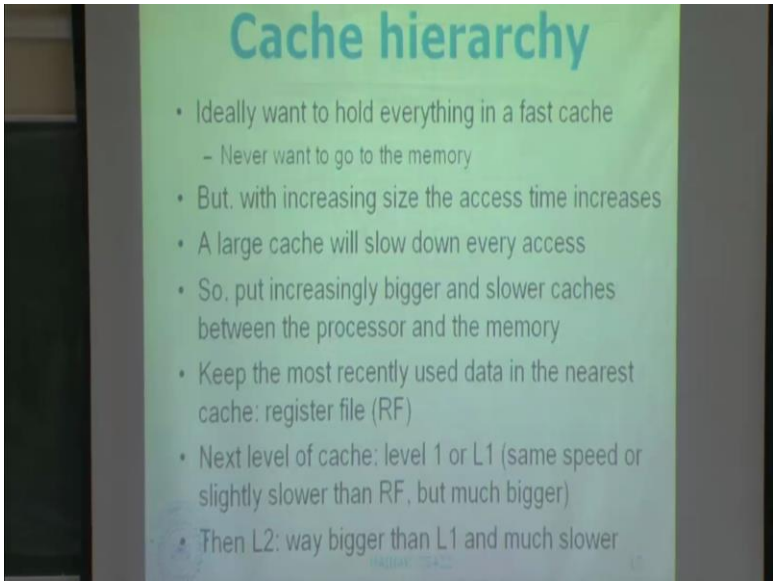
Computer Architecture
Prof. Mainak Chaudhuri
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Module - 1
Lecture - 26
Virtual Memory and caches

(Refer Slide Time: 00:45)



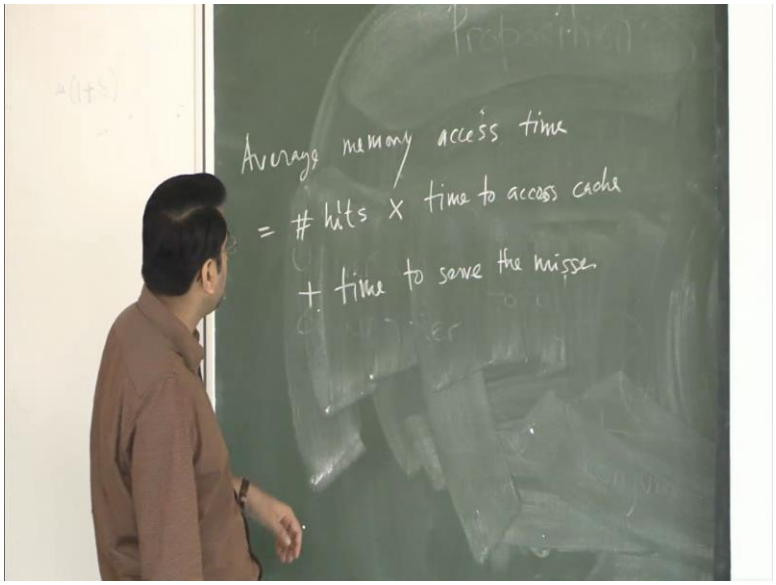
So, we roughly talked about what a cache is. You access at the cache, index into the cache and find out whether the data is available or not. So, today what we will do is we will start with something called cache hierarchy. That is what we use in this processor. So, it is working with; the observation is that ideally I want to hold everything in a fast cache. So, I completely wanted to avoid going into the memory.

But as we know, with increasing size the access time increases that we are talking about. So, any structure if you make it larger, it will take more time in the access; which essentially means, a large cache will slow down every access. So whenever you access, the access time will get longer.

So ((refer time: 01:15)) should be very clear here. If I have a very large cache, I have a large number of hits. So, I will never could have ((Refer Time: 01:28)) never missing up cache, if I am

making it large enough. But, every hit will be extremely slow. So, if you look at the total time spent, that is, essentially the product of number of hits and the time to access the cache plus a miss, how much time you text to get the data from the memory. Yeah. That is a total time spent in accessing the data; whatever amount of data you have.

(Refer Slide Time: 01:57)



So, that is often called the average memory access time. That is, a number of hits multiplied by time to access cache plus time to serve the misses. Or, that this also can be decomposed into number of misses times, times of miss. So, essentially a large cache nullifies the second term completely. An infinite cache will not have the second term. Well, not exactly true. Initially if you start executing, you have to bring some data from ((Refer Time: 02:54) and once you have all the data in the cache, you will never miss again.

So, when there is a large cache tries to maximize this particular term, number of hits, alright, but this is going to be also very large. So, it depends on how these are balanced. And, a large cache normally is not very good. So, what people do is they put increasingly bigger and slower caches between the processor and the memory. And, the rational here is that again the locality principles that we discussed last time is working here. That is, when you access a particular data point, the speculation is that a near future will access the data point once again as a temporal value.

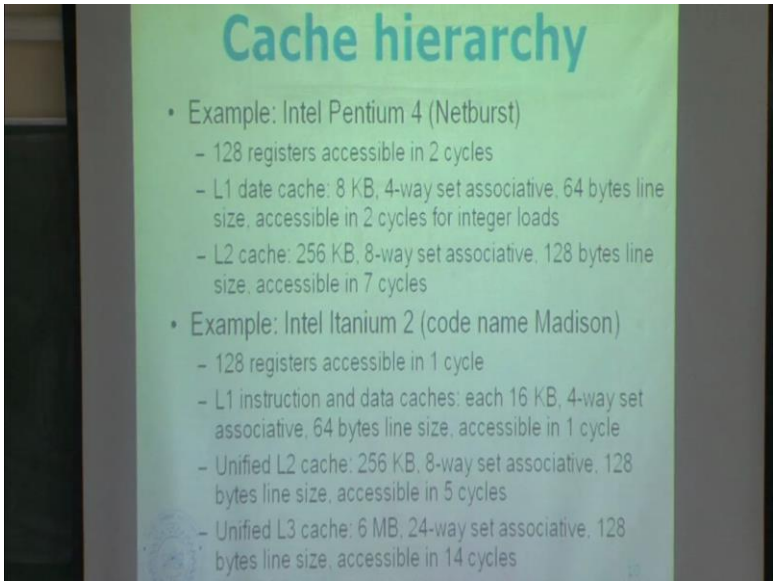
So, what you are doing here essentially is that, you put a small cache close to the processor.. You bring a data point, put it in the cache and the hope is that you will access it again, if future can have a hit. For over time, these data will become old and will become useless. And, that data will eventually get replacements from other data. Again, that will draw again.

