JULIAN SHUN: All right. So we've talked a little bit about caching before, but today we're going to talk in much more detail about caching and how to design cache-efficient algorithms. So first, let's look at the caching hardware on modern machines today.

So here's what the cache hierarchy looks like for a multicore chip. We have a whole bunch of processors. They all have their own private L1 caches for both the data, as well as the instruction. They also have a private L2 cache. And then they share a last level cache, or L3 cache, which is also called LLC.

They're all connected to a memory controller that can access DRAM. And then, oftentimes, you'll have multiple chips on the same server, and these chips would be connected through a network. So here we have a bunch of multicore chips that are connected together.

So we can see that there are different levels of memory here. And the sizes of each one of these levels of memory is different. So the sizes tend to go up as you move up the memory hierarchy. The L1 caches tend to be about 32 kilobytes. In fact, these are the specifications for the machines that you're using in this class. So 32 kilobytes for both the L1 data cache and the L1 instruction cache.

256 kilobytes for the L2 cache. so the L2 cache tends to be about 8 to 10 times larger than the L1 cache. And then the last level cache, the size is 30 megabytes. So this is typically on the order of tens of megabytes. And then DRAM is on the order of gigabytes. So here we have 128 gigabyte DRAM. And nowadays, you can actually get machines that have terabytes of DRAM.

So the associativity tends to go up as you move up the cache hierarchy. And I'll talk more about associativity on the next couple of slides. The time to access the memory also tends to go up. So the latency tends to go up as you move up the memory hierarchy. So the L1 caches are the quickest to access, about two nanoseconds, just rough numbers. The L2 cache is a little bit slower-- so say four nanoseconds. Last level cache, maybe six nanoseconds.

And then when you have to go to DRAM, it's about an order of magnitude slower-- so 50 nanoseconds in this example. And the reason why the memory is further down in the cache hierarchy are faster is because they're using more expensive materials to manufacture these things. But since they tend to be more expensive, we can't fit as much of that on the machines. So that's why the faster memories are smaller than the slower memories.

But if we're able to take advantage of locality in our programs, then we can make use of the fast memory as much as possible. And we'll talk about ways to do that in this lecture today. There's also the latency across the network, which tends to be cheaper than going to main memory but slower than doing a last level cache access.

And there's a lot of work in trying to get the cache coherence protocols right, as we mentioned before. So since these processors all have private caches, we need to make sure that they all see a consistent view of memory when they're trying to access the same memory addresses in parallel. So we talked about the MSI cache protocol before. And there are many other protocols out there, and you can read more about these things online. But these are very hard to get right, and there's a lot of verification involved in trying to prove that the cache coherence protocols are correct.

So any questions so far? OK.

So let's talk about the associativity of a cache. So here I'm showing you a fully associative cache. And in a fully associative cache, a cache block can reside anywhere in the cache. And a basic unit of movement here is a cache block. In this example, the cache block size is 4 bytes, but on the machines that we're using for this class, the cache block size is 64 bytes. But for this example, I'm going to use a four byte cache line.

So each row here corresponds to one cache line. And a fully associative cache means that each line here can go anywhere in the cache. And then here we're also assuming a cache size that has 32 bytes. So, in total, it can store eight cache line since the cache line is 4 bytes.

So to find a block in a fully associative cache, you have to actually search the entire cache, because a cache line can appear anywhere in the cache. And there's a tag associated with each of these cache lines here that basically specify which of the memory addresses in virtual memory space it corresponds to. So for the fully associative cache, we're actually going to use most of the bits of that address as a tag. We don't actually need the two lower order bits, because the things are being moved at the granularity of cache lines, which are four bytes. So the two lower order bits are always going to be the same, but we're just going to use the rest of the bits to store the tag.

So if our address space is 64 bits, then we're going to use 62 bits to store the tag in a fully associative caching scheme. And when a cache becomes full, a block has to be evicted to make room for a new block. And there are various ways that you can decide how to evict a block. So this is known as the replacement policy.

One common replacement policy is LRU Least Recently Used. So you basically kick the thing out that has been used the farthest in the past. The other schemes, such as second chance and clock replacement, we're not going to talk too much about the different replacement schemes today. But you can feel free to read about these things online.

So what's a disadvantage of this scheme? Yes?

AUDIENCE: It's slow.

JULIAN SHUN: Yeah. Why is it slow?

AUDIENCE: Because you have to go all the way [INAUDIBLE].

JULIAN SHUN: Yeah. So the disadvantage is that searching for a cache line in the cache can be pretty slow, because you have to search entire cache in the worst case, since a cache block can reside anywhere in the cache. So even though the search can go on in parallel and hardware is still expensive in terms of power and performance to have to search most of the cache every time.

So let's look at another extreme. This is a direct mapped cache. So in a direct mapped cache, each cache block can only go in one place in the cache. So I've color-coded these cache blocks here. So the red blocks can only go in the first row of this cache, the orange ones can only go in the second row, and so on. And the position which a cache block can go into is known as that cache blocks set. So the set determines the location in the cache for each particular block.

So let's look at how the virtual memory address is divided up into and which of the bits we're going to use to figure out where a cache block should go in the cache. So we have the offset, we have the set, and then the tag fields. The offset just tells us which position we want to access within a cache block. So since a cache block has B bytes, we only need log base 2 of B bits as the offset.

And the reason why we have to offset is because we're not always accessing something at the beginning of a cache block. We might want to access something in the middle. And that's why we need the offset to specify where in the cache block we want to access.

Then there's a set field. And the set field is going to determine which position in the cache that cache block can go into. So there are eight possible positions for each cache block. And therefore, we only need log base 2 of 8 bits-- so three bits for the set in this example. And more generally, it's going to be log base 2 of M over B. And here, M over B is 8.

And then, finally, we're going to use the remaining bits as a tag. So w minus log base 2 of M bits for the tag. And that gets stored along with the cache block in the cache. And that's going to uniquely identify which of the memory blocks the cache block corresponds to in virtual memory.

