PROFESSOR: OK, so by this time, most of you should have had your meetings with your masters. How many of you had meetings with your masters? Who didn't yet?

AUDIENCE: Today.

PROFESSOR: Today? And Friday. Who hasn't scheduled a meeting? So are you talking to them, and do you know--

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK, make sure you get masters meetings scheduled because this is a very important part. We are getting feedback from the masters. I think they are giving you guys a lot of good advice, and so make sure you schedule, you go, you meet, get that result, use that result. It's very hard to arrange that kind of a senior person spending a lot of time with only two of you at a given time. That's a really good ratio to have, so take advantage of that.

OK, so also, I think at this point, most of the beta issues for project one, we have been probably worked with, and they have some issues some people looked at that they are performance code. They were not happy because the thing is this. You can always run everything instantaneously if it doesn't have to produce a correct answer. You can just return, and so part of performance is also it has to have some correctness. So certain amount of correctness has to be passed before we can give performer grade.

So some people have this question. Why are you giving me a correctness grade because it seemed to run? But if you haven't done the right checks and stuff like that, then it's unfair for other people because there's no way people who do the

correct implementation can match that performance because we're using that as upper bound. So make sure you do that.

And another issue people had was when you go to masters, you're going to work with one or two other students. And in your group, you're going to share all the material with your group member and nobody else. But between the beta and final, you get opportunity to basically get input from your masters, and also see what the other people who are coming to a project in your discussion have done. So it's OK for you to see those code.

You can't take the code home. You can't just say give me a print-out, I am going to take home and re-implement. That doesn't count, but if you see some clever things they have done, it's all here to learn.

Class is all the mostly about learning. However we have to grade you, so it's not all about grading, it's about learning. So we are trying to get a balance where we opportunity for you to learn from your peers. So if you go look at other people's code, if you find something cool they have done, learn.

And also if you [UNINTELLIGIBLE] for you [UNINTELLIGIBLE] say, look, I did something cool. It's a good learning experience in there, so that's one opportunity you have to talk to other people on other groups, to basically learn something or find some interesting tidbits of how to get good performance. And hopefully, you learn some interesting things and able to implement that in your final. OK?

So today we are going to talk about memory systems and performance engineering. So if you look at basic idea of what memory systems, you want to build a computer that seemed to have a really fast memory. You don't want things to be slow. For example, a long time ago, people build these computers where memory's always slow, everything [UNINTELLIGIBLE] further away. It doesn't help anybody.

The way we know how to do that is to build small amount of memory very close to the processor. You can't build a huge amount of things all close to you. That doesn't work, doesn't scale like that. And we made a cache just like that, and you build

larger and larger memory [UNINTELLIGIBLE] going down, and the illusion is you want to give the illusion of you have huge amount of memory.

Everybody's very close to you. OK, so it looks like you have millions of friends, that you talk to everybody, and that doesn't happen. But how do you give that illusion? And the way that you give that illusion is when you use normal programming practice, when you normally using data, people have found there are two properties.

The first thing is called temporal locality. That means if I use some data item, there's a good chance I use that data item again very soon. There is something I haven't used for a long time, probably has a good chance I will not use it for a very, very long time. So can I take advantage of temporal locality? So that mean some data will be get for use a lot of time, in a very quick [UNINTELLIGIBLE], so those things should be very near to you because you might.

Other one is spatial locality. That means if I use some data item, there's a good chance you'll be using data items next to that, closer to that. So can we take advantage of that? So those two properties help you help the compiler to fool the system, fool everybody, thinking that it has this huge amount of memory very close to you, but internally doesn't. Unfortunately, since it's trying to fool you, it doesn't work all the time, and when it doesn't work, it's good to recognize and able to fix those things.

So I showed this picture sometime ago, too. Memories in a big hierarchy. L1, L2, in fact, we have L3 cache and a memory, disk, tapes, it can go up and down. And the key thing is when you go here, you're starting from registers, you have very small amount of things, very fast access. Here, we have almost infinite amount of things, very slow access.

And the two reasons why this has to be smart because first of all, those things are very expensive. And second of course, you can't [UNINTELLIGIBLE] too much, and when you go down, things get somewhat cheaper. So that was one of the kind of economic incentive here.

So I talked about cache issues, and I'm going to go through this again because we are really going to dig deep into these things. So when you have a cache, that means you have a small memory close to you that's basically shadowing something sitting behind you. So when you ask for the data in the cache, there are many reasons why it's not there.

First thing is cold miss. What that means is you're asking for something that you have never asked before. It's the first time you're asking, so the data is probably sitting way behind in your memory where, [UNINTELLIGIBLE] in a disk. We haven't even loaded it in there. And the first time you have to get it, so that get pulled in, and so this seems to be a place where there's nothing you can do.

But another part of locality is a thing called prefetching. These modern processors have a crystal ball. They carry most of the things that branch prediction and stuff is trying to predict what you want to do. You can do the same thing in memory. You can look at what you have done.

And Intel has this huge amount of circuitry that they don't tell anybody what they do, but they're trying to predict what you might fetch next, what data you might look for next. And if it is working well, something you had never, ever looked before, it might deduce that you might need it and go get it for you. So if that works, great.

So then there's thing called capacity miss. What that means is as I said, I am trying to keep my best friends, or the people I will be working with a lot-- Oh, nice. What's going on here? Please start later. OK. --the people I'm going to work a lot with, close to me.

The problem is I can have only certain number of friends, and if you have more than that, I can't keep everybody. So hopefully, your program is going to use a lot what we call the working set that can fit in the cache. And at that point, you get all of those people, like I said, get everybody into one room, and the problem is if your party spills over from the room, then there's a lot of issues. But if everybody fits in the room, things are very nice. You can have a good interaction, continuous

interaction, with them.

And what happens is, if not, the worst case is the caches have eviction policy, normally is called least recently used. That means if I don't have room in the cache, I figure out the cache line that I haven't touched for longest, and I get rid of that, bring the next one. So that's a good policy. It fits in your locality thing, but when will did not work? [UNINTELLIGIBLE] going to figure out when least recently used will create a really bad problem in capacity.

AUDIENCE: [INAUDIBLE] just one. Say you're [INAUDIBLE] set, and that set is, by one, bigger than the [INAUDIBLE] missing.