So, this small cache close to your processor is a very fast cache. That is giving you most of the hits from the cache. If the first time that you touch a data, it will have to accommodate it. Then, you put a slightly bigger cache after it; usually accommodate with some more data. And, it will be slightly slower and so on and so forth.

So, overall this is going to be better than this. And, this is the punch line. That is, skip this most recently used data in the nearest cache, which is a register file; however, what you put in the register file is not really under the control of the hardware or the architecture. This is decided by the compiler, when you compile a program; what data goes into what register and in what time.

The next level of the cache is the L 1 cache, the level 1 cache. It is the same speed or slightly slower than the register file. But, normally much bigger than the register file. Then you have the L 2 cache; way bigger than L 1 and much slower. Then we have the L 3 cache; it is even bigger and slower. And, today industry is even contemplating in L 4 cache; belongs to this ((Refer Time: 05:15)) So you put; you know gradually larger and slower caches between the processor and the memory within this particular latency gap gradually.

(Refer Slide Time: 05:27)



So, again an example; look at couple of examples. So Intel Pentium 4 is an old processor. So, I am talking about Intel Pentium 4 had several incarnations gradually; means, increasing frequency. Netburst was the first architecture that came out. So, here is the memory hierarchy of Netburst, which had one twenty eight registers accessible in two cycles. So, in this class whenever I talk about registers, I really mean physical registers. We talked about the distinction between physical and logical registers. Ok right.

So, of course all Intel processors will have only eight logical registers. We have X 86((Refer Time: 06:06) registers. Here, we are talking about the physical registers inside the processor. It is not visible to the compiler managed by the hardware. So, it has one twenty eight registers accessible in two cycles. However, again even though this is under the management of hardware, remember that what data goes into what register is still decided by the compiler; because it is strict mapping from logical to physical register all the time.

Then you have the L 1 data cache, which is eight kilo byte in size, four-way set associative, sixty four bytes line size accessible in two cycles for integer loads. Your L 2 cache; which is two fifty six kilo byte, eight- way associative, one twenty eight byte line size accessible in seven cycles.

As you can see, when you go down let us see it increases the size also. Intel Itanium 2; so, again Itanium 2 had several incarnations. So, this is the Madison processor. It has one twenty eight

registers accessible in the cycle. L 1 instruction data caches sixteen kilobyte each, four-way set associative, sixty four byte line size accessible in one cycle.

Unified L 2 cache: two fifty six kilo byte cycles; unified L 3 cache: six megabyte in fourteen cycles. So, it is bigger and very slower. There are couple of things to notice here. That is, you might ask, well, both of these processors have the same number of registers, this one is accessible in one cycle, but this takes two cycles why?

The question is both of these processors have the same number of registers, but here the register file requires two cycle access time, here it requires one cycle access time. Why is that? Any guess? Exactly. Cycle times are different. So, Intel Pentium 4 has a much faster clock frequency. Itanium 2 has a slower clock frequency. That is it. So, the actual time is something same.

If you look at the total size of the register file here, how much is it? So, Intel Pentium 4 the one that we are talking about here is thirty two bit machine. So, its register is four bytes. So, how much is it amount to five hundred and twelve bytes? Accessible in two cycles.

Whereas, your L 1 data cache is sixteen times larger in eight kilo bytes; has the same limit. How is that possible? How will you get? What could be the reason? So at one point of time, we made a statement that in a memory structure, if you have more ports, it slows down. So, can you argue along that line?

How many? What decides the number of ports in register file? Student: It depends on how many ((Refer Time: 09:02). That decides what?

Student: Number of required; the need ports. Who decide the right ports?

Student: The number of …

What is that? What is it depending on? Number of issues, that is all. What about the cache? Number of ports in the cache? Who decides that number of read and write ports in the cache?

Student: Sir, write has only one… only one write is required.

Student: ((Refer Time: 10:03). Yes you are right. So, why you write to the cache?

Student: ((Refer Time: 10:12) from cache you will write back to memory eventually. That is the replacement. That is write to the memory. We are talking about write to the cache. Student: ((Refer Time: 10:27) cache miss. It is a cache miss, then.

Student: You are bringing the new data. Yes, you are bringing the new data at the time you write to the cache. Why did you read from the cache? Student: As the instruction ((Refer Time: 10: 39).

What is ((Refer Time: 10:42) what would be the requirement compared to the register file ports? What would be the requirement in the caches in the ((Refer Time: 10:50) port?

Student: ((Refer Time: 10:53).

Pardon.

Student It will be half of the… Not really half, a very smaller. Student: load instructions. It is smaller. Yes.

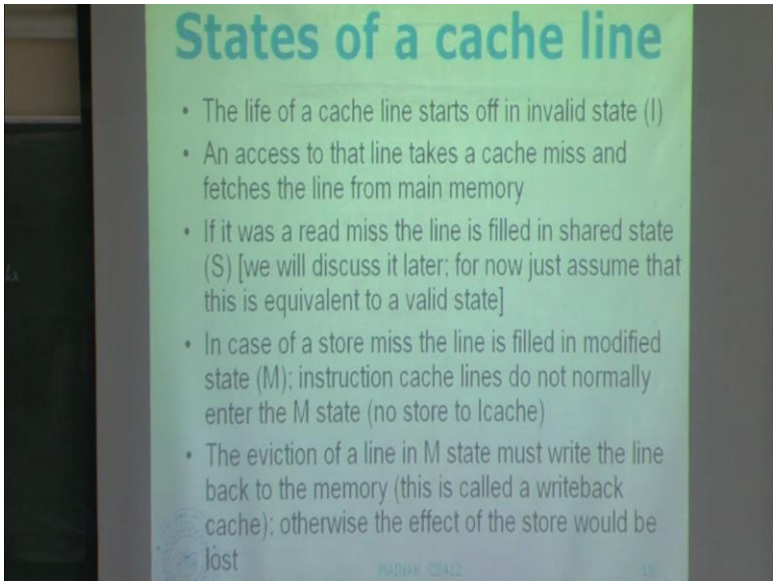
Load and store; both will read from the cache. Store will do some extra work. It will first read the data about, do the modification and write the data back to the cache. So, data cache ports is a subset of the peak issue width of the processor. Then, the peak issue width will get all types of instructions. And, only a subset of that will go to the data cache.

Whereas, the register file will have to accommodate all other instructions. Locate all the instructions. So, it has a much larger number of ports. For example: in Pentium 4, if we assume that the issue may be six, yes right, then your register file requires at most eighteen ports. Twelve read six write. Whereas, your data cache will thoroughly require couple of read ports. Assuming that, you can only send your pair of load store in at most two memory operations every second.

