be happening. It might be in the [UNINTELLIGIBLE] system, some kind of process in the operating system might be happening in one thread, it might be happening in an executable file or a module. If you know the symbol of something, you can say, oh, huh, that mode, that actually is this function, and if you know even more information about the program, you can say, aha, that's false, and in fact it's this line of the program. So you get this information from the application when you compile that you can say debug information, say aha, this happens in this line of the program that's having this problem. So you can break it down and get that info like that.

[PHONE RINGING]

Somebody's trying to call me. Hold on. OK.

Secondly, we care about significant activity. If the activity just happened only a few times, happened only nanoseconds of execution time, who cares? You don't want to do that. You want to focus on things that matter. And the key thing about this [UNINTELLIGIBLE]. If you spend most of your time in insignificant things, you would get insignificant performance improvement. If you found the significant part and work on that, you can get significant gains. So the key thing is to find the significant ones.

[PHONE RINGING]

Excuse me. OK.

And the final activity. So activity means time is spent in doing something, and some activities are bad ones. Like if you actually do running instruction, that's good. If your running is useful instruction, a long time, you're OK. But you might doing actually [UNINTELLIGIBLE], [UNINTELLIGIBLE] [? misprediction ?], stuff like that, that's bad activity. You [UNINTELLIGIBLE] say, aha. I am spending a lot of time doing something that I can avoid, and how do I avoid that? So that's what you want to look at.

And you have two ways of doing that, and I want to give you an analogy to figure out what it is. So assume you are going, visiting a bunch of different places. You're on a city tour. You are visiting different parts of the city. And I want to figure out, where do you spend most of your time? And that's a hard problem, because I am sitting in my office, and say, OK, you're going [UNINTELLIGIBLE] the city. I want to figure out where you spend most of the time.

I have two ways of doing that. I'm a busy person. What I can do is every 30 minutes, I can call you and say, where are you? And you say, OK, I'm in this library, I'm in this-- and then at that point, I can have a histogram and say, he's still in the library. And I can call again, where are you? And I can log that. And at some point, at the end of the day, I'll have a histogram saying, every time I call, you found something. If you spend a lot of time in the science museum, I will have a bunch of [UNINTELLIGIBLE]. We'll see [UNINTELLIGIBLE] become then, and if you are not [UNINTELLIGIBLE], if you have only one [UNINTELLIGIBLE] find out, OK, you're not spending time there. That's one way to do that.

Let's go [UNINTELLIGIBLE] saying, for every landmark, I create a telephone booth. And every time you enter a landmark, you [UNINTELLIGIBLE] the telephone booth, you've got to call an operator and say what time it is, so [UNINTELLIGIBLE]. And then you call me and say, aha, I'm entering this landmark, the time now is 5:50, and I write it down. And every time you exit the landmark, you call me and say, I'm exiting this landmark. Time now is-- I write it down. OK? These are kind of two ways of doing that. That's like an instrumentation solution.

So the sampling collector-based periodically interupt the processor and look at where you are, and depending on where you are, you can mark that off. And it's called time-based sampling means every time, every 100 milliseconds or something, in [UNINTELLIGIBLE] processing, you stop and say where you are. Event-based sampling means you are counting number of events like cache meters. Every hundred cache meters, you stop and say, OK, where has this happened? Of course, if the cache misses happen in a regular pattern, you might be in trouble, because every hundred, you might be the same place, then you would get a skewed number. But most probably, there's a statistical variation in there. You can get [? account index ?] and figure out where these things happen. So if you're looking at where all the cache [UNINTELLIGIBLE] is happening, you don't look at every miss, because there are millions of miss. Every 10,000 miss, you figure out where you are, and statistically, one is missing many times, that will adapt in that column.

And nice thing about that is, there's nothing you need to do. Now installation, no changes to application you need to do. [UNINTELLIGIBLE] here, you know how to go in changes, phone booth everywhere. Wide coverage. You can cover everything. So that means, assume you install phone booth in all the fixed landmarks, but there's a service [UNINTELLIGIBLE] down. I didn't see. I don't have a phone booth there. But here, if I'm calling you, I know I'm [UNINTELLIGIBLE] you, anyplace you are, I will cover, even though I haven't anticipated that point. Very low overhead, because I can decide to call you every 30 minutes or call you every 1 minute, if I really care or worry about you, and I can control the overhead that I'm looking at, basically.