PROFESSOR: Very good. So what you're going through, you're going round some data again and again, and that amount of data is one bigger than your cache line. You are going to have no locality because [UNINTELLIGIBLE], and when you go to, just before you use that data, they said, oops, I need to bring one more data. I need to evict something, and who am I going to evict? The guy I'm going to use next because it was the least recently used thing.

So it goes, and then you go there, and you bring that. Who's it going to evict? You're going to evict the one I just brought. You're going to evict the next one that I am just about to use and bring that one.

And you go use that, and then to use the next one, you're going evict the one you're just about to use. So by going and doing that, you basically get no locality. So that's a really bad problem in here.

So another interesting thing is called a conflict misses. Conflict misses says normally, when you have a cache-- so what we have is you have this large storage, which [UNINTELLIGIBLE] the memory, you're going to map into smaller storage in here, which is cache. So one way to say is any of this value line can be anywhere here. That's called fully associative cache.

Implementing that is somewhat complicated, so the other extreme is called direct map cache. What that means is this segment, the same size here, can map into

here, and then the next segment also can map into here. That means if you get this cache line here, it can be only mapped into one place in this here.

This cache line can only be here, and also this cache line can also only be here if it is the same offset. So that means for every cache line, there's only one location in the cache. So here, what would be really sad scenario?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, I mean I assume I have a lot of these slots. I read something here, and the next time I read something here, next time I read something here, and I go round and round this way. I might be touching only very few number of data items, but they also mapping into one cache line even though my working set is very small than the cache. I still don't have any cache usage because I am having conflicts.

And then there are two other misses when you're going to multi-process. We'll go through them in later lectures. One is called true sharing. That means there are caches private to each core, and what happens is if one core uses some data, you've have to get the cache line there. If the other core wants to use that data, you have to bring the cache line back there, and the cache lines can ping-pong [UNINTELLIGIBLE] that.

False sharing is even a worst way. So what happens is if the first processor here touches this value of the cache line, second processor touches this value of the cache line. We're not sharing anything, but unfortunately, the two things I'm using sits next to each other, so when I ride this thing, I get the cache line, he doesn't. When he need to ride this thing, he has to get the cache line. So I am seemingly using independent data, but I am bouncing cache line in between, and if that happened, that going to have a huge big performance impact.

So these other things. The last two, we'll get to a little bit later. So today I am going to start with modeling how caches work with a very simplistic cache. So here's, assume, my cache. I have 32 kilobytes in here.

This is a direct map cache. That means that in my memory, 32 kilobyte chucks get

mapped to direct [UNINTELLIGIBLE]. This 32 kilobyte get mapped here. That 32 kilobyte get mapped here.

And the cache line inside is 64 bytes, 64, and that means I have basically [UNINTELLIGIBLE PHRASE]. 32 kilobyte in here. OK? So just remember this as we go on doing this, we will use this as a formula. OK?

And we assume if you getting the cache, [UNINTELLIGIBLE] single cycle, if you miss 100 cycles. OK, so it's nice numbers to do that. And so the first thing is we have a reference like this. OK?

You go from [i] equals 0 to [i] less than very large number, A[i]. I just go to accessing memory like one after another after another after another. So you see how this is going on in here?

So also I am accessing integers, so that means four bytes at a time in cache line. So what happens here? So I assume size of int is four, so you're getting four bytes in here. So yeah, you're doing s reads to A. I'm accessing s elements in here.

And you have 16 elements of a per cache line. OK, you see why 16 if there? Because I have 64 in here, 64 bytes in here. Each time axises four bytes, so I have 16 of integers in cache line.

AUDIENCE: [INAUDIBLE] bits or bytes?

PROFESSOR: Bytes. Byte. Did I say bits? No, bytes. So what happens is if it axises the same element, 15 of every 16 is in the cache because when we are going one after the first guy, cache miss, I had to go bring it, and the next 15, I have brought to it. It's in the cache.

So what should be my cost? [UNINTELLIGIBLE] cost of memory access. I am accessing s data, and 1/16th of that is a cache miss, 15/16th is a cache hit. So basically total [UNINTELLIGIBLE] is a 15/16th of s is a cache hit, and other 1/16th I have 100 times cycles I need because it's a cache miss. Everybody good so far?

OK, so what type of locality do we have here? First of all, do we have locality? Who thinks we have locality? OK, lot of people thinks we have locality. What type?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Spatial locality because what we are doing is we are accessing the nearby elements even though we are not getting back any of the data yet, so we get some spacial locality axis here. That's good, so this is our very -- most simple thing we can do. So what kind of misses are we going to have in the cache? Cold misses. I'm missing because I have never seen that data before, so it's a cold miss in here.

OK, so that's that. Let's look at this one. I'm accessing A[0] s times. OK? What's the total access time?

How many cache misses am I going to get? One, so I basically have 100 time for the first cache miss, and s minus 1 for the rest basically it's a hit. OK? That's good, so what kind of locality's this?

Spatial or temporal. There's not too many choices. Or none.

How many people think spatial? How many people think temporal? How many people said there's none locality? OK, that's a lot of temporal. OK, you want to get [UNINTELLIGIBLE]. That's temporal locality here because you're accessing to the same thing again and again and again, same data. So this is--

Oh, OK. Want to restart, OK. Wait a little. So this is only time we notice in a hurry. Every time, I go to do something, I get hourglass.

So what kind of misses are we getting? Trick question. I got one cold miss, and that's about it. And the rest I don't have any misses. So if I have a miss, it's a cold miss.

OK, so here I am doing something interesting. So what the heck is in this list? So I'm accessing A[i] [UNINTELLIGIBLE] this 2 to the power number, one shifted to the entire miss, 2 to the power N. I shifted between 4 and 13.

What this 13? Why is 13? Less than 13. What's 2 the power of 13? I think I got this right.

AUDIENCE: 8,192.

PROFESSOR: What's 2 to the power of 13?

AUDIENCE: 8,192.

PROFESSOR: Yeah, 8K. And I have 32K. So 8K of 32 [UNINTELLIGIBLE]. 8K of integers, which is each as 4 bytes is how much? 32K.

So what this says is everything I access should fit into the cache. I am accessing data like that. I'm going back and accessing data like that. I'm going back and accessing data like that, and it should all fit into the cache.