So, that is what is impacting that the latency here. If the register file has the same number of ports, it will have much longer latency than in the cache. If it is just because lot number of ports, the register file latency is smaller. Alright ok. So, that is about the hierarchy. Just notice how the latency increases. You know, usually jumps a lot; five to fourteen L 3 cache. Of course, the size is also much larger.

Today's processors, normally the high end server based processors have much larger L3 caches over there. Although in this course, probably we do not have time. How exactly these very large caches are organized in a big, but just keep in mind that they are very large. They occasionally cross. They routinely cross ten megabytes as often they are more than twenty megabytes cache eventually.

(Refer Slide Time: 13:03)



So, we talked a little bit about states of cache line, when giving an example. So, let us try to open that up a little bit more. So, life of a cache line starts off in invalid state, that is, the lines not in a cache. An access to that line takes a cache miss and fetches the line from main memory.

If it was a read miss, the line is filled in the shared state. We discuss it later if there is time. For now, just assume that this is equivalent to a valid state. So, shared state is just a valid state. In case of a store miss, the line is filled in modified state because you know that you are in modified line. Instruction cache lines do not normally enter the M state because there is normally no store to the instruction cache. Instructions are readable. We need instructions and execute that. That is it.

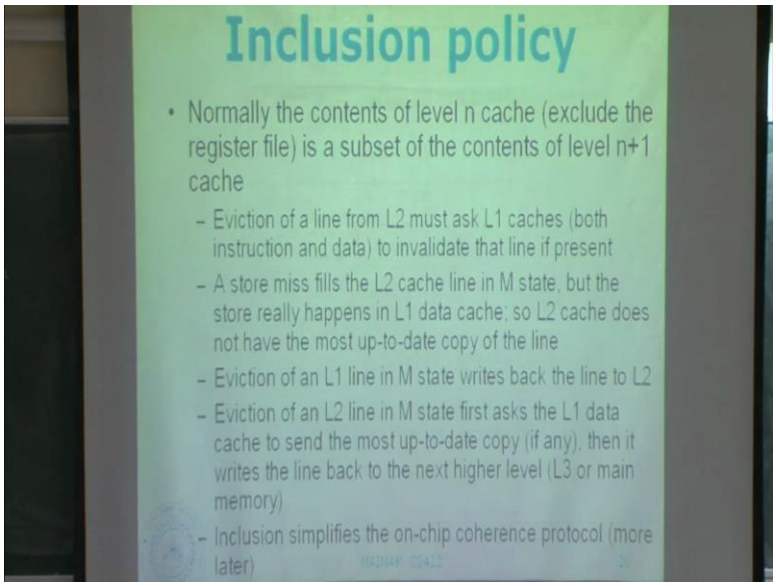
Eviction of the line in M state must write the line back to the main memory. This is called a write back cache. Otherwise, the effect of the store would be lost. So, there is a second type of cache which is called a write through cache. Well, whenever you do a store you not only update your

cache, you update the main memory also. In which case, you would not require the M state because the memory is always up to date. So, you only require two states; valid and in valid. That is all in the cache.

For write through caches there are two places. One is called write allocate; another one is called write no allocate. So in write allocate write through caches what you do is, on a store miss you allocate the block in your cache. And, you do the store in both the places in the cache and in the memory.

In write no allocate, you do not even allocate the block in the cache in the store miss. You only send the data to the memory and update the data in the memory disk. Ok alright. So most of the time, we will be dealing with write back caches. We will touch upon little bit on write through caches. And, in your homework you will actually get to compare write back and write through cache performance wise. You will see why? ((Refer Time: 15:06).

(Refer Slide Time: 15:09)



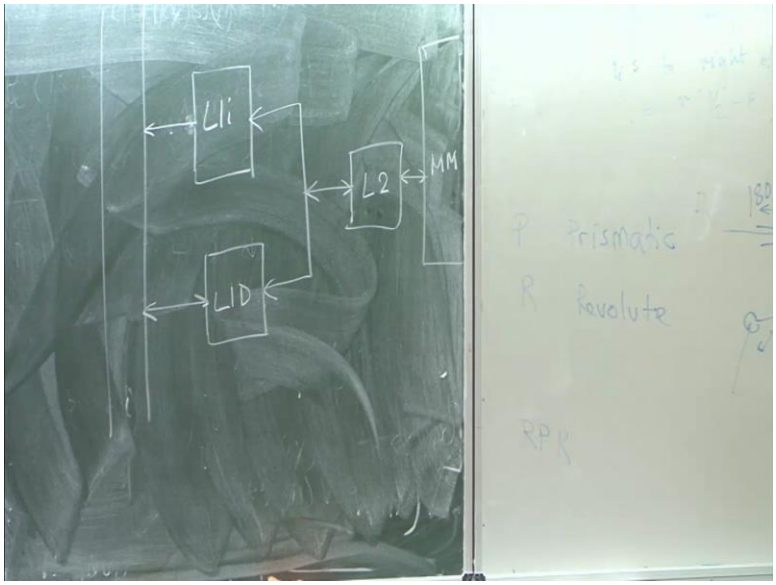
There is something called an inclusion policy in the cache hierarchy. So, how it defines? So, normally the contents of level n cache, exclude the register file, is the subset of the contents of level n plus one cache. If this n value is satisfied for all values of M, when you say that in the cache hierarchies, it increases.

So, essentially what I am saying is that whatever I have in my L 1 cache is directly to be there in L 2 cache, even in case of error. Ok alright. Now, today's processors usually prevent something. So usually it is, part of the hierarchy is inclusive; part of the hierarchy is not inclusive. Ok alright.

So, for example, in a three level hierarchy in Intel's processors, typically L 1 and L 2 will not be inclusive. Ok alright. Their own caches actually satisfy these properties. But L 3 will be inclusive with respect to L 1 and L 2 contents. So anything, the union of L 1 L 2 is guaranteed to be L 3. Alright. So, that is what it really means. But if something is in L 2, I mean, if something is in L 1, it is not guaranteed to be in L 2. For example. So that this property is not satisfied to L 1 L 2. So, but however in this discussion, we will be assume that we are talking about inclusive hierarchy, where the contents of L 1 are guaranteed to be in L 2.

So, eviction of the line from L 2 must ask L 1 caches, both the instruction and data, to invalidate that line if present. Nothing it should be obvious because if you do not, you are going to use this property because line is been evicted from L 2. It must also be evicted from L 1 caches at that time. Otherwise, there will be a line in L 1 cache; which is not in L2. It is valid. Is this clear? Any question? Is it ok? A store miss fills the L 2 cache line M state, but the store really happens in L 1 data cache.