And you can verify that the sum of all these quantities here sums to w bits. So in total, we have a w bit address space. And the sum of those three things is w.

So what's the advantage and disadvantage of this scheme? So first, what's a good thing about this scheme compared to the previous scheme that we saw? Yes?

AUDIENCE: Faster.

JULIAN SHUN: Yeah. It's fast because you only have to check one place. Because each cache block can only go in one place in a cache, and that's only place you have to check when you try to do a lookup. If the cache block is there, then you find it. If it's not, then you know it's not in the cache.

What's the downside to this scheme? Yeah?

AUDIENCE: You only end up putting the red ones into the cache and you have mostly every [INAUDIBLE], which is totally [INAUDIBLE].

JULIAN SHUN: Yeah. So good answer. So the downside is that you might, for example, just be accessing the red cache blocks and then not accessing any of the other cache blocks. They'll all get mapped to the same location in the cache, and then they'll keep evicting each other, even though there's a lot of empty space in the cache. And this is known as a conflict miss. And these can be very bad for performance and very hard to debug. So that's one downside of a direct map cache is that you can get these conflict misses where you have to evict things from the cache even though there's empty space in the cache.

So as we said, finding a block is very fast. Only a single location in the cache has to be searched. But you might suffer from conflict misses if you keep axing things in the same set repeatedly without accessing the things in the other sets. So any questions? OK.

So these are sort of the two extremes for cache design. There's actually a hybrid solution called set associative cache. And in a set associative cache, you still sets, but each of the sets contains more than one line now. So all the red blocks still map to the red set, but there's actually two possible locations for the red blocks now.

So in this case, this is known as a two-way associate of cache since there are two possible locations inside each set. And again, a cache block's set determines k possible cache locations for that block. So within a set it's fully associative, but each block can only go in one of the sets.

So let's look again at how the bits are divided into in the address. So we still have the tag set and offset fields. The offset field is still a log base 2 of b. The set field is going to take log base 2 of M over kB bits. So the number of sets we have is M over kB. So we need log base 2 of that number to represent the set of a block. And then, finally, we use the remaining bits as a tag, so it's going to be w minus log base 2 of M over k.

And now, to find a block in the cache, only k locations of it's set must be searched. So you basically find which set the cache block maps too, and then you check all k locations within that set to see if that cached block is there. And whenever you want to whenever you try to put something in the cache because it's not there, you have to evict something. And you evict something from the same set as the block that you're placing into the cache.

So for this example, I showed a two-way associative cache. But in practice, the associated is usually bigger say eight-way, 16-way, or sometimes 20-way. And as you keep increasing the associativity, it's going to look more and more like a fully associative cache. And if you have a one way associative cache, then there's just a direct map cache. So this is a sort of a hybrid in between-- a fully mapped cache and a fully associative cache in a direct map cache. So any questions on set associative caches ? OK.

So let's go over a taxonomy of different types of cache misses that you can incur. So the first type of cache miss is called a cold miss. And this is the cache miss that you have to incur the first time you access a cache block. And if you need to access this piece of data, there's no way to get around getting a cold miss for this. Because your cache starts out not having this block, and the first time you access it, you have to bring it into cache.

Then there are capacity misses. So capacity misses are cache misses You get because the cache is full and it can't fit all of the cache blocks that you want to access. So you get a capacity miss when the previous cache copy would have been evicted even with a fully associative scheme. So even if all of the possible locations in your cache could be used for a particular cache line, that cache line still has to be evicted because there's not enough space. So that's what's called a capacity miss. And you can deal with capacity misses by introducing more locality into your code, both spatial and temporal locality. And we'll look at ways to reduce the capacity misses of algorithms later on in this lecture.

Then there are conflict misses. And conflict misses happen in set associate of caches when you have too many blocks from the same set wanting to go into the cache. And some of these have to be evicted, because the set can't fit all of the blocks. And these blocks wouldn't have been evicted if you had a fully associative scheme, so these are what's called conflict misses. For example, if you have 16 things in a set and you keep accessing 17 things that all belong in the set, something's going to get kicked out every time you want to access something. And these cache evictions might not have happened if you had a fully associative cache.

And then, finally, they're sharing misses. So sharing misses only happened in a parallel context. And we talked a little bit about true sharing a false sharing misses in prior lectures. So let's just review this briefly.

So a sharing miss can happen if multiple processors are accessing the same cache line and at least one of them is writing to that cache line. If all of the processors are just reading from the cache line, then the cache [INAUDIBLE] protocol knows how to make it work so that you don't get misses. They can all access the same cache line at the same time if nobody's modifying it.

But if at least one processor is modifying it, you could get either true sharing misses or false sharing misses. So a true sharing miss is when two processors are accessing the same data on the same cache line. And as you recall from a previous lecture, if one of the two processors is writing to this cache line, whenever it does a write it needs to acquire the cache line in exclusive mode and then invalidate that cache line and all other caches. So then when one another processor tries to access the same memory location, it has to bring it back into its own cache, and then you get a cache miss there.

A false sharing this happens if two processes are accessing different data that just happened to reside on the same cache line. Because the basic unit of movement is a cache line in the architecture. So even if you're asking different things, if they are on the same cache line, you're still going to get a sharing miss. And false sharing is pretty hard to deal with, because, in general, you don't know what data gets placed on what cache line.

There are certain heuristics you can use. For example, if you're mallocing a big memory region, you know that that memory region is contiguous, so you can space your access is far enough apart by different processors so they don't touch the same cache line. But if you're just declaring local variables on the stack, you don't know where the compiler is going to decide to place these variables in the virtual memory address space. So these are four different types of cache misses that you should know about.

And there's many models out there for analyzing the cache performance of algorithms. And some of the models ignore some of these different types of cache misses. So just be aware of this when you're looking at algorithm analysis, because not all of the models will capture all of these different types of cache misses.

So let's look at a bad case for conflict misses. So here I want to access a submatrix within a larger matrix. And recall that matrices are stored in row-major order. And let's say our matrix is 4,096 columns by 4,096 rows and it still stores doubles. So therefore, each row here is going to contain 2 to the 15th bytes, because 4,096 is t2 to the 12th, and we have doubles, which takes eight bytes. So 2 to the 12 times to the 3rd, which is 2 to the 15th.