Do you see what I do? I access up to 2 to the power of 18, and I go back and access that again. I'm just going back and back because of the model operation. Everybody see what's going on? So how many cache misses should I have?

AUDIENCE: There are four cache [INAUDIBLE].

PROFESSOR: OK, how many cache lines I would be accessing if I am doing i1 2 to the power N? [i] mod 2 to the power N.

AUDIENCE: [INAUDIBLE]

PROFESSOR: So one miss for each axis line the first time around. Afterward, it's only in the cache. So how many axis lines? 2 to the power N.

Is that right? I think this is wrong. Oh, yeah, this is right because every axis-- So in the cache line, how many--

AUDIENCE: [INAUDIBLE]

PROFESSOR: This is four. So there are 16 entries here. 16 entries in here. OK, only one of them basically misses that, so that means if 2 to the power N axises--

AUDIENCE: [INAUDIBLE]

PROFESSOR: You are doing 2 to the power N axises before you go back in here. So what you're doing is you are-- you make this many axises before you go back again. OK? And how many axises assigned to be in the same cache line? You have 64 bytes in the cache. OK each axis is four. 16.

AUDIENCE: [INAUDIBLE]

PROFESSOR: 16. So that's true [UNINTELLIGIBLE].

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yeah, but what I'm saying is I might not access the [UNINTELLIGIBLE] because if any small. I only access a certain amount of cache line, so I'm accessing a part of the cache. I might not go through the-- can everybody see this, how this is working? Because if I'm only accessing, we'll say, accessing one, if this is, we'll say, 2 to the power of 5.

OK, I am not going to access the entire cache line. [UNINTELLIGIBLE] cache is I'm going to [UNINTELLIGIBLE] go through the cache. So this many cache lines I'm accessing. OK, and the first time I access that, I get a cache miss, and after that, it say everything is in the cache.

Everybody's following me? Or are we like lost in here? How many people are lost? OK, let's do this.

So what happens is if you have a modular 2 to the power N. What that means is I'm going to keep accessing one to somewhere 2 to the power N, and the next one, I am going back here again. That's my axises, basically, because it goes [UNINTELLIGIBLE] accessing to the power of N.

I nicely said n is between 4 and 13. So n is 13 means the maximum I can do is 8K. 8K increase. 8K increase equals 32K kilobytes of memory.

So that means you don't have any option of overwriting the cache. So you go to

cache, [UNINTELLIGIBLE]. You don't wrap around in the cache. Do you see that? OK, you know wrap arounding is so bad, that means all these things is going to fit in the cache, and then when you get [UNINTELLIGIBLE], this is going to be in the cache because the data you access is smaller than the entire cache.

OK, so what that means is only the first line has to have some cache misses. So how many cache misses you are going to have in the first line? Or any of the lines going to have cache misses because the first line still fit in the cache. [UNINTELLIGIBLE] anything. We don't need everything.

How many cache misses have in here? The interesting thing is because these are four bytes, we have a 64-byte cache line, and each is four bytes. So that means I can do 16 axises every time I have a cache miss.

OK, so up to here to here is 2 to the N. My cache misses I do [UNINTELLIGIBLE] basically. My cache misses is basically this much. Everything else is in the cache, and after the first line, everything else is in the cache again, [UNINTELLIGIBLE] nicely, got my working set to fit in the cache. I'm really happy.

Everybody see this? How many people don't see it now? OK, good. So let's move to something a little bit more complicated.

What kind of locality do we have?

AUDIENCE: Temporal and spatial.

PROFESSOR: Good. We have temporal and spatial. You have spatial because every time you go, you only get the first line of the cache. Rest is in the cache. From that, I got a 16x improvement because I have temporal locality.

And since I only access when I go back to the data, after accessing this, since it's already in the cache, I get spatial locality because of the 16. 16 things are in the cache, I am going through that. I get spatial locality. I get temporal locality because next time I access, it's still in the cache. I haven't taken anything out of the cache.

OK, what kind of misses? Should be cold misses again. Cold misses because the

first time I [UNINTELLIGIBLE] things. I haven't seen that data. After I get it, everything is in the cache.

So now, here's interesting case. Now, I am doing 2 to the power N, where N is great than 14. Now, what happens? Now, what happens in our picture here is I am going--

So as you might cache is somewhere here. At this point, I fill out the cache. I still keep going, and minute I access something [UNINTELLIGIBLE] cache, what's going to happen? So I went through accessing 34 kilobyte. If I access the next one, what happens?

OK, no, no, no shutting down, please. OK, what happened? So I am accessing more than I can fit into the cache. Next time access, what's going to happen? Anyone want to take a wild guess?

AUDIENCE: [INAUDIBLE]

PROFESSOR: [UNINTELLIGIBLE PHRASE] before that. [UNINTELLIGIBLE] in there. So what happens is then am I ever going to have any temporal locality? No.

OK, so now, first access to each line basically misses forever because by the time I get back to the [UNINTELLIGIBLE], it's not there. So basically, it's gone out of the cache. So what that means is my total access time is basically every 16th element, I am getting cache miss forever.

So what that means, every 15th of 16, I have a hit. So what I have is 15 out of 16, I have a cache hit. 1/16th of the time, I have a cache miss forever.

So what kind of locality do I have now? I have spatial locality. I don't have any temporal locality. I don't ever go back to something in the cache again. So what I have is spatial locality, so what type of misses now do I have?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Cold, right.

AUDIENCE: Is it like shared misses? [INAUDIBLE]

PROFESSOR: It's not shared. When you fill out the cache, you go back to [UNINTELLIGIBLE]. You fill the cache. So what type of miss is that?

AUDIENCE: Capacity?

PROFESSOR: Capacity miss. OK, so you have basically cold and capacity misses happening now. OK?

AUDIENCE: One question. [INAUDIBLE] that you multiplied whenever you're trying to load from memory, is that like an arbitrary number, or is that the actual cost of going--

PROFESSOR: Every machine, you can get this beautiful table. I will go off what your machines have. We still have a number saying this is your miss [UNINTELLIGIBLE], this is the thing-- I mean, so some of these things you realize because of all of the complexity. It's not that nice and simple, but this, I just pull out of hat, and it's kind of normal. So what do I do, now?

I am doing the mod, but I am multiplying [i] by 16. So now, I am accessing this value, this value, this value, this value in here. I'm accessing this value, this value, this value, this value. One value in each cache line. OK? How much cache misses do you think I'm going to get?