(Refer Slide Time: 17:24)



So, we have L 1 instruction cache, we have L 1 data cache, we have L 2 cache and we have our main memory. So, this is what we are looking at here and this is my pipeline here, the processor pipeline which would access the instructions from the L 1 instruction cache and you read and write to the L 1 data cache. Ok alright.

So, what I am saying here is that, let suppose that there is a store instruction which will follow this path. We will first go to L 1 data cache, then to the L 2 cache, then to the main memory. If you do not find the data, so again I am assuming that I miss in all the caches. So, a store is a store access first go to L 1 data cache; misses here. And goes to L 2 cache; misses here. Reads the data from memory, it fills the data on its written path in L 2 and L 1 data cache, right, both to maintain inclusion.

So, the question is what would be the state of the data that the data block in these two caches. So, what it states is that the data block in L 2 caches will be in M states. Even though the store actually happens in this cache, so when you verify the data, you actually do not notify the new data into L 2 cache. L 2 cache is still having the old data. So alright

So, L 2 cache does not have the most up- to- date copy of the line. The reason is that it is not needed. So implicitly, I am actually saying that the L 1 data cache is a write back cache. Only if it is evicted, that particular block, the new data will be written back to the L 2 cache. Alright. That is what it really means. Ok alright.

So, eviction of a L 1 line in M state lines back the line to L 2 M. This is when the L 2 cache gets the new data.. Until then, it does not have it. Now… because now the question is that you ask why I did this. Then, why should I fill the data in M state in L 2 cache? Even though, the L 2 cache is having the core data. So, reason is this. If the line is evicted from L 2 cache in M state, then you are in trouble because unless if you send the correct data back to memory, there is a chance of losing the new data actually.

So, whenever you evict an M state in to cache block, you first ask the L 1 data cache to send the most up-to-date copy if any. Then, it writes a line back to the next higher level; that is, L 3 or main memory into the… So, why inclusion is important? It simplifies something called a coherence protocol, which we will not at all discuss in this course.

So, there is something called the coherence protocol; where you will discuss a little bit when you talk about two output devices, but not much in detail. So, just keep in mind that there is a reason why this is done. Not that one fine morning somebody decide to do this. any question?

Student: ((Refer Time: 20:50) when whenever we evict the block, then it asks the ((Refer Time: 21:00) yes. So, at that time using the statements of L 1 cache we can modify that. Yes right. But you have to think about the low level implementation details. So, anyway if two cache evicts the block, it needs to know whether it requires the data response or not, from the L 1 cache. It has to eliminate the buffer to put the data. So, the purpose is reservation. Nothing else. It reserves some buffer for filling the data that comes back from L1 line and L 2 data cache.

Student: This is that after every store, we have to modify the statements of L 2 cache. We have to ((Refer Time: 21:38).

Yes. Well, it is actually needed in a multiprocessor environment. That is why. Today's processor is a multiprocessor. And, in multiprocessor you can avoid doing that. If you are doing a store, you have to tell others that we do one store. Others may have the copy of the whole data.

Student: ((Refer Time: 22:03) no, you do not.

Yes. That is what. So, just to clarify what they are actually discussing. Consider a sequence of accesses that processor first loads the data in the valid state in L 1 data cache and also in L 2 cache. Subsequently, a processor wants to write to that particular cache. So, then what has to happen?

What he is saying is that at this point, I first tell that L 2 cache to move to M state; because that is what it actually tells you. That is the invalid. If you have a dirty block in L 1 data cache, the corresponding block in L 2 cache is possibly in M state.

So he is saying that, that it is a bigger one. And, the answer is yes. It is a known one, which you actually cannot avoid in multiprocessors; which I will not clarify here. But intuitively, why you need to do this is that you may have a copy of data shared by two processors. If one processor modifies the value of that data, other must know the modified value.

So, before one processor does a store to that data, you must tell the other guy that this data is no longer a correct value. So, there has to be a notification that has to go. any other question from this? Yes.

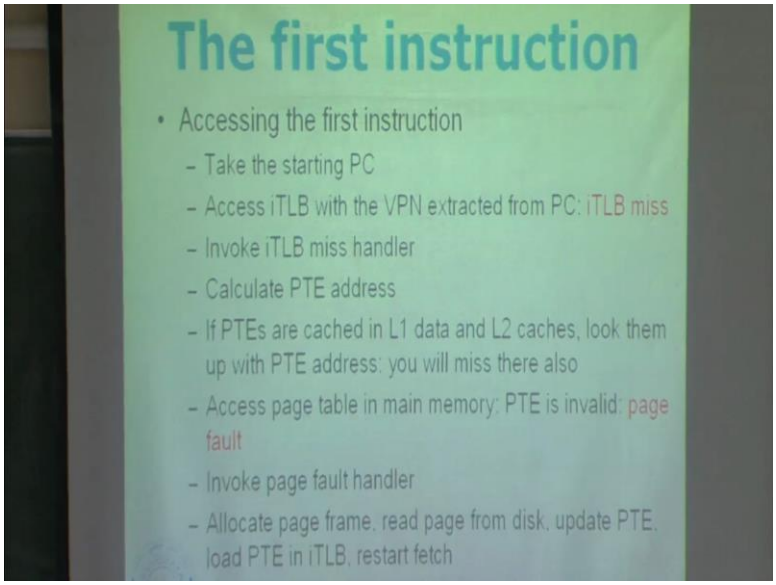
Student: ((Refer Time: 22:31). No because otherwise if the block gets evicted from L 2, the memory may not get update in the correct data. That is what.

Student: you know but eviction of the L 2 line M state first ask the L 1 data cache. But it is not in M state.

yes right. But, what he has suggested? I will again iterate that. What he is saying is that, well, in any case on eviction from L 2 cache, we will ask the L 1 cache all the time for maintaining inclusions, while at the M state. It is just a matter of the low level implementation because if it is in M state, L 2 will know that there is a potential data response that can come from L 1 data cache. So, it may reserve some buffer space to fill that data. Ok alright. So, you just; it is just a low level implementation detail. If you want we go with that, then yes, you are right. We do not need to switch the L 2 cache into M state. We do not need that actually. Any other question?

There are actually many other states, which I am not really talking about; which does complicate matters, but give you some extra benefits like the case that we discuss may actually be handled with some extra states. But, I will not go into that detail.