We're going to assume the word width is 64, which is standard. We're going to assume that we have a cache size of 32k. And the cache block size is 64, which, again, is standard. And let's say we have a four-way associative cache.

So let's look at how the bits are divided into. So again we have this offset, which takes log base 2 of B bits. So how many bits do we have for the offset in this example? Right. So we have 6 bits. So it's just log base 2 of 64.

What about for the set? How many bits do we have for that? 7. Who said 7?

Yeah. So it is 7. So M is 32k, which is 2 to the 15th. And then k is 2 to the 2, b is 2 6. So it's 2 to the 15th divided by 2 the 8th, which is to the 7th. And log base 2 of that is 7.

And finally, what about the tag field?

AUDIENCE: 51.

JULIAN SHUN: 51. Why is that?

AUDIENCE: 64 minus 13.

JULIAN SHUN: Yeah. So it's just 64 minus 7 minus 6, which is 51. OK.

So let's say that we want to access a submatrix within this larger matrix. Let's say we want to acts as a 32 by 32 submatrix. And THIS is pretty common in matrix algorithms, where you want to access submatrices, especially in divide and conquer algorithms.

And let's say we want to access a column of this submatrix A. So the addresses of the elements that we're going to access are as follows-- so let's say the first element in the column is stored at address x. Then the second element in the column is going to be stored at address x plus 2 to the 15th, because each row has 2 to the 15th bytes, and we're skipping over an entire row here to get to the element in the next row of the sub matrix. So we're going to add 2 to the 15th.

And then to get the third element, we're going to add 2 times 2 to the 15th. And so on, until we get to the last element, which is x plus 31 times 2 to the 15th. So which fields of the address are changing as we go through one column of this submatrix?

AUDIENCE: You're just adding multiple [INAUDIBLE] tag the [INAUDIBLE].

JULIAN SHUN: Yeah. So it's just going to be the tag that's changing. The set and the offset are going to stay the same, because we're just using the lower 13 bits to store the set and a tag. And therefore, when we increment by 2 to the 15th, we're not going to touch the set and the offset.

So all of these addresses fall into the same set. And this is a problem, because our cache is only four-way associative. So we can only fit four cache lines in each set. And here, we're accessing 31 of these things. So by the time we get to the next column of A, all the things that we access in the current column of A are going to be evicted from cache already. And this is known as a conflict miss, because if you had a fully associative cache this might not have happened, because you could actually use any location in the cache to store these cache blocks.

So does anybody have any questions on why we get conflict misses here? So anybody have any ideas on how to fix this? So what can I do to make it so that I'm not incrementing by exactly 2 to the 15th every time? Yeah.

AUDIENCE: So pad the matrix?

JULIAN SHUN: Yeah. So one solution is to pad the matrix. You can add some constant amount of space to the end of the matrix. So each row is going to be longer than 2 to the 15th bytes. So maybe you add some small constant like 17. So add 17 bytes to the end of each row.

And now, when you access a column of this submatrix, you're not just incrementing by 2 to the 15th, you're also adding some small integer. And that's going to cause the set and the offset fields to change as well, and you're not going to get as many conflict misses. So that's one way to solve the problem.

It turns out that if you're doing a matrix multiplication algorithm, that's a cubic work algorithm, and you can basically afford to copy the submatrix into a temporary 32 by 32 matrix, do all the operations on the temporary matrix, and then copy it back out to the original matrix. The copying only takes quadratic work to do across the whole algorithm. And since the whole algorithm takes cubic work, the quadratic work is a lower order term. So you can use temporary space to make sure that you don't get conflict misses.

Any questions? So this was conflict misses. So conflict misses are important. But usually, we're going to be first concerned about getting good spatial and temporal locality, because those are usually the higher order factors in the performance of a program. And once we get good spatial and temporal locality in our program, we can then start worrying about conflict misses, for example, by using temporary space or padding our data by some small constants so that we don't have as if any conflict misses.

So now, I want to talk about a model that we can use to analyze the cache performance of algorithms. And this is called the ideal-cache model. So in this model, we have a two-level cache hierarchy. So we have the cache and then main memory.

The cache size is of size M, and the cache line size is of B bytes. And therefore, we can fit M over V cache lines inside our cache. This model assumes that the cache is fully associative, so any cache block can go anywhere in the cache. And it also assumes an optimal omniscient replacement policy.

So this means that where we want to evict a cache block from the cache, we're going to pick the thing to evict that gives us the best performance overall. It gives us the lowest number of cache misses throughout our entire algorithm. So we're assuming that we know the sequence of memory requests throughout the entire algorithm. And that's why it's called the omniscient mission replacement policy. And if something is in cache, you can operate on it for free. And if something is in main memory, you have to bring it into cache and then you incur a cache miss.

So two performance measures that we care about-- first, we care about the ordinary work, which is just the ordinary running time of a program. So this is the same as before when we were analyzing algorithms. It's just a total number of operations that the program does. And the number of cache misses is going to be the number of lines we have to transfer between the main memory and the cache. So the number of cache misses just counts a number of cache transfers, whereas as the work counts all the operations that you have to do in the algorithm.

So ideally, we would like to come up with algorithms that have a low number of cache misses without increasing the work from the traditional standard algorithm. Sometimes we can do that, sometimes we can't do that. And then there's a trade-off between the work and the number of cache misses. And it's a trade-off that you have to decide whether it's worthwhile as a performance engineer.

Today, we're going to look at an algorithm where you can't actually reduce the number of cache misses without increasing the work. So you basically get the best of both worlds. So any questions on this ideal cache model?

So this model is just used for analyzing algorithms. You can't actually buy one of these caches at the store. So this is a very ideal cache, and they don't exist.