AUDIENCE: [INAUDIBLE]

PROFESSOR: 100 for every access is basically beginning a cache line, you're taking one value and if I go back to that value, I have already filled up the cache. So basically, I have first access in the cache. And what's your total access time? Anyone will take a guess?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yep, 100 times x because I have no-- OK, so this is a [UNINTELLIGIBLE] locality. That was clear. What kind of misses am I getting now?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Now, I'm getting conflict misses. I get a cold miss at the beginning, first time around, and then every time I do something, I'm getting conflict miss because the thing is, now, this N, I am only accessing small amount of data. It doesn't fit in the cache, but I'm not still getting any kind of sharing, so I'm having conflict misses.

[UNINTELLIGIBLE] second time, I guess I will probably jump over this. I just did OK, if I go random access, what happens? So let me jump over that. Actually can do a calculation to figure out how many, probabilistically, how much you might have and stuff like that. OK.

So now, if you look at what's going on, when you have no locality, you'd have no locality. That's a pretty obvious statement. And then if you have spatial locality and no temporal locality, we are streaming data. That means you are going through the data, but you're not getting back to it fast enough, so I have temporal locality. So I stream through data, so that means I go through data in a streaming fashion. OK?

So what we have is if working set fits in cache, you have InCache mode. If working set is too big for cache, you can get some streaming mode and still get some locality because we are bringing the lines in here. And you can do other things like optimizers like prefetching we'll get to. And other issues [UNINTELLIGIBLE] this last one, but to deal with cache axises.

So if you have more than one axis-- so here what I have is too nice arrays. [UNINTELLIGIBLE] is 2 to the power size array 2 to the 19 power arrays. OK, so now what happens when I put this in the cache?

Look what happens in here. So what happens is this array get mapped in this one. This array get maps this nicely in here. So what happens is this line can only be in this cache line, and this line also only can be in this line of the cache. Do you see what's going on, now?

So if you do this one, basically every time you access something-- See, assume I'm

accessing this data item, so I'm doing A, B. I access A, I get this line. I access B, what happens?

AUDIENCE: [INAUDIBLE]

PROFESSOR: So this [UNINTELLIGIBLE] is gone. It is gone. Now, I am accessing the next one in that cache line we made.

AUDIENCE: How do you know what cache line-- or how [INAUDIBLE] cache? [INAUDIBLE]

PROFESSOR: Normally, in language like C, if people two arrays next to each other, they should be mapped next to each other in memory. And if you want to be more adventurous, you can go to the assembly and then see what their locations is. So if you put A, A next to each other, you normally kind of get it next to each other.

So what this one is this is pretty bad. I should have gotten at least some spatial locality. I'm getting nothing. Do you see what's going on here?

Do you know what's going on here? Know why I'm not getting any spatial locality? Because I am bouncing.

When I access this one, this one is gone even though I have more data in the same cache line, so I'm kind of bouncing two cache lines because of that. What's a good solution for this? Back there.

AUDIENCE: [INAUDIBLE] A and B could be mapped to the same cache line.

PROFESSOR: The thing A is a nice 2 to the power. A size is a multiple of the cache size.

AUDIENCE: [INAUDIBLE]

PROFESSOR: The size of A is a multiple of the cache size. So what happens is if you have memory in here, you map something like A, C here. If this is a multiple of cache line, so assume this is 32K times some number, and you map B here, and then this is normally what happens in C. You put all adjacent map allocations next to each other in memory. Do you got A here, and you start having B in here, next to it.

OK, so what happens is now if A starts to assume address, we'll say 100, this is 100 plus 32K times N, basically. And you do this mod 32K, this mod 32K, this same number. It maps into the same line. OK, so I kind of gave you the problem. What can be the solution?

AUDIENCE: [INAUDIBLE] allocate one line.

PROFESSOR: Yes, it's called padding. OK, so I have no locality in here, let me go to the next slide, what kind of misses? I have cold and conflict. What I can do is just basically add a little bit of padding. I did 16 here.

Normally, you do a prime number so it will not clash with anything. Add a little bit of padding at the end of array, and that means the next array will not be conflicted most of the time. So a lot of times when you declare things next to each other, it's always good to add a little bit of a padding in between.

Normally, a smaller prime number would be a good thing to add a padding. Now, I start getting back my nice locality because the two things are mapping the two cache lines. What type of locality do I get here? Anybody want to take any guess what type of locality I have here?

AUDIENCE: Question, sorry. [INAUDIBLE] you actually added just an extra 16.

PROFESSOR: Yes.

AUDIENCE: Why would that just make the A and B [INAUDIBLE]?

PROFESSOR: Because what happens is normally it doesn't make A and B into leaving the cache, but it makes A[0] not exactly matching to B[0]. A[0] will interleave with something like B[16] or something. This thing might map into the same place.

AUDIENCE: I mean, but I [INAUDIBLE].

PROFESSOR: Yes, normally in a computation, people do that, A[i] equals B[i] plus something.

AUDIENCE: Oh, OK, so [INAUDIBLE]. Like I understand what you're saying, but I don't

understand how that [INAUDIBLE] because they're both accessing [i] at the same time. Why would they [INAUDIBLE]?

PROFESSOR: So here's the problem. Assume I have [UNINTELLIGIBLE] 32K. I'm assuming [UNINTELLIGIBLE]. OK, so first one is, we'll say, you start with 100, 100 plus [i], and the other one is 100 plus 32K plus [i].

OK, then you take a mod 32K, and this end up being 100 plus [i] with another 100 plus [i]. So this is basically [UNINTELLIGIBLE]. You still have one or one element in the cache it's all going to map into. You see that? But now, if I add 16 more to this one, then this map into 116 plus [i].

AUDIENCE: I don't see that reflected in the code. You just [INAUDIBLE], so how is B different from A in your code?

PROFESSOR: Because [UNINTELLIGIBLE] my S. I added 16 to S, so my A is a little bit bigger, now.

AUDIENCE: But isn't B also [INAUDIBLE]?

PROFESSOR: Yeah, B's also bigger here. I don't care. B goes down here. B, I add a little bit more to the B. It doesn't matter. I padded B, too.

So what that means is the first A[i] and B[i] are not in the same cache line.

AUDIENCE: How do we allocate A and B right next to each other?