(Refer Slide Time: 25:01)

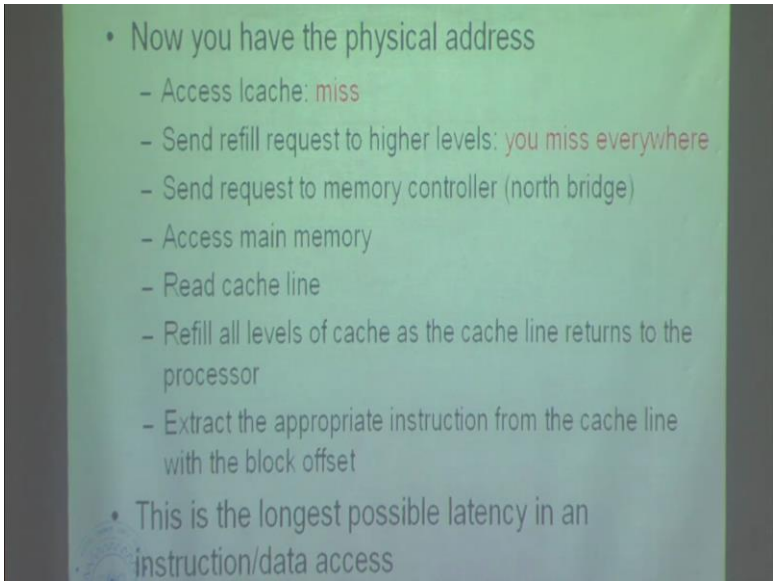


So with that, let us try to trace the sequence of events that happened by the first instruction of a newly started program executes. So, you take the starting program counter. You access the instruction T L B; right because the program counter is a virtual address translated to physical address to access the instruction.

So, we access the instruction T L B with the virtual page number that we extract from the P C. Since it is a first instruction, they will probably have an instruction to a new guess. No need to find cross sections. So, we invoke the i T L B miss handler. The i T L B miss handler has the responsible of calculating the page table entry address. If the page table entries are cached in L 1 data and L 2 caches; this one we discussed little bit last time that you can do this. Look them up with the page table entry address. However, in this case you have to miss there also to do the cross section.

Then, you access the page table in main memory. And, the page table entries going to be valid in this case; which means, you take a page fault. So, we invoke the page fault handler, allocate the page frame, read page from disk, update the page table entry, load the page table in an instruction T L B and restart the instruction fetch.

(Refer Slide Time: 26:21)



So, now we have the physical address; because this time of course, the page table will get the translation from i T L B. So, you can access the instruction cache which is going to be a miss. Instruction you all have it to the cache. Send the refill request to higher levels. We will miss everywhere. So in this case, we follow this part. You miss here, then you look up L 2, you are going to miss here. So, till memory. And, you send the request to the memory controller, which is to be called a north bridge, long back. Now, it is not really relevant anymore because memory controllers are now inside the processor chain. So, what guarantees that this particular instruction page will be found in memory? How do you know? Because I say that the next step is your memory controller has a passive device. It gets a request, it gets an address, it sends the request to the memory as I said. What guarantees you that this particular page is in memory correctly? Yes. How you know that the data that in the memory will return to the memory controller and will be correct? because at least, fine, there is no check. Memory controller gets an address and forwards it to the memory. That is it. Answer is already discussed.

Student: ((Refer Time: 27:48)

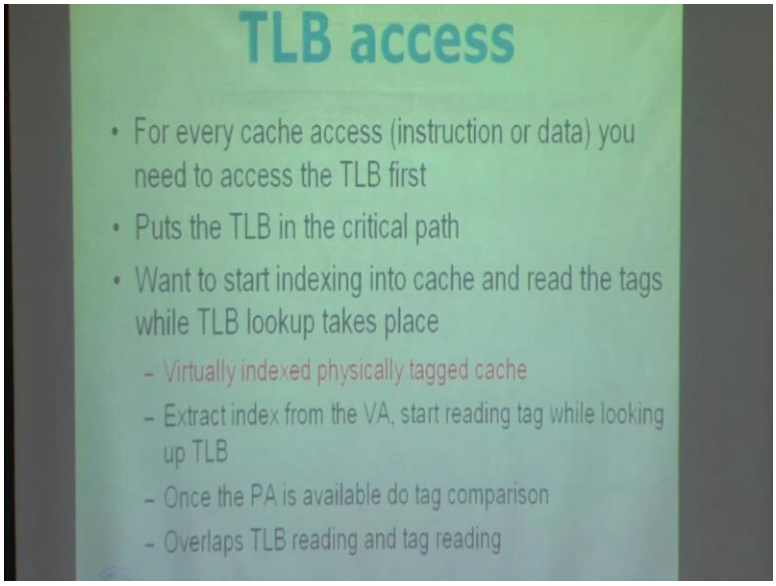
In the last line.

Student: ((Refer Time: 27:54)

Exactly. So, this particular action makes sure that the data will be in the memory. Ok alright. So, you read the cache block from main memory. So, now the cache block comes back and while coming back it will fill the L 2 cache, it will fill the L 1 instruction cache and return the particular instruction or data to the processor.

So, yes, exactly I have to fill instruction from the cache line with the block offset and then the pager can start. So, now the processor has instruction. Finally, they decode and execute. Right. So, there is a longest possible latency in an instruction data access. This is a sequence step to do; probably what happens. Yes.

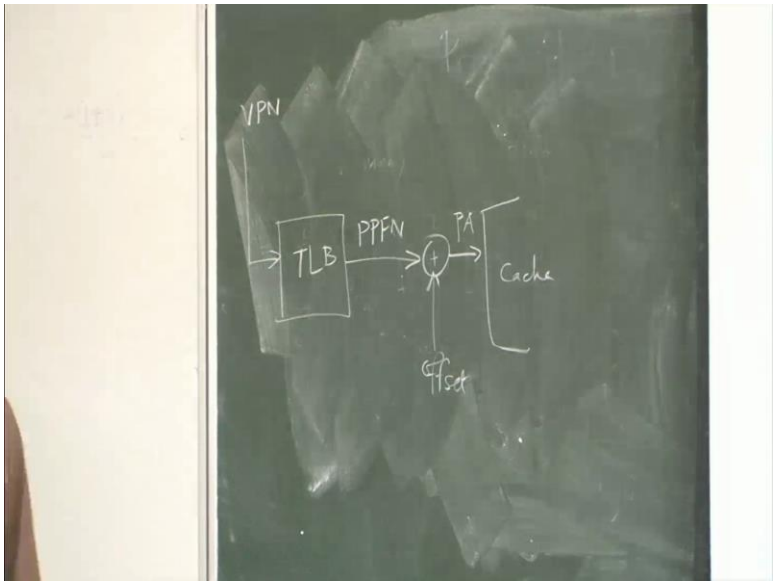
(Refer Slide Time: 28:59)



So, little bit more on the T L B access. The important observation to make here is that for every cache access instruction or data, you need to access the T L B first. So, that is what you see in your physical address. So, it puts the T L B in the critical path of every instruction. If you have a slow T L B, your instruction execution is very slow.