The problem is its approximate position. Because if you spend 5 hours at MIT and 30 seconds at Harvard, I will never know that you went to Harvard, because I might have not called you while you were there at Harvard. You might think it's boring and come back, and I never knew that you did that. And also, the other thing is, I don't know exactly how many times you went to a place, because I might have called you 10 times, and I found you in the museum of the science 10 times, but you might have gone there 20 times, and I might have missed you under the times I didn't call, or you might have gone there only five times and stayed for a long time, and I might have called multiple times. I don't know that, and I don't know the count of times you are gone, which is hard to know.

So the main thing is there might be things that are not that statistically significant that you might miss. And if you care about that, it's not a good bet. But most of the time we care about the really statistically significant things, so this is a really good method.

The other part is perfect accuracy, because every time you go somewhere, you have to call me, and I know how many times you went to the museum of science. I know exactly the time you enter and exited a [UNINTELLIGIBLE]. I know all of those things [INAUDIBLE].

The problem of that is kind of low granularity. I can't put phone booth at every corner, and if you had called me at every corner, it's going to be way too expensive. So I have low granularity, very good information, and also high overhead. Because if you're going in and out a lot to a building, you have to call, you have to stop and call, it's very high overhead. And there's not much of a way for me to control that. And it's also high touch. That means I have to build all that infrastructure, I have to go and modify your application, basically, in something [UNINTELLIGIBLE] compile time application, get modified to basically have all these [UNINTELLIGIBLE] installed into the application.

So you can look a lot of different types of events. So in Intel, if you look at core performance counters, there are hundreds of performance counters. And some of these performance counters have no sense whatsoever. So for example, Intel has this counter called number of bogus branches. What's a bogus branch? I mean, why should Intel be doing anything bogus? It's some [UNINTELLIGIBLE] came out. But there are some counters that are useful, so you're sort of getting [UNINTELLIGIBLE] by thousands of things available, we focus on things that we care about. Things like branch events, load store events, cache meter, prefetchers. Those are the important things, and some multicore events that we can look at, and get some interesting information.

And a lot of times, just by looking at numbers doesn't make too much sense. You know you had $5 billion, $300 million branches missed. Is it a good high or low? You have no idea. The right thing is, OK. [UNINTELLIGIBLE] to number of instruction executed how much it would be. The most important numbers are ratios. So from the branches executed, how many were missed? That's a lot more important than, you have 5 billion branch misses. That doesn't say anything to me. So most of the time, you have to figure out the right ratios, and what makes sense to look at those ratios.

So now what I want to do is to go through a couple of examples and show what I can see in these different program behaviors. Any questions so far? Next lecture we are going to go through hands-on examples, going through some of these profiling tools.

So it's kind of fun-- what I did was finally discover what architecture is doing, and what is the modern multicore is capable of doing. In fact, some of these examples, so this set of examples were done on a Core Two. So some of the numbers might be very different in [UNINTELLIGIBLE], because the architecture might do better in some special prediction stuff. I don't know. It might be fun to move it it again.

So what I have is a program that I first created two interesting arrays to access this array. One array has numbers 1 to n, MAXA, is stored in this array. This has 1 to MAXA in random order stored in this array. So when I use this as a way to access the main array, and if I use this in [UNINTELLIGIBLE] 0 to n, I will actually access A [UNINTELLIGIBLE] 0. It's almost like saying AI. But I use this here. But if I use this one, I'm doing random [UNINTELLIGIBLE].

Well, the first program I'm doing is just nothing but going through this one. I will have autoloop that will go through many, many times, so I can actually [UNINTELLIGIBLE] time. So I just go through-- just nothing but trying to just go through that. Second thing I did, I just put a condition. Say, j is than MAXA half. That means halfway through the program, I will go and update this one. Another half, I'm not doing anything.

Then I said, OK, look. I divide by every fourth [UNINTELLIGIBLE]. So what I is j and with 03 in there. That means 3 means [? 2b ?] in the 1 1. So we study 0. That means last two [UNINTELLIGIBLE] has to be 0. To form in delta every fourth element, I will update. Otherwise I won't.