PROFESSOR: So normally in C, if you declare something like int A [100], int B [100], more or less, they will be allocated next to each other. If you look at the assembly listing, it'll say code allocated next to each other in memory. Even in the stack, if you do that, if you allocate things, compiler has no incentive to go and move things around.

AUDIENCE: This is only for global and [INAUDIBLE]?

PROFESSOR: Global variable, local variables.

AUDIENCE: Does any of it still apply if you use malloc?

PROFESSOR: The thing about malloc is--

AUDIENCE: [INAUDIBLE]

PROFESSOR: It might do something [UNINTELLIGIBLE], but on the other hand, if it is fitting on the same page, it might also do something like that, too. So it depends on how we do that. At the beginning, if you keep mallocing things, it might be [UNINTELLIGIBLE]. They might be a little bit off because malloc put some metadata with each status.

If you ask for [UNINTELLIGIBLE], you're not getting exactly 100. You're getting a little bit of a bigger size. But you might have some configuring to do, but after some time, when you have done a lot of [UNINTELLIGIBLE], we have no idea where it's going to go. It's random.

Anymore questions? OK, good, that is a good question. So what kind of locality do I have here? [UNINTELLIGIBLE] two lines, I am accessing each, and I got 16 times 15 out of 16 hits. What's that?

AUDIENCE: Spatial?

PROFESSOR: Spatial locality. And of course, if you have spatial locality, you get cold misses, basically. And I think [UNINTELLIGIBLE], you get what other type of misses?

AUDIENCE: [INAUDIBLE]

PROFESSOR: [UNINTELLIGIBLE] conflict here. Basically, capacity. Capacity miss, exactly.

So in order to avoid this, what we have done is we make cache-- instead of mapping into one place before, and what we said was we had this memory, and assume we have 32 sets. And if you have a cache like this, we make this entire map into this one, and this map into this one. What people have done is instead of doing that, what you can make is make the cache, instead of one big block, two small blocks.

So what that means is this can happen to this one. My drawings are not that great,

but I hope you get the picture. This map into this one, so what that means is each reference here, so maps here, and these two basically conflicts. But now with that [UNINTELLIGIBLE] too, I can put this one here or here. I have two places to put the value, basically, and that [UNINTELLIGIBLE].

OK, so that means if I access something conflicting, [UNINTELLIGIBLE], you cannot just two things. Still, there are places in the cache for them to go. And one way to do people is fully associative cache. It can go anywhere, but that's very hard to do in hardware, but you can do some small number of ways.

So in here, if you do two basic associative cache, in the first axis in here, even though these are conflicted, on can go to one [UNINTELLIGIBLE] set, another one can go to the other set. OK, so two associative cache [UNINTELLIGIBLE]. So total access time, again, I have nice locality in here, spatial locality, and I have both cold and, I guess, capacity misses.

But if you have two associative cache, what's the next problem? How about if you have three different things? So we have two different things like that. OK, now that I only have two associated [UNINTELLIGIBLE], it doesn't work because, in a [UNINTELLIGIBLE], I will replace the last thing, and you might still end up with nothing in here because you are replacing nothing, and the next thing replaced the other thing.

And basically after you gave B, C, then you go to A. A Is not there. When you go to B, B is not there, C is not there. You kind of go back to the same problem in here, and then basically have to access everything again. There is some spatial locality but cache can't use it. Basically, you have conflict misses.

OK, so you see that? So associative is good because normally people assume you don't need that much associativity, but it's limited. So if you have too many things, you might run into this problem again.

AUDIENCE: [INAUDIBLE]

PROFESSOR: You mean capacity?

AUDIENCE: Yeah, [INAUDIBLE] capacity.

PROFESSOR: Less.

AUDIENCE: [INAUDIBLE] elements in a cache before you pick [INAUDIBLE] because it looks as though the matrix is is using half the cache by itself.

PROFESSOR: Because it's associated, you mean? You have multiple there? I mean, the thing is since it's associated, no, because you have enough room. I mean, if you're going [UNINTELLIGIBLE] everything, you have two places to put the name because it maps. That's a good question.

So what happens is now, assume this many things fit into the cache before. Now, cache has two banks, and in each bank, only this many things fit in, but the next thing can go to the other bank. So if you go through [UNINTELLIGIBLE], you have the same behavior even if you have associativity. OK, you can go and-- because if the cache is the same size, the size is what matters for if you are going through [UNINTELLIGIBLE] axis.

AUDIENCE: So far, [INAUDIBLE] machines that we're working on, do they have a fixed kind of cache, or can we decide like--

PROFESSOR: The fixed kind of cache. It's built into the hardware. I will show the table to say what type of associativity is in there.

AUDIENCE: Is there not a way to say [INAUDIBLE] three-way associative? Can you have something on top of it so that it looks like it's two-way associative or something?

PROFESSOR: I mean, you can do things like that in software, but they're very expensive. I mean, think about cache is trying to fool the cache in that direction cannot be that easy because they are like really hard-baked into hardware. that's a good question, but there are things you can do, not exactly modeling something, but kind of putting offsets and stuff like that so it doesn't get into these kind of conflicts.

So yeah, I will answer this thing. Why don't you have more? Because if you go back

to your [UNINTELLIGIBLE] lectures. Perhaps it might have explanation in it's hardware why it's much harder to build.

And if you have the linked lists now-- so assume I have linked lists we are going through in here. And what's my [UNINTELLIGIBLE]? Basically, depending on allocation and using some things, so best case is everything is in cache, so I have a small list. And I go through again and again, round and around my list, everything's in cache and [UNINTELLIGIBLE] S. That's really great.

Next best is everything is adjacent. OK, everything might not be in the cache, but everything at least in adjacent memory. So basically, first time I allocate my malloc did a nice thing for me, and at that time, I basically get a nice basic spatial locality since my size is 16 in here. So I get this much.

And the worst case is random. Malloc puts it all over the place. Question?

AUDIENCE: [INAUDIBLE] cache, does the thing behind it [UNINTELLIGIBLE] in front of it [UNINTELLIGIBLE] also going to? Do those both have a spatial locality?

PROFESSOR: What happens is normally, we assume 64 element cache, so it's at that boundary, so basically 2 to the 8 boundaries is what happens. So what happens is cache line begins at 0 address, 64 address here. So if you get 63, that means this value is not in the cache, so you get this cache line.