But it turns out that this optimal omniscient replacement policy has nice theoretical properties. And this is a very important lemma that was proved in 1985. It's called the LRU lemma. It was proved by Slater and Tarjan. And the lemma says, suppose that an algorithm incurs Q cache misses on an ideal cache of size M. Then, on a fully associative cache of size 2M, that uses the LRU, or Least Recently Used replacement policy, it incurs at most 2Q cache misses.

So what this says is if I can show the number of cache misses for an algorithm on the ideal cache, then if I take a fully associative cache that's twice the size and use the LRU replacement policy, which is a pretty practical policy, then the algorithm is going to incur, at most, twice the number of cache misses. And the implication of this lemma is that for asymptotic analyses, you can assume either the optimal replacement policy or the LRU replacement policy as convenient. Because the number of cache misses is just going to be within a constant factor of each other. So this is a very important lemma. It says that this basically makes it much easier for us to analyze our cache misses in algorithms.

And here's a software engineering principle that I want to point out. So first, when you're trying to get good performance, you should come up with a theoretically good algorithm that has good balance on the work and the cache complexity. And then after you come up with an algorithm that's theoretically good, then you start engineering for detailed performance. You start worrying about the details such as real world caches not being fully associative, and, for example, loads and stores having different costs with respect to bandwidth and latency. But coming up with a theoretically good algorithm is the first order bit to getting good performance.

Questions?

So let's start analyzing the number of cache misses in a program. So here's a lemma. So the lemma says, suppose that a program reads a set of r data segments, where the i-th segment consists of s sub i bytes. And suppose that the sum of the sizes of all the segments is equal to N. And we're going to assume that N is less than M over 3. So the sum of the size of the segments is less than the cache size divided by 3.

We're also going to assume that N over r is greater than or equal to B. So recall that r is the number of data segments we have, and N is the total size of the segment. So what does N over r mean, semantically? Yes.

AUDIENCE: Average [INAUDIBLE].

JULIAN SHUN: Yeah. So N over r is the just the average size of a segment. And here we're saying that the average size of a segment is at least B-- so at least the size of a cache line. So if these two assumptions hold, then all of the segments are going to fit into cache, and the number of cache misses to read them all is, at most, 3 times N over B.

So if you had just a single array of size N, then the number of cache misses you would need to read that array into cache is going to be N over B. And this is saying that, even if our data is divided into a bunch of segments, as long as the average length of the segments is large enough, then the number of cache misses is just a constant factor worse than reading a single array. So let's try to prove this cache miss lemma.

So here's a proof so. A single segment, s sub i is going to incur at most s sub i over B plus 2 cache misses. So does anyone want to tell me where the s sub i over B plus 2 comes from? So let's say this is a segment that we're analyzing, and this is how it's aligned in virtual memory. Yes?

AUDIENCE: How many blocks it could overlap worst case.

JULIAN SHUN: Yeah. So s sub i over B plus 2 is the number of blocks that could overlap within the worst case. So you need s sub i over B cache misses just to load those s sub i bytes. But then the beginning and the end of that segment might not be perfectly aligned with a cache line boundary. And therefore, you could waste, at most, one block on each side of the segment. So that's where the plus 2 comes from.

So to get the total number of cache misses, we just have to sum this quantity from i equals 1 to r. So if I sum s sub i over B from i equals 1 to r, I just get N over B, by definition. And then I sum 2 from i equals 1 to r. So that just gives me 2r.

Now, I'm going to multiply the top and the bottom with the second term by B. So 2r B over B now. And then that's less than or equal to N over B plus 2N over B. So where did I get this inequality here? Why do I know that 2r B is less than or equal to 2N? Yes?

AUDIENCE: You know that the N is greater than or equal to B r.

JULIAN SHUN: Yeah. So you know that N is greater than or equal to B r by this assumption up here. So therefore, r B is less than or equal to N. And then, N B plus 2 N B just sums up to 3 N B. So in the worst case, we're going to incur 3N over B cache misses. So any questions on this cache miss lemma?

So the Important thing to remember here is that if you have a whole bunch of data segments and the average length of your segments is large enough-- bigger than a cache block size-- then you can access all of these segments just like a single array. It only increases the number of cache misses by a constant factor. And if you're doing an asymptotic analysis, then it doesn't matter. So we're going to be using this cache miss lemma later on when we analyze algorithms.

So another assumption that we're going to need is called the tall cache assumption. And the tall cache assumption basically says that the cache is taller than it is wide. So it says that B squared is less than c M for some sufficiently small constant c less than or equal to 1. So in other words, it says that the number of cache lines M over B you have is going to be bigger than B.

And this tall cache assumption is usually satisfied in practice. So here are the cache line sizes and the cache sizes on the machines that we're using. So cache line size is 64 bytes, and the L1 cache size is 32 kilobytes. So 64 bytes squared, that's 2 to the 12th. And 32 kilobytes is 2 to the 15th bytes.

So 2 to the 12th is less than 2 to the 15th, so it satisfies the tall cache assumption. And as we go up the memory hierarchy, the cache size increases, but the cache line length stays the same. So the cache has become even taller as we move up the memory hierarchy.

So let's see why this tall cache assumption is going to be useful. To see that, we're going to look at what's wrong with a short cache. So in a short cache, our lines are going to be very wide, and they're wider than the number of lines that we can have in our cache.

And let's say we're working with an m by n submatrix sorted in row-major order. If you have a short cache, then even if n squared is less than c M, meaning that you can fit all the bytes of the submatrix in cache, you might still not be able to fit it into a short cache. And this picture sort of illustrates this.

So we have m rows here. But we can only fit M over B of the rows in the cache, because the cache lines are so long, and we're actually wasting a lot of space on each of the cache lines. We're only using a very small fraction of each cache line to store the row of this submatrix. If this were the entire matrix, then it would actually be OK, because consecutive rows are going to be placed together consecutively in memory. But if this is a submatrix, then we can't be guaranteed that the next row is going to be placed right after the current row. And oftentimes, we have to deal with submatrices when we're doing recursive matrix algorithms.