And the next thing I did was I updated-- OK. Let me ask you a question. So this program, the output-- that means what happened to A-- the output of A is equivalent to some other program, one of these three programs. Which one is the [? outward ?] equivalent? See what you can make up.

So first three programs or each will have separate different outputs. A at A would be different if you run it. Except the fourth program will produce the exact same results as some other program, on the first three programs. Which one?

Wake up, wake up, wake up! I hear something. I hear some-- second one. Somebody said the second one. How many people agree with that? How many people disagree? OK. This is not a statistical example. It can be exact. Why say the second one? Second one is right. So what that means is, in fact, instead of a j, I'm using inc j. Inc j is exactly j, because it goes, inc 0 is 0, inc 1 is 1, inc 2 is 2. I just created the same [UNINTELLIGIBLE] array in here. This is something to get that.

And finally, I am doing the [UNINTELLIGIBLE] A basically from x. So what that means, I'm using the same amount of data will get updated, but it's very different order in there.

So now, before I go to the next slide-- question?

AUDIENCE: Maybe you said this, but what is the A array?

SAMAN AMARASINGHE: Yeah, it's some numbers. Some random set of numbers. I don't care about that. I just updated this one.

So in here, which program do we think run the faster? So let me ask you this-- which program updated the least amount of updates for A? Third one, yeah, because it's only doing every fourth element. Which did the most? First one is doing every element, and everybody else will do probably half of the updates. So knowing that, which program will run faster? First one is updating everything. [UNINTELLIGIBLE], OK. [UNINTELLIGIBLE] plus 1, OK.

Which program will run slow? Last one. You guys are onto something. You have seen my slide, you actually know what the hell is going on.

So what I show here is, you can't read this one, but I [UNINTELLIGIBLE] on time, and then I create a ratio. So if you look at the ratio-- so aha. The first program, I always normalize to that, runs 1, and that runs the fastest, if you tried, and the last program ran the slowest. And the interesting thing is, the [UNINTELLIGIBLE], the other, the two, the second and fourth one, that wrote about half [UNINTELLIGIBLE], basically ran all of the same. But the one that wrote every fourth element was actually slower than others. Why is it going on?

So what you can do is, first look at how many instructions would it add up, and how many instruction were executed? So if you look at that, what you see is, the first one executes the least amount of instructions, because it doesn't have to do all this [UNINTELLIGIBLE] when it's updating. The interesting thing is, the one that wrote half has the most amount of instructions. One that wrote only 1/4 has middle amount of instructions. That kind of makes sense in here. But this still doesn't explain the slowdown. That means if this is the only case, the divide by 4 should run faster than the other three. That doesn't make sense. And then what you look at, look at [UNINTELLIGIBLE] branches in there, and if you look at the branches, what you see is--

Let me skip this one, actually. I want to get this one. This is called clocks per instruction. What this is is, if the instructions ran slow, these instructions will take a lot more clock cycles. That means the instructions are not doing well. That means they are stalling, they are taking a lot more time. Normally, we should be able to run [UNINTELLIGIBLE] about four instruction [UNINTELLIGIBLE] cycles should get a CPA of 0.25, but that's very rare, and if you look at that, if I normalize, I get 1 in here normalized. And then what you see is second and fourth test, almost the same CPI, and the third has worse, and the last one is even worse. So why is that? Why are those instructions doing bad?

And then you can go look at that misprediction instruction, and this is exactly what happens. Divide by 4. This is a condition. The branch predictor couldn't predict it that well, because [UNINTELLIGIBLE] predicted in the first three, and it's [UNINTELLIGIBLE] predicted, OK, after three, next one is also going 1 direction, oops, it's in another direction. OK, going first three predictions, going one direction, knowing that information. So divide by 4, looks really bad, then predict 3. And then the random one was just completely off. Random one, basically predictions of three

[INAUDIBLE]. That's why this is.

So in fact, if you just want to calculate the run time, if you think every misprediction cost you 21 instructions, you kind of get the exact total first. You can't help it. So after you meet the instruction, take one cycle, and everybody misprediction instruction, take 21 cycles, and you just calculate that. And this [UNINTELLIGIBLE], I just came, took it out of a hat, and then voila, I get a number that's very close to run time. So what that means is, I can kind of understand what the behavior, this explains what's going on. So I can have a model in my head for what's going on. So it's all about misprediction in here.