So the cache line doesn't shift. It's fixed chunks in here, so that if you get address 0, you get this one, and all these things. If you get something in the middle, you'll get the left and right [UNINTELLIGIBLE], but if you get here, you only get the right. So this basically 0 and 64, they are boundaries [INAUDIBLE]. If you like 65, that's it.

But that said, if you're accessing [UNINTELLIGIBLE], that's this thing called prefetcher that say you are going through linear pattern, and you might need that next, and it'll bring this up for you. And what you find, I'll show later, that in fact, the machine you're using has a really powerful prefetcher, so it can hide a lot of these things. So we'll get to that.

So this is my total access time in here. Random case. Random case is basically, every time I access something, it's not in the cache.

And then if you have a bigger struct in here-- but assume I have this big struct in here, but I am only accessing one element of this struct in here. I do a lot. I created this large struct, and I am [UNINTELLIGIBLE] things in the struct, and I just basically looking at my data. Only one element. What's the problem here?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Exactly, so I bring the entire cache line, but I'm only using a little bit of it. Because I bring all these things into the cache, it comes with axis and that, but I'm only using data. So basically, what happens is, if everything in the cache, I have access time S. That's great. That's not a problem, [UNINTELLIGIBLE] streaming.

Even though I'm getting spatial locality, I only get half of it because even though I'm bringing the cache line, only two have datas because this is 32 bytes long. It's in my cache. Even though I am bringing all this data, I only two of them in the cache because rest of this other data is kind of hanging in there with me, and so I don't get that. And random, of course, you have no locality.

So how can I basically make this better? How can I make this better? OK, I just showed it very fast. OK.

AUDIENCE: [INAUDIBLE]

PROFESSOR: So instead, if I ask you, I have this large data structure, but most of time, I'm only accessing a small part of it, this is kind of a little bit of a hack. I mean, this is kind of going against, I guess 6005 kind of building, but this actually works. What you can do is, you can go back and instead of doing structs, what you can do is you can do arrays of each data item. And now what happens is that I am basically bringing these, and then the data I am accessing is basically right now next to each other.

So you might have this very large data structure representing something you have a lot of, then. But most of the time, you're only using few of them, so you don't have to

be structs of arrays. You might have couple of different data structures.

One is the one that you access regularly hopefully you get them next to each other. And the nice thing about arrays is it guarantees things are next to each other. Of course, was managing it is harder.

AUDIENCE: [UNINTELLIGIBLE]

PROFESSOR: You can do both. So that means instead of doing the structs, you use arrays of each field.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Instead of. So instead of doing this one, you just do this one.

AUDIENCE: [INAUDIBLE] instead of being able to pass stuff over. Now you have to kind of be able to know [UNINTELLIGIBLE], so what that is where because you cannot just pass the struct [UNINTELLIGIBLE].

PROFESSOR: No, you can't pass the struct. You basically have to pad the index in here for this structure, basically. So you have to know something in there.

So this is a lot messier, but it can actually give you a performance if you do this type of access, so it's not something you want everywhere. That's why you still want to use structures and nice classes and stuff. But this, if you're doing very large data, but most of the you're accessing only small amount, this kind of representation can help.

So I want to get a little bit into matrix multiplying. So what we're looking at is this four by four matrix, and you can represent it using this representation. But to make it clear, what I'm going to do is I'm just going to allocate this entire thing as a single array, and I'm going to do that [UNINTELLIGIBLE] calculation myself because if you go a little bit down, you realize this is allocated in a little bit of a weird way.

And so this is what we call row-major because all rows are next to each other, and you can access allocated column-major. Basically that means now A001, instead of

that, 1, 0 is next to 0, 0, 1, 0, 2, 0. And then basically saying same thing, [UNINTELLIGIBLE] instead of saying 4i plus j, it's just 4j plus i. You can do both. So this is my nice matrix multiply function, because the reason I did it this way, it's just very clear how many axis I'm going to use.

So if you look at C, let's look at only the inner loop. Most of the mapping matters in the inner loop. I go for k in 0 to size. Only first axis on C is going to miss because it's going up [UNINTELLIGIBLE] same K, da-da-da-da-da, so C is very nice in here. Only get one miss in here.

You have a question? OK. A has a streaming pattern because every time I update K, I go to the next element, next element, next element, next element. [INAUDIBLE]. What type of a pattern do I have in B?

AUDIENCE: Looks like [INAUDIBLE].

PROFESSOR: Yeah, exactly. I have jumping on memory. I'm jumping one to another.

So if you look at that, I am going through A like that, and B, and going through C element at a time, basically. So if you look at doing this in order to calculate one element of A, [INAUDIBLE] calculating these, what you have to do is I am getting B and C this way. I'm going through row here and column, and this row [UNINTELLIGIBLE] fit into the cache line through that because this is the memory. But C is just jumping all over the memory, and I have no locality in here.

So what can we do here? What's the simplest thing in here? How do we get some locality into the C?

AUDIENCE: [INAUDIBLE]

PROFESSOR: So yeah, you're coming, you're getting it. That's the next thing that's most important. That's the more advanced thing you can do. We'll get there. What was a simple thing you can do for C?

AUDIENCE: [INAUDIBLE]

PROFESSOR: [UNINTELLIGIBLE] So here's the [UNINTELLIGIBLE] execution time, original, and voila, when you transpose that, you can actually get a huge benefit in here because, now, I get spatial locality in both sides in here. And so now when you're transposed, I am going through like this.

So now what I did was I am giving this CPI, and what happens is when you transpose, I have a 5x improvement in sync CPI. That means instructions actually not waiting, not doing nothing. I have a pretty good reduce of cache misread. because, now, I'm actually going through cache very nicely.

So the next thing we can do is, what she just pointed out, blocking. So instead of matrix multiply, just multiplying this matrix at a time, you can multiply submatrices and calculate small submatrices. And if you do that, the submatrix can nicely fit in the memory, so let me show you that. This is the blocking code. Let me show you what that means.

So what we are doing is you are multiplying this matrix by this matrix in there and getting this matrix. So what that means is, normally, what you do is, to get one element, you have to multiply a row and a column. Normally, we are calculating a row of A at a time. In order to get a row of A, I had to get a row of B [UNINTELLIGIBLE] entire C has to participate into calculate A. So by the time I have calculated 1024 elements, I have gone through this many axises to new elements basically because I am getting B again and again, but I had to go through the entire C in here.