So this is what's wrong with short caches. And that's why we want us assume the tall cache assumption. And we can assume that, because it's usually satisfied in practice. The TLB be actually tends to be short. It only has a couple of entries, so it might not satisfy the tall cache assumption. But all of the other caches will satisfy this assumption.

Any questions? OK.

So here's another lemma that's going to be useful. This is called the submatrix caching llama. So suppose that we have an n by m matrix, and it's read into a tall cache that satisfies B squared less than c M for some constant c less than or equal to 1. And suppose that n squared is less than M over 3, but it's greater than or equal to c M. Then A is going to fit into cache, and the number of cache misses required to read all of A's elements into cache is, at most, 3n squared over B.

So let's see why this is true. So we're going to let big N denote the total number of bytes that we need to access. So big N is going to be equal to n squared. And we're going to use the cache miss lemma, which says that if the average length of our segments is large enough, then we can read all of the segments in just like it were a single contiguous array.

So the lengths of our segments here are going to be little n. So r is going to be a little n. And also, the number of segments is going to be little n. And the segment length is also going to be little n, since we're working with a square submatrix here.

And then we also have the cache block size B is less than or equal to n. And that's equal to big N over r. And where do we get this property that B is less than or equal to n? So I made some assumptions up here, where I can use to infer that B is less than or equal to n. Does anybody see where? Yeah.

AUDIENCE: So B squared is less than c M, and c M is [INAUDIBLE]

JULIAN SHUN: Yeah. So I know that B squared is less than c M. C M is less than or equal to n squared. So therefore, B squared is less than n squared, and B is less than n.

So now, I also have that N is less than M over 3, just by assumption. And therefore, I can use the cache miss lemma. So the cache miss lemma tells me that I only need a total of 3n squared over B cache misses to read this whole thing in. Any questions on the submatrix caching lemma?

So now, let's analyze matrix multiplication. How many of you have seen matrix multiplication before? So a couple of you.

So here's what the code looks like for the standard cubic work matrix multiplication algorithm. So we have two input matrices, A and B, And we're going to store the result in C. And the height and the width of our matrix is n. We're just going to deal with square matrices here, but what I'm going to talk about also extends to non-square matrices.

And then we just have three loops here. We're going to loop through i from 0 to n minus 1, j from 0 to n minus 1, and k from 0 to n minus 1. And then we're going to let's C of i n plus j be incremented by a of i n plus k times b of k n plus j. So that's just the standard code for matrix multiply.

So what's the work of this algorithm? It should be review for all of you. n cubed.

So now, let's analyze the number of cache misses this algorithm is going to incur. And again, we're going to assume that the matrix is in row-major order, and we satisfy the tall cache assumption. We're also going to analyze the number of cache misses in matrix B, because it turns out that the number of cache misses incurred by matrix B is going to dominate the number of cache misses overall.

And there are three cases we need to consider. The first case is when n is greater than c M over B for some constant c. And we're going to analyze matrix B, as I said. And we're also going to assume LRU, because we can. If you recall, the LRU lemma says that whatever we analyze using the LRU is just going to be a constant factor within what we analyze using the ideal cache.

So to do this matrix multiplication, I'm going to go through one row of A and one column of B and do the dot product there. This is what happens in the innermost loop. And how many cache misses am I going to incur when I go down one column of B here?

So here, I have the case where n is greater than M over B. So I can't fit one block from each row into the cache. So how many cache misses do I have the first time I go down a column of B? So how many rows of B do I have? n. Yeah, and how many cache misses do I need for each row? One. So in total, I'm going to need n cache misses for the first column of B.

What about the second column of B? So recall that I'm assuming the LRU replacement policy here. So when the cache is full, I'm going to evict the thing that was least recently used-- used the furthest in the past. Sorry, could you repeat that?

AUDIENCE: [INAUDIBLE].

JULIAN SHUN: Yeah. So it's still going to be n. Why is that?

AUDIENCE: Because there are [INAUDIBLE] integer.

JULIAN SHUN: Yeah. It's still going to be n, because I can't fit one cache block from each row into my cache. And by the time I get back to the top of my matrix B, the top block has already been evicted from the cache, and I have to load it back in. And this is the same for every other block that I access. So I'm, again, going to need n cache misses for the second column of B. And this is going to be the same for all the columns of B.

And then I have to do this again for the second row of A. So in total, I'm going to need theta of n cubed number of cache misses. And this is one cache miss per entry that I access in B. And this is not very good, because the total work was also theta of n cubed. So I'm not gaining anything from having any locality in this algorithm here.

So any questions on this analysis? So this just case 1. Let's look at case 2.

So in this case, n is less than c M over B. So I can fit one block from each row of B into cache. And then n is also greater than another constant, c prime time square root of M, so I can't fit the whole matrix into cache. And again, let's analyze the number of cache misses incurred by accessing B, assuming LRU.

So how many cache misses am I going to incur for the first column of B?

AUDIENCE: n.

JULIAN SHUN: n. So that's the same as before. What about the second column of B? So by the time I get to the beginning of the matrix here, is the top block going to be in cache? So who thinks the block is still going to be in cache when I get back to the beginning? Yeah. So a couple of people. Who think it going to be out of cache?

So it turns out it is going to be in cache, because I can fit one block for every row of B into my cache since I have n less than c M over B. So therefore, when I get to the beginning of the second column, that block is still going to be in cache, because I loaded it in when I was accessing the first column. So I'm not going to incur any cache misses for the second column. And, in general, if I can fit B columns or some constant times B columns into cache, then I can reduce the number of cache misses I have by a factor of B. So I only need to incur a cache miss the first time I access of block and not for all the subsequent accesses.

And the same is true for the second row of A. And since I have m rows of A, I'm going to have n times theta of n squared over B cache misses. For each row of A, I'm going to incur n squared over B cache misses.

So the overall number of cache misses is n cubed over B. And this is because inside matrix B I can exploit spatial locality. Once I load in a block, I can reuse it the next time I traverse down a column that's nearby. Any questions on this analysis?