So if you remember, the point is that you take a virtual page number, look up the T L B in data physical page frame number, you ((Refer Time: 29:38)) that with the page offset to get a physical address. And then, you can look up the cache. So, that puts your TLB in the critical path; right because the cache access cannot be done until the TLB access is completed.

(Refer Slide Time: 29:51)



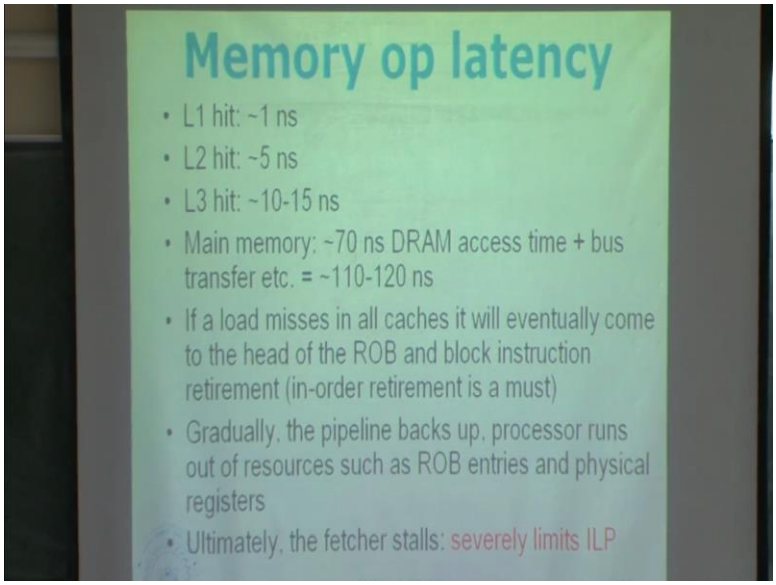
So, this is the physical address. So, I put a plus here. This is the ((Refer Time: 30:17) So, is this correct? So, this is the sequential operation. I look up the T L B and the cache in sequence. There is no overlap. So, what I ideally want is to start indexing into the cache and read the tags, while T L B look up takes place. So, I want to do this look up and this look up in parallel. How can I do that?

So, the only way to do it is to index a cache with the virtual address. It has to begin with; I only have the virtual address and nothing else. So, the question is can I use one of the virtual address to index the cache. And, I can have my tag in the cache. Then, I forget the physical address because by the time I look up the cache in the tag comes out, I would have the physical page frame number for the T L B, so that I can now do a tag comparison.

So, this is called virtually indexed physically tagged cache. So, that is what is used today in all commercial processors, at least for the L 1 cache. Ok alright. So, we extract index for the virtual address, start reading tag by looking up the T L B. Once the physical address is available, you do the tag comparison. So, it overlaps the T L B reading and cache tag reading. Is it clear?

So, it relaxes these latency a little bit more because now you can afford to make it clear that it is small; even can afford to make the cache a little bit slower, but still not using because now these two will go in parallel.

(Refer Slide Time: 31:59)



So, here is the typical latencies in a memory hierarchy. Do not take these as authentic numbers. These are just examples. Just to show you what the caches are. So, everyone hit latencies about nanoseconds today. L 2 hit latencies about 5 nanoseconds; L 3 is about 10 to 15 nanoseconds. Main memory is about 70 nanoseconds D R A M access time plus bus transfer etcetera. So, gives you about 110 to 120 nano seconds. Ok alright.

And, if you have a more complicated system, things may value more. So, here I am talking about variance of about 10 nanoseconds; variance of 10 nanoseconds. The gap between minimum and maximum may be even larger. So, point here is that there is a very big jump from L 3 to main memory. ok alright.

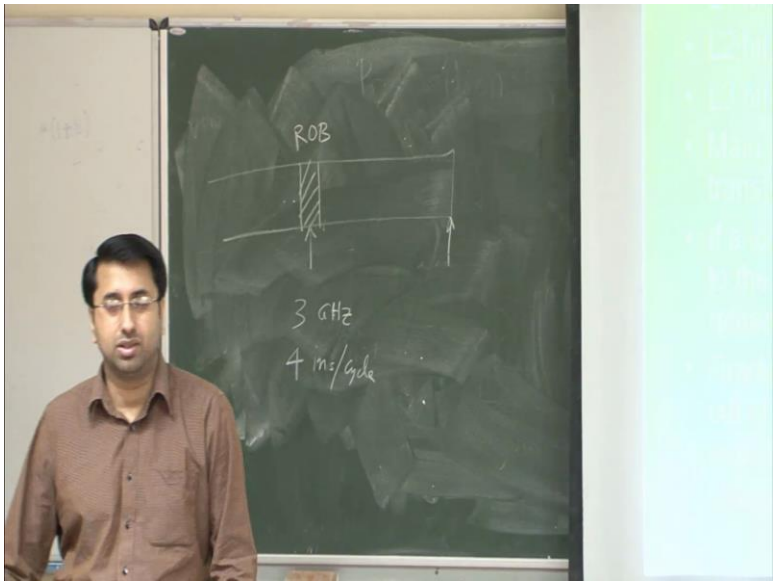
So, L 3 cache is at the last level of the cache. This is your last line of the cache. If you miss there, you are going to take a very big performance here. That is pretty obvious. So, your last level cache should be very intelligent; should be able to identify which data blocks are important, should retain them and make room for them by pulling away the useless blocks.

So, it has to do some form of a prediction by looking into the future in some way and say that, well, I think this block is used in future, so I will keep it. That block does not look like to be useful, so I will throw it away and make room for some other useful blocks. So, your cache

controller has to be very smart. So, it should be able to do these things. And, it is a very ((Refer Time: 33:51)) because you can see actually why it is. There are places for that.

If your last level cache is a down cache, your processor is not going to be a hot research topic. That is pretty obvious because this jump is enormous actually. So, just to give you the problem little bit more completely, if a load misses in all caches, it will eventually come to the head of the ROB. right. You have the ROB. right.

(Refer Slide Time: 34:20)



So, this is my ROB. So, this is the head. So, this is where my retirement is currently running. I am retiring instructions from here. And, let us suppose there is a load here, which is currently executing. This one is currently looking up the cache. Alright. So, this particular load instruction looks up the L 1 data cache, misses in the L 1 data cache and looks up the L 2 cache. Remember that in the meantime, the ROB is progressing. The head is actually moving gradually. This retires instructions.