I did two other experiments in here. I have the slide sitting here, but I don't think I have too much time to go. I want to go to the numbers in there. Are you ready, [UNINTELLIGIBLE]? OK. Why don't you go put your laptop in. You have the dongle?

AUDIENCE: [INAUDIBLE]

SAMAN AMARASINGHE: OK. So voila. So something you guys waited all this time, and were up all these days developing. So let's see how everybody did. OK, without further ado. I haven't even seen these numbers. OK, what happened? He's trying to-- OK, good. This is [UNINTELLIGIBLE] screen, OK. Do you have it? OK.

AUDIENCE: OK. Something to keep in mind is, we haven't actually investigated what's causing people to have build failures or crashes yet. So it could be your fault, it could be ours. We'll figure that out.

SAMAN AMARASINGHE: So if smaller, better or higher, better, what's--

AUDIENCE: [INAUDIBLE] better. This is run time. That

SAMAN AMARASINGHE: This is really good! This is actually--

AUDIENCE: [INAUDIBLE] The baseline is the flat code that you see in the middle.

SAMAN AMARASINGHE: So this is really good. That means there were kind of two groups. Most of the people got everything right, and they're now the bottom. And there's another group that missed something, and it's in the second camp.

AUDIENCE: I think some of these projects are from people who dropped the class at some point. So there's probably [INAUDIBLE] might not be actually representative of the class.

SAMAN AMARASINGHE: So last year, we got like almost exponential curve, actually. Last year, what happened was there were a couple of people at the bottom, and everybody worked at the top, basically. Which is actually good. So you guys actually figured out what's going on.

AUDIENCE: So next one is even more interesting.

SAMAN AMARASINGHE: Wow!

AUDIENCE: So this is [INAUDIBLE] that we gave you guys. Most of you optimized it down to 0 seconds. So I coded in a harder test case, and here's the results on that. Still pretty good.

[INAUDIBLE] There's some very specific optimizations [INAUDIBLE].

Yeah.

SAMAN AMARASINGHE: So exactly. There are people who are climbing up, I don't know exactly what, but it's very clear that [UNINTELLIGIBLE], probably data representation tag that people missed in here. Which is really now interesting. Yes.

AUDIENCE: And for Pentominos, this is very incomplete, because the set of tests that we wanted to run are still currently running. So I just picked a random test case, just to give everybody an idea of how everybody did, and set the time out very low so that I could actually finish it during this lecture.

And what's [INAUDIBLE]?

Yes. Somebody solved it instantaneously, and it searches for the first 5,000 solutions to some random puzzle that I pulled out.

SAMAN AMARASINGHE: So [UNINTELLIGIBLE]?

AUDIENCE: That I haven't verified yet.

SAMAN AMARASINGHE: Aha. So it might be, there might be just the scratchings, so we don't know that, and then that could be correct answer, but--

AUDIENCE: Which one is the baseline?

The baseline is actually the 90-second mark.

[INAUDIBLE]

SAMAN AMARASINGHE: [UNINTELLIGIBLE] the baseline?

AUDIENCE: No. 10 seconds and above are either timeout or [INAUDIBLE].

SAMAN AMARASINGHE: Oh, they're probably mostly timeouts.

AUDIENCE: [INAUDIBLE]

SAMAN AMARASINGHE: Yes.

So there you are. This one, people would have some work to do, I guess, to get the performance down. But this is pretty good. This is not like [UNINTELLIGIBLE]. I mean, last couple of years, we had people, [UNINTELLIGIBLE] someone who was 1,000 [UNINTELLIGIBLE], you had to plot it in a log thing. So in fact, if I do a square root within the grid, this is pretty nice. So [UNINTELLIGIBLE] is not that bad.

So this is great. This is where you get [UNINTELLIGIBLE] performance wise And if you are in the bottom, you can go have a beer. Otherwise, you might have to go back and figure out what you missed.

AUDIENCE: [INAUDIBLE]

SAMAN AMARASINGHE: Oh, if you're [UNINTELLIGIBLE]. Yes, exactly yes. I should say that.