So let's look at the third case. And here, n is less than c prime times square root of M. So this means that the entire matrix fits into cache. So let's analyze the number of cache misses for matrix B again, assuming LRU. So how many cache misses do I have now?

So let's count the total number of cache misses I have for every time I go through a row of A. Yes.

AUDIENCE: Is it just n for the first column?

JULIAN SHUN: Yeah. So for the first column, it's going to be n. What about the second column?

AUDIENCE: [INAUDIBLE] the second [INAUDIBLE].

JULIAN SHUN: Right. So basically, for the first row of A, the analysis is going to be the same as before. I need n squared over B cache misses to load the cache in. What about the second row of A? How many cache misses am I going to incur?

AUDIENCE: [INAUDIBLE].

JULIAN SHUN: Yeah. So for the second row of A, I'm not going to incur any cache misses. Because once I load B into cache, it's going to stay in cache. Because the entire matrix can fit in cache, I assume that n is less than c prime times square root of M. So total number of cache misses I need for matrix B is theta of n squared over B since everything fits in cache. And I just apply the submatrix caching lemma from before. Overall, this is not a very good algorithm. Because as you recall, in case 1 I needed a cubic number of cache misses.

What happens if I swap the order of the inner two loops? So recall that this was one of the optimizations in lecture 1, when Charles was talking about matrix multiplication and how to speed it up. So if I swapped the order of the two inner loops, then, for every iteration, what I'm doing is I'm actually going over a row of C and a row of B, and A stays fixed inside the innermost iteration.

So now, when I analyze the number of cache misses of matrix B, assuming LRU, I'm going to benefit from spatial locality, since I'm going row by row and the matrix is stored in row-major order. So across all of the rows, I'm just going to require theta of n squared over B cache misses. And I have to do this n times for the outer loop. So in total, I'm going to get theta of n cubed over B cache misses.

So if you swap the order of the inner two loops this significantly improves the locality of your algorithm and you can benefit from spatial accounting. That's why we saw a significant performance improvement in the first lecture when we swapped the order of the loops. Any questions?

So does anybody think we can do better than n cubed over B cache misses? Or do you think that it's the best you can do? So how many people think you can do better? Yeah. And how many people think this is the best you can do? And how many people don't care?

So it turns out you can do better. And we're going to do better by using an optimization called tiling. So how this is going to work is instead of just having three for loops, I'm going to have six for loops. And I'm going to loop over tiles.

So I've got a loop over s by s submatrices. And within each submatrix, I'm going to do all of the computation I need for that submatrix before moving on to the next submatrix. So the three innermost loops are going to loop inside a submatrix, and a three outermost loops are going to loop within the larger matrix, one submatrix matrix at a time.

So let's analyze the work of this algorithm. So the work that we need to do for a submatrix of size s by s is going to be s cubed, since that's just a bound for matrix multiplication. And then a number of times I have to operate on submatrices is going to be n over s cubed. And you can see this if you just consider each submatrix to be a single element, and then using the same cubic work analysis on the smaller matrix.

So the work is n over s cubed times s cubed, which is equal to theta of n cubed. So the work of this tiled matrix multiplies the same as the version that didn't do tiling. And now, let's analyze the number of cache misses.

So we're going to tune s so that the submatrices just fit into cache. So we're going to set s to be equal to theta of square root of M. We actually need to make this 1/3 square root of M, because we need to fit three submatrices in the cache. But it's going to be some constant times square root of M.

The submatrix caching level implies that for each submatrix we're going to need x squared over B misses to load it in. And once we loaded into cache, it fits entirely into cache, so we can do all of our computations within cache and not incur any more cache misses. So therefore, the total number of cache misses we're going to incur, it's going to be the number of subproblems, which is n over s cubed and then a number of cache misses per subproblem, which is s squared over B.

And if you multiply this out, you're going to get n cubed over B times square root of M. So here, I plugged in square root of M for s. And this is a pretty cool result, because it says that you can actually do better than the n cubed over B bound. You can improve this bound by a factor of square root of M.

And in practice, square root of M is actually not insignificant. So, for example, if you're looking at the last level cache, the size of that is on the order of megabytes. So a square root of M is going to be on the order of thousands. So this significantly improves the performance of the matrix multiplication code if you tune s so that the submatrices just fit in the cache.

It turns out that this bound is optimal. So this was shown in 1981. So for cubic work matrix multiplication, this is the best you can do. If you use another matrix multiply algorithm, like Strassen's algorithm, you can do better.

So I want you to remember this bound. It's a very important bound to know. It says that for a matrix multiplication you can benefit both from spatial locality as well as temporal locality. So I get spatial locality in the B term in the denominator. And then the square root of M term comes from temporal locality, since I'm doing all of the work inside a submatrix before I evict that submatrix from cache. Any questions on this analysis?

So what's one issue with this algorithm here? Yes.

AUDIENCE: It's not portable, like different architecture [INAUDIBLE].

JULIAN SHUN: Yeah. So the problem here is I have to tune s for my particular machine. And I call this a voodoo parameter. It's sort of like a magic number I put into my program so that it fits in the cache on the particular machine I'm running on. And this makes the code not portable, because if I try to run this code on a another machine, the cache sizes might be different there, and then I won't get the same performance as I did on my machine.

And this is also an issue even if you're running it on the same machine, because you might have other programs running at the same time and using up part of the cache. So you don't actually know how much of the cache your program actually gets to use in a multiprogramming environment.

And then this was also just for one level of cache. If we want to optimize for two levels of caches, we're going to have two voodoo parameters, s and t. We're going to have submatrices and sub-submatrices. And then we have to tune both of these parameters to get the best performance on our machine. And multi-dimensional tuning optimization can't be done simply with binary search. So if you're just tuning for one level of cache, you can do a binary search on the parameter s, but here you can't do binary search. So it's much more expensive to optimize here.

And the code becomes a little bit messier. You have nine for loops instead of six. And how many levels of caches do we have on the machines that we're using today?

AUDIENCE: Three.