So if this much data is bigger than the cache, I have no locality in here. But if, instead of calculating a row, if I calculate a 32 by 32 block, which is again 1024 elements I am calculating. If I do that, I need a block here and a block here, and I only type this much data. OK, I'm getting the same amount of output, but because of reuse, I touch a lot less inputting here.

And just by doing this transformation, I can really reduce the amount of data. Basically, I can create a smaller working set, go through that much more, and reduce the amount of cache misses you get. So by doing that, I basically go out

another 3x improvement in my CPI, really brought down my L1 cache misses, and basically eliminated L2 cache misses. OK, so I got a really good speed up in here.

So you can block multiple levels, L1, L2, L3, because every time you figure out the workings and size, you block so the data in that will be safe in the cache in here. And this is kind of nasty to do by hand. We'll see.

So the other interesting thing is call stack locality. So if you do something like fib in here, traditional fib, if you do fib(4), you calculate a fib(4), fib(3), fib(2), you kind of build a stack, go up back again in here. So if you keep doing that, the nice thing is you're going up, but its-- Hopefully, one of these days, I am going to hit the wrong button, and--

What kind of locality do we have if we are accessing data like this in my call stack? What kind of locality do we have? I mean, I access here, fib(4), and I get back to fib(4) here. So within each frame, I am accessing data next to each other, so I can have a little bit of a spatial locality because, each frame, all the data is next to each other. But across frames, normally, I will have some temporal because I will get back to that data some time. OK?

OK, before I summarize, this one. If you are reading a lot of data, processing through stage one, [UNINTELLIGIBLE] and producing [UNINTELLIGIBLE], that means you're reading a huge amount of data through stage one, save, save data again, do stage two, you have really bad cache locality. One thing you can do is read a small chunk, do all the stages on that small chunk, save it, and go to another chunk.

So instead of doing the entire data, doing something [UNINTELLIGIBLE] producing, you can do this chunk-ifying, and by doing that, you can get a lot of locality. So once you bring in data, you like to do as many things as possible to that data before it gets out of the cache. That's the entire thing about locality, and you can restructure code through that.

So if you look at that, what we have done today, we have looked at things like

associativity, cache lines, cache sets in here, and how to address and map to a cache, we kind of did this thing, and we look at [UNINTELLIGIBLE] streamings versus InCache versus no locality type patterns. And then when you look at data structure transformation, we look at how to basically replace structured arrays and split data structures to make sure that you get a lot of nice locality if you access a certain amount of data, and pack data to get smaller cache footprints kind of things. And we also look at computer transformations, things like transpose copying, compute copy, outtype things you can do, so you can actually get nice locality, so you can do both data and compute transformations. And [UNINTELLIGIBLE] axises and [? re-cast ?] calculation stages, so this is kind of summary.

Next thing I want go is look at what machines you guys are using. So this is the machines we used last year. These are quad cores, and it had L1 cache, so these [UNINTELLIGIBLE]. So this has 32 kilobyte L1 cache, data cache [UNINTELLIGIBLE] two caches in here for data and instructions, each core, and then the cache line size is normally 64 bytes [UNINTELLIGIBLE].

And here, latency [UNINTELLIGIBLE], you get three cycles. If not, you get 14 cycles to go to to L2 in here if data doesn't meet there. And associated with this [UNINTELLIGIBLE PHRASE]. So that was what we did last year.

This year, we have a bigger machine that has multiple levels of caches. So each core has L1 instruction and data cache. It's kind of the same, 32 kilobyte cache, four nanosecond latency in here. Eight, three and four [UNINTELLIGIBLE], they kind of reduced that, but they added a 256-kilobyte L2 cache in here, and then a pretty large humongo 12-megabyte L3 cache in here.

The interesting thing is here, this pretty fast access to these ones. If you miss in here, this is about 2 and 1/2 times cost from going from here to here. From here to [UNINTELLIGIBLE], about a five times cost, so here to here is a five times cost.

The interesting thing that actually surprises a lot us was here to here, it's cost is not that different. [UNINTELLIGIBLE PHRASE] So actually memory, even though it's sitting out in there, they have managed to get it pretty cheap access. Normally, my

intuition say this should be much higher because you're going off-chip, so this is the structure here. It's a lot more complicated.

So here's something I produced last year. So what I did was I ran this program. And in these machines, there's a way to turn off the prefetch of the previous [UNINTELLIGIBLE]. So when you've turned on the prefetch and then you run, what you are doing is I am accessing data with a working set, small working set, little bit bigger working set. So this is basically when you consider working set size.

So you get a graph like this. I am hitting my L1 cache, and if it is more than that, I [UNINTELLIGIBLE] L1 cache, I go to my L2 cache. And if L2 misses, I have to go to main memory. Very traditional kind of a graph in here.

Another a little bit of an interesting thing, which is instead of accessing next item, I kind of skipped in here. So this trying to force how to get conflict misses. And so if you line up things like this exactly, what happens is you get these conflict misses, and you get this number.

So I have interesting story to tell. When I was a graduate student, we were running this bunch of benchmarks, and normally when you run a benchmark, you try it for an equal [UNINTELLIGIBLE], stuff like that. And we got this graph like this. I mean, we expect a graph going like this. OK, that how a benchmark [UNINTELLIGIBLE].

Then at one point, by accident, instead of typing an equal 64-something, I typed an equal 63, and suddenly, I got a performance like this. Wait a minute, what's going on? So in here, what I got was not a line. I got 16, 32, 64, 128, nice graph like that.

And then I just did 63, like down to here. I'm like, geeze, that can't be right. And then I start looking at 62, which is here. 61 is here.

So I realize if I enumerate all those things, I got a graph like this, basically. Kind of something like this. And the thing is, if I had never tried the 63, I would have basically thought my performance was like this, so far worse. Never realized because it was all having conflicts in memory because I'm [UNINTELLIGIBLE] 2 to the power numbers. So this is what happens if you just look at 2 to the power

numbers, you'll think, yeah, this is the performance you're getting, without realizing that, in fact, you can get this far.

So this is good. So I ran it on the new machine. The problem with our new machine is there's no way to turn off prefetching.