So, the point is that if before I get the data for the load, if this head moves down here, I have to stall because I cannot retire the load at this one. So, now not only I can retire the load, I cannot retire any of these instructions because ((Refer Time: 35:13) has to be loaded. So, that is a big problem; which means, I have only this much of time to get the data for the load to make sure that the processor does not stall.

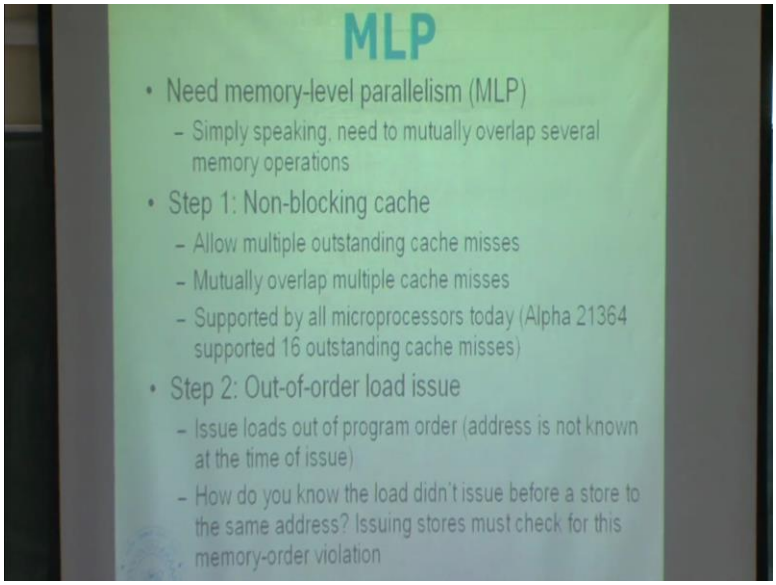
Now, if a particular load instruction misses in all the caches and it has to go to memory, you can imagine the large amount of time that it has to take. So, we did a calculation last time. I should remind you about the calculation. Suppose, your processor runs at 3 giga hertz. Let suppose that you retire 4 instructions per cycle, so what does it mean? So, your cycle time is one third nanosecond. So, if I assume that let us say; let us be optimistic; let suppose that we have a hundred nanosecond memory access in the memory time, so within hundred nanosecond, let us assume that I will get the data back.

So, that is essentially how many cycles? Three hundred cycles, I retire four instructions in every cycle. So, I need an ROB of ((Refer Time: 36:28)) hundred, right, to be able to hide the vacancy of this load; which is impossible. Today's ROB sizes are two hundreds; may be two hundred.

So, this is the problem. Why loads impose the problem? And, this is that reason; we cannot retire anything, here also important. And, if you have an ROB of length hundred, it will take only twenty five cycles to get your stall point; which is a very small fraction of three hundred cycle latency.

So, this is why today actually if you look at applications that access big data, the large amount of data. Here, numbers like ninety percent stall time. This is the reason. So, one way of resolving it is to have the smart last level cache. You make it smart enough, so that it can retain important data. So, you would not have to go to memory often. So yes... So, gradually the pipeline backs up, the processor runs out of resources. And, ultimately the fetcher stalls; which severely limits I L P. so, that is the basic problem.

(Refer Slide Time: 37:52)



So, we will talk about some of the simple solutions here. That the processor industry has adopted over time to alleviate this particular problem somehow. So, essentially what you need is memory-level parallelism. So till now, we have been talking about instruction in parallelism; where you find out instructions which can be executed in parallel. Let them execute in parallel or you offer resources and ROB. So, the same question you can ask about memory approaches. So, can I execute multiple memory operations in parallel? That is what memory level parallelism.

So simply speaking, you need to mutually overlap several memory operations. Why is that useful, because there if you have two memory operations executing concurrently, for both of them we will see a latency of hundred nanoseconds. It is not like two hundred nanoseconds. Not that one. You can overlap in time.

So, how do you achieve that? So, first step is to have a non-blocking cache. What is that? You will allow multiple outstanding cache misses. That is the first requirement. That is, you cannot now say that whenever my cache gets a first miss, it is going to reject all subsequent requests, until this miss is dissolved. Then, of course you are not going to take any memory level hierarchy. That is all about this question; because the cache itself is acting as a blocking agent. That is why…

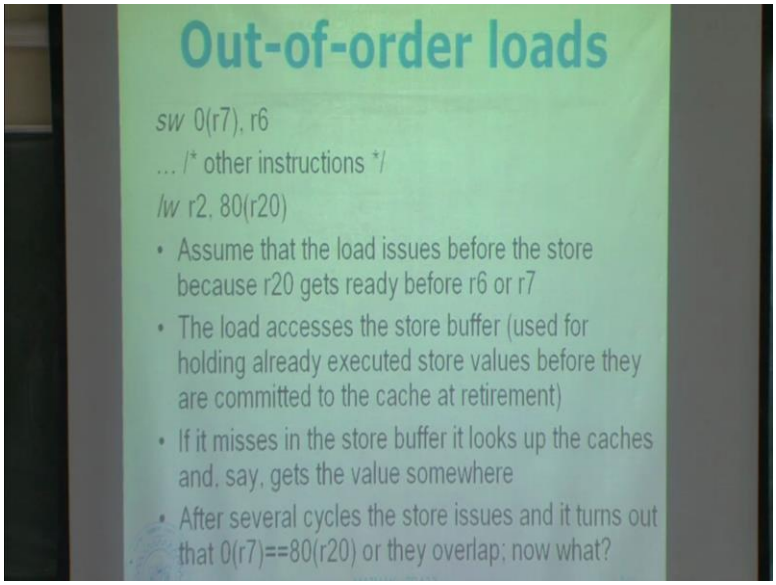
So, this is the first requirement that the cache must allow multiple outstanding cache misses, even when it have bunch of misses outstanding, which is still be able to access more requests. Allow that to go to the cache. Maybe ((Refer Time: 39:30)) alright. They should proceed.

So, you visually overlap multiple cache misses supported by all microprocessor today. For example, Alpha 21364 supported 16 outstanding cache misses. So, today the numbers are roughly around this. How many outstanding caches that you can support? The reason why there has to be a limit is because we have to remember somewhere that what are the reasons that are still outside with respect to a table. And that limits the number that you can actually…

And, this table is actually pretty much in the critical path. So, that is why it has to be small. It cannot be very large. So, is that is this solution clear to everybody? That is the first step; that you have to wait in the cache. Sorry, you cannot have a normal cache.

The second one is out- of- order load issue. That is, issue loads out of program order. This one also we have disused earlier. The address is not known at the time of issue. So, what you do is you first; the first page of the load issue actually computes the address, comes back and computes this address to the store before it. Alright ok. Then only, you can go and access the memory. So, how do you know the load did not issue before the store to the same address? Issuing stores must check for this memory-order violation. So, let us talk about the memory in detail.