JULIAN SHUN: Three. So for three level cache, you have three voodoo parameters. You have 12 nested for loops. This code becomes very ugly. And you have to tune these parameters for your particular machine. And this makes the code not very portable, as one student pointed out. And in a multiprogramming environment, you don't actually know the effective cache size that your program has access to. Because other jobs are running at the same time, and therefore it's very easy to mistune the parameters.

Was their question? No? So any questions? Yeah. Is there a way to programmatically get the size of the cache? [INAUDIBLE].

JULIAN SHUN: Yeah. So you can auto-tune your program so that it's optimized for the cache sizes of your particular machine.

AUDIENCE: [INAUDIBLE] instruction to get the size of the cache [INAUDIBLE].

JULIAN SHUN: Instruction to get the size of your cache. I'm not actually sure. Do you know?

AUDIENCE: [INAUDIBLE] in--

AUDIENCE: [INAUDIBLE].

AUDIENCE: Yeah, in the proc--

JULIAN SHUN: Yeah, proc cpuinfo.

AUDIENCE: Yeah. proc cpuinfo or something like that.

JULIAN SHUN: Yeah. So you can probably get that as well.

AUDIENCE: And I think if you google, I think you'll find it pretty quickly.

JULIAN SHUN: Yeah.

AUDIENCE: Yeah. But even if you do that, and you're running this program when other jobs are running, you don't actually know how much cache your program has access to. Yes? Is cache of architecture and stuff like that optimized around matrix problems?

JULIAN SHUN: No. They're actually general purpose. Today, we're just looking at matrix multiply, but on Thursday's lecture we'll actually be looking at many other problems and how to optimize them for the cache hierarchy. Other questions?

So this was a good algorithm in terms of cache performance, but it wasn't very portable. So let's see if we can do better. Let's see if we can come up with a simpler design where we still get pretty good cache performance.

So we're going to turn to divide and conquer. We're going to look at the recursive matrix multiplication algorithm that we saw before. Again, we're going to deal with square matrices, but the results generalize to non-square matrices.

So how this works is we're going to split our [INAUDIBLE] matrices into four submatrices or four quadrants. And then for each quadrant of the output matrix, it's just going to be the sum of two matrix multiplies on n over 2 by n over 2 matrices. So c 1 1 one is going to be a 1 1 times b 1 1, plus a 1 2 times B 2 1.

And then we're going to do this recursively. So every level of recursion we're going to get eight multiplied adds of n over 2 by n over 2 matrices. Here's what the recursive code looks like. You can see that we have eight recursive calls here. The base case here is of size 1. In practice, you want to coarsen the base case to overcome function call overheads.

Let's also look at what these values here correspond to. So I've color coded these so that they correspond to particular elements in the submatrix that I'm looking at on the right. So these values here correspond to the index of the first element in each of my quadrants. So the first element in my first quadrant is just going to have an offset of 0. And then the first element of my second quadrant, that's going to be on the same row as the first element in my first quadrant. So I just need to add the width of my quadrant, which is n over 2.

And then to get the first element in quadrant 2 1, I'm going to jump over and over two rows. And each row has a length row size, so it's just going to be n over 2 times row size. And then to get the first element in quadrant 2 2, it's just the first element in quadrant 2 1 plus n over 2. So that's n over 2 times row size plus 1.

So let's analyze the work of this algorithm. So what's the recurrence for this algorithm-- for the work of this algorithm? So how many subproblems do we have?

AUDIENCE: Eight

JULIAN SHUN: Eight. And what's the size of each Subproblem n over 2. And how much work are we doing to set up the recursive calls? A constant amount of work.

So the recurrences is W of n is equal to 8 W n over 2 plus theta of 1. And what does that solve to? n cubed. So it's one of the three cases of the master theorem.

We're actually going to analyze this in more detail by drawing out the recursion tree. And this is going to give us more intuition about why the master theorem is true. So at the top level of my recursion tree, I'm going to have a problem of size n. And then I'm going to branch into eight subproblems of size n over 2. And then I'm going to do a constant amount of work to set up the recursive calls.

Here, I'm just labeling this with one. So I'm ignoring the constants. But it's not going to matter for asymptotic analysis.

And then I'm going to branch again into eight subproblems of size n over 4. And eventually, I'm going to get down to the leaves. And how many levels do I have until I get to the leaves?

Yes?

AUDIENCE: Log n.

JULIAN SHUN: Yeah. So log n-- what's the base of the log? Yeah. So it's log base 2 of n, because I'm dividing my problem size by 2 every time. And therefore, the number of leaves I have is going to be 8 to the log base 2 of n, because I'm branching it eight ways every time. 8 to the log base 2 of n is the same as n to the log base 2 of 8, which is n cubed.

The amount of work I'm doing at the top level is constant. So I'm just going to say 1 here. At the next level, it's eight times, then 64. And then when I get to the leaves, it's going to be theta of n cubed, since I have m cubed leaves, and they're all doing constant work.

And the work is geometrically increasing as I go down the recursion tree. So the overall work is just dominated by the work I need to do at the leaves. So the overall work is just going to be theta of n cubed. And this is the same as the looping versions of matrix multiply-- they're all cubic work.

Now, let's analyze the number of cache misses of this divide and conquer algorithm. So now, my recurrence is going to be different. My base case now is when the submatrix fits in the cache-- so when n squared is less than c M. And when that's true, I just need to load that submatrix into cache, and then I don't incur any more cache misses. So I need theta of n squared over B cache misses when n squared is less than c M for some sufficiently small constant c, less than or equal to 1.

And then, otherwise, I recurse into 8 subproblems of size n over 2. And then I add theta of 1, because I'm doing a constant amount of work to set up the recursive calls. And I get this state of n squared over B term from the submatrix caching lemma. It says I can just load the entire matrix into cache with this many cache misses.

So the difference between the cache analysis here and the work analysis before is that I have a different base case. And I think in all of the algorithms that you've seen before, the base case was always of a constant size. But here, we're working with a base case that's not of a constant size.

So let's try to analyze this using the recursion tree approach. So at the top level, I have a problem of size n that I'm going to branch into eight problems of size n over 2. And then I'm also going to incur a constant number of cache misses. I'm just going to say 1 here. Then I'm going to branch again.