OK, so this is what you're getting. Everything, because I am accessing this nice linear, looks same. This is exactly what you want to happen.

What it's doing is, minute you go to access, it realizes you accessing linearly. It start fetching, fetching, fetching data in front you, it's going to catch up. It's not feeding you data, so you're not going to have any cache missed because you're going through this nice, linear pattern, and you just basic end up with this beautiful graph, which gives the feeling your working set could be, now, gigabytes long. At this point, I am fitting into the entire L3 cache, and still look like this works beautiful because this is what you want for it to figure out.

So now I'm like, geeze, this is great. This means my cache works, it's very nice, but I want to show you guys what's going on, so that doesn't help. So I came up with this program.

So this program, what I did was I created a working set, but I'm accessing randomly within that working set. I calculated this address. This is my poor man's random number generator, so basically I'm adding number in here and multiplying this, and then the masking these to get the right [UNINTELLIGIBLE] so I get within that set in here. And then I just access data, so I'm getting [UNINTELLIGIBLE]. So I get a number like this, and this jump up here, and then I had this kind of behavior.

This is a lot more complicated. I need to figure out what the hell is going on in here, so I said, OK, let's now look at performance counters. I look at L1 cache miss. OK, up to 32K, I have no cache misses. Boom, cache misses jump up. Perfect.

And I have this little bit of a kink in here, this beautiful describes because I have 32K L1 cache, and until then, I have no things, and then when the cache start missing, I'm jumping up. Perfect. I got that description here.

Then I look at my L2 cache misses. This is, I think, the L2 cache in this machine might have a little bit more complex [UNINTELLIGIBLE] telling us because this is not going like here had jumping up. That's a 256K [UNINTELLIGIBLE] get.

Is this 256? Yeah. [INAUDIBLE] 250 is [UNINTELLIGIBLE] these things. OK, so we have 32 256 2L megabytes in here. Even if you have 256 in here--

AUDIENCE: [INAUDIBLE]

PROFESSOR: Hm?

AUDIENCE: Should it be 256 or 32--

PROFESSOR: [UNINTELLIGIBLE] because everything's backed up, it's hierarchy [UNINTELLIGIBLE], so it's not added to, it's not next to each other, because at this point, L1 is now useless. It'll run everything is just going to L2, so L2 is the one who was serving it. Come on. So what happens in here, this normally should start jumping here, 256, but it start jumping earlier, so there might be some other interesting thing going on here. It might be interesting to investigate.

So this is OK, so still I am pretty close. I realized this jump is basically because of L2 happen, this jump in here. L2 start missing. Then I look at L3. It's here.

What's happening here? So this was a big mystery the two of us were trying to debug what one hour ago. Not one hour. Class is going. Like two hours ago.

We was trying to figure out, OK, wait a minute, this doesn't make sense. My performance is here. L3 should be just basically a nice flattening out here, and just jumping. It's not doing that, and we are like what's going on here.

So then we were scratching our heads, and we start looking. I say, ha, there were other things going on. We looked at this thing called TLB. Let me tell you what TLB is. What we realize is at this point, we are [? seeing ?] TLB misses.

How many people know what a TLB is? OK, good, so let me spend the last couple

of minutes explaining what a TLB this, and that kind of explain what's going on in this graph. So TLB means-- Normally, data is stored in memory using real addresses. That means there's a memory size in here, there's an address format, there's certain [UNINTELLIGIBLE].

And then the program for it use virtual memory. That means programs want to feel like they have a lot more memory than actual, physical memory available. So what happens is you have virtual memory address, but somebody has to map virtual memory to physical memory.

So the way it work is there's a thing called translation lookaside buffer, TLB, that says if you give this virtual address, here's the right physical address. So of course you can't do it for each address by address. What you have is you have 4K pages, so the address space is broken down into 4K pages. Each page will have an entry in this TLB, saying if you get this virtual address, here's the right physical address.

So normally what happens is when the operating system they created this large table, for every place in memory, you use this table, and it loads into memory. Then the hardware will fetch and cache some of those entries in this TLB cache in there. So normally what happens is in here, it caches 52L entries. It can keep 52L entries to 52L pages, saying here's the mapping.

The problem with 52L pages is in a 4K byte page size, 52L pages can only hold two megabytes of data. If you're accessing more than two megabytes of data, you have to go more than 52L pages, and at that point, that mapping is not in the TLB, and that means that you have to go do a mapping translation. The problem for this is these 4K pages is a very ancient artifact.

There's the new machines, you can I ask for what we call large-pages or super, but you can ask for 2-megabyte or 4-megabyte pages. If you ask for super-pages, then of course, 52L entry's good enough. It can hold enough things not to have TLB miss, at least while you're in the cache.

But since Linux, without doing anything, only asks for 4K pages, before you run out

of your L3 cache, you run out of the TLB table. And it has to start fetching and [UNINTELLIGIBLE] data table, that means you had to actually fetch the table entries, and so you have to do more fetchings. So that is why this is even running higher, because it's trying to fetch this TLB entry and stuff like that. Question?

AUDIENCE: Is there a way to then actually fiddle with the TLB so that you have to fit your pages rather than--

PROFESSOR: So in allocation, you can ask for super-pages or large-pages. But in Linux, normally just ask for 4K pages, so that the operating systems haven't kind of caught up to the current hardware. And this is was built [UNINTELLIGIBLE] doesn't make sense TLB to run out before you run out the cache. I mean, that's kind of mismatch in here. But it's built assuming you'll get large-pages, but if you don't, you're going to run out of the TLB in there.

So this is why sometimes performance engineering is interesting, exciting, and frustrating, because you assume you know where the hell this is going. You assume this is going to go here and that, and suddenly, this miss shows up, and so we have to scratch our heads. And luckily, we found what's going on, but we could have spent hours or days trying to figure out what's going on here because this is one [UNINTELLIGIBLE] say OK, let's try TLB, and it worked.

That was really good, but I have, many cases, spent days trying to figure out why something is going on because performance is kind of a [UNINTELLIGIBLE] of everything. There's no nice abstractions and stuff like that. If you think abstraction wise, somewhere outside the abstraction, that's going to kill you.

So that is why it can be interesting, because you have to know everything. And the fun thing is, once you understand performance, you basically understand end-to-end, soup to nuts, what's going on. OK, with that encouraging thought, let's see how you guys do in the next part of project two.