And then, eventually, I'm going to get to the base case where n squared is less than c M. And when n squared is less than c M, then the number of cache misses that I'm going to incur is going to be theta of c M over B. So I can just plug-in c M here for n squared.

And the number of levels of recursion I have in this recursion tree is no longer just log base 2 of n. I'm going to have log base 2 of n minus log base 2 of square root of c M number of levels, which is the same as log base 2 of n minus 1/2 times log base 2 of c M. And then, the number of leaves I get is going to be 8 to this number of levels here. So it's 8 to log base 2 of n minus 1/2 of log base 2 of c M. And this is equal to the theta of n cubed over M to the 3/2.

So the n cubed comes from the 8 to the log base 2 of n term. And then if I do 8 to the negative 1/2 of log base 2 of c M, that's just going to give me M to the 3/2 in the denominator. So any questions on how I computed the number of levels of this recursion tree here?

So I'm basically dividing my problem size by 2 until I get to a problem size that fits in the cache. So that means n is less than square root of c M. So therefore, I can subtract that many levels for my recursion tree. And then to get the number of leaves, since I'm branching eight ways, I just do 8 to the power of the number of levels I have. And then that gives me the total number of leaves.

So now, let's analyze a number of cache misses I need each level of this recursion tree. At the top level, I have a constant number of cache misses-- let's just say 1. The next level, I have 8, 64. And then at the leaves, I'm going to have theta of n cubed over B times square root of M cache misses. And I got this quantity just by multiplying the number of leaves by the number of cache misses per leaf.

So number of leaves is n cubed over M to the 3/2. The cache misses per leaves is theta of c M over B. So I lose one factor of B in the denominator. I'm left with the square root of M at the bottom. And then I also divide by the block size B.

So overall, I get n cubed over B times square root of M cache misses. And again, this is a geometric series. And the number of cache misses at the leaves dominates all of the other levels. So the total number of cache misses I have is going to be theta of n cubed over B times square root of M.

And notice that I'm getting the same number of cache misses as I did with the tiling version of the code. But here, I don't actually have the tune my code for the particular cache size. So what cache sizes does this code work for? So is this code going to work on your machine? Is it going to get good cache performance?

So this code is going to work for all cache sizes, because I didn't tune it for any particular cache size. And this is what's known as a cache-oblivious algorithm. It doesn't have any voodoo tuning parameters, it has no explicit knowledge of the caches, and it's essentially passively auto-tuning itself for the particular cache size of your machine.

It can also work for multi-level caches automatically, because I never specified what level of cache I'm analyzing this for. I can analyze it for any level of cache, and it's still going to give me good cache complexity. And this is also good in multiprogramming environments, where you might have other jobs running and you don't know your effective cache size. This is just going to passively auto-tune for whatever cache size is available.

It turns out that the best cache-oblivious codes to date work on arbitrary rectangular matrices. I just talked about square matrices, but the best codes work on rectangular matrices. And they perform binary splitting instead of eight-way splitting. And you're split on the largest of i, j, and k. So this is what the best cache-oblivious matrix multiplication algorithm does. Any questions?

So I only talked about the serial setting so far. I was assuming that these algorithms ran on just a single thread. What happens if I go to multiple processors? It turns out that the results do generalize to a parallel context. So this is the recursive parallel matrix multiply code that we saw before. And notice that we're executing four sub calls in parallel, doing a sync, and then doing four more sub calls in parallel.

So let's try to analyze the number of cache misses in this parallel code. And to do that, we're going to use this theorem, which says that let Q sub p be the number of cache misses in a deterministic cell computation why run on P processors, each with a private cache of size M. And let S sub p be the number of successful steals during the computation. In the ideal cache model, the number of cache misses we're going to have is Q sub p equal to Q sub 1 plus big O of number of steals times M over B. So the number of cache misses in the parallel context is equal to the number of cache misses when you run it serially plus this term here, which is the number of steals times M over B.

And the proof for this goes as follows-- so every call in the Cilk runtime system, we can have workers steal tasks from other workers when they don't have work to do. And after a worker steals a task from another worker, it's cache becomes completely cold in the worst case, because it wasn't actually working on that subproblem before. But after M over B cold cache misses, its cache is going to become identical to what it would be in the serial execution. So we just need to pay M over B cache misses to make it so that the cache looks the same as if it were executing serially.

And the same is true when a worker resumes a stolen subcomputation after a Cilk sync. And the number of times that these two situations can happen is 2 times as S p-- 2 times the number of steals. And each time, we have to pay M over b cache misses. And this is where this additive term comes from-- order S sub p times M over B.

We also know that the number of steals in a Cilk program is upper-bounded by P times T infinity, in the expectation where P is the number of processors and T infinity is the span of your computation. So if you can minimize the span of your computation, then this also gives you a good cache bounds. So moral of the story here is that minimizing the number of cache misses in a serial elision essentially minimizes them in the parallel execution for a low span algorithm.

So in this recursive matrix multiplication algorithm, the span of this is as follows-- so T infinity of n is 2T infinity of of n over 2 plus theta of 1. Since we're doing a sync here, we have to pay the critical path length of two sub calls. This solves to theta of n. And applying to previous lemma, this gives us a cache miss bound of theta of n cubed over B square root of M. This is just the same as the serial execution And then this additive term is going to be order P times n. And it's a span times M over B

So that was a parallel algorithm for matrix multiply. And we saw that we can also get good cache bounds there.

So here's a summary of what we talked about today. We talked about associativity and caches, different ways you can design a cache. We talked about the ideal cache model that's useful for analyzing algorithms. We talked about cache-aware algorithms that have explicit knowledge of the cache. And the example we used was titled matrix multiply. Then we came up with a much simpler algorithm that was cache-oblivious using divide and conquer.

And then on Thursday's lecture, we'll actually see much more on cache-oblivious algorithm design. And then you'll also have an opportunity to analyze the cache efficiency of some algorithms in the next homework.