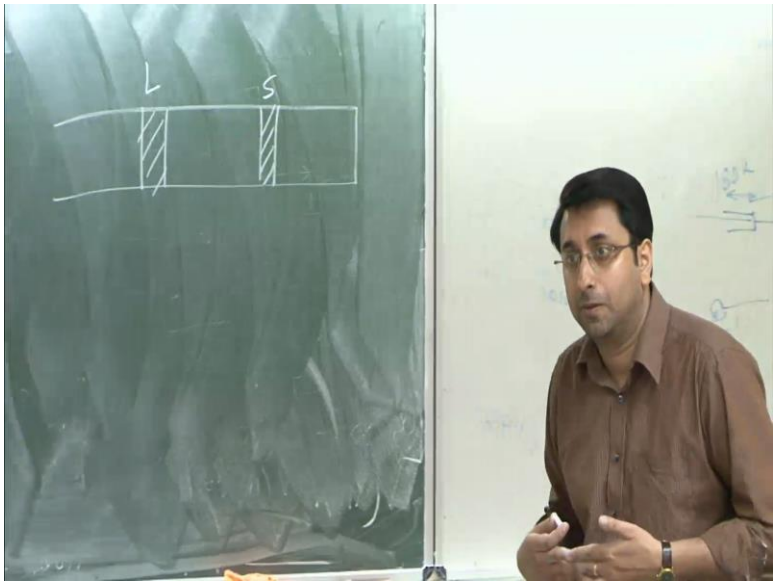
(Refer Slide Time: 40:56)



So, here in the example, I have a store here and then bunch of instructions and then I have a load. So alright. So, let us assume that the load issues before the store because r 20 gets ready before r 6 or r 7. For the store to issue, I need to make sure that these are available; r 6 and r 7. For load to issue, i will make sure that r 20 is available. So, let us assume that whoever was generating r 20 completed first and so now the load can issue.

So, load access to the store buffer. So, this is essentially the value filled in the store tree. I am talking about. Using this for holding already executed store values, before they are committed to the cache at retirement. If it misses in the store buffer, it looks up the caches and say, gets the value somewhere. After several cycles, the store issues and it turns out that these two addresses are actually same. So or they overlap.

(Refer Slide Time: 42:06)



Now, the thing is the load must have got all values. So, maybe I have forgotten. So, let me try to remind you. So, let us suppose that we have the single issue. And, this load instruction here is this one. Let us say this is a load. Let suppose this is the store; that is store instruction. So, previously we said that the condition for a load to issue is that all the stores before it must have completed their addresses. That is what we said. Now what I am doing is I am going to relax that because that we have already said that this looks very conservative; because many of the loads will not depend on any of the stores, where you delaying the issue of those loads..

So, now saying that, well, I do not care. I will issue the load as soon as this operand gets ready alright; which means, some of the stores may not have executed yet; which means, some of the stores may not have completed their addresses. So, what does the load do when it issues? It first computes its own address, comes back, and compares its address with all the stores before it. And here, we are assuming that it does not match with anybody. And, the reason is that this two has not yet executed. Ok alright. Because r 6 or r 7, one of them or both of them may not be.

So, then the load goes, happily accesses the cache and gets the data somewhere. Ok alright. It supplies the data to its dependents that are sitting here. They also start executing. Eventually r 6 and r 7 will get ready. The stored issues executes. And you find that these addresses are actually same. So, load has not only captured the wrong data, it has supplied the wrong data to many

dependence after it. How do you recover from this error? That is the problem, any suggestion? Is the problem clear to everybody?

So, we are; now the problem arises because we are trying to issue loads very aggressively out of all. So, how do I fix it? So, in the previous slide where I have a comment here; issuing stores must check for this memory- order- violation. ((Refer Time: 44:30)) the load instruction still inside the processor somewhere or has it retired? Yes

Why?

Student: ((Refer Time: 44:42)).

Exactly. So, retirement has to be in order. So, the load has not got escaped. So, you can cache it and fix it. How?

Student: ((Refer Time: 44:59)).

Has loaded from ((Refer Time: 45:05)).

So, issuing store should check all loads after it. That have already executed. And, do a comparison of addresses. And, if you have a multiple matches, what should we do? We have a multiple loads executed in the same address. Student: it is all of them. It is all of them. alright. So, by fixing what you need? Can you elaborate?

Student:((Refer Time: 45:40)) overwrite the value. ((Refer Time: 45:46)).

Change the value. That is all? Instructions which are supposed to read those registers have already read and executed. What about those instructions?

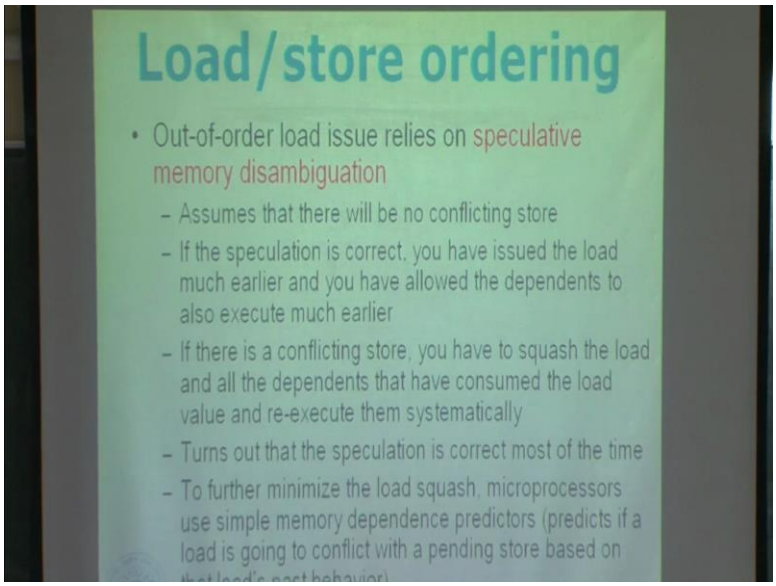
Student: ((Refer Time: 46:00)).

So, you have to also find out the dependents of these loads, which are executed. So, there are two modes of fixing. So, first step is of course that which means store; must check all subsequent loads that have already executed. By checking I mean, compares the address. Of course, an issuing store cannot compare address. First it issues, computes its address and then only it can do that.

So, it checks all the loads. The simple solution is that it picks the oldest load that matches and removes all instructions out of it. So, it re-executes everything after that. So, of course it will do some extra amount of work. It actually re-executes many instructions which do not depend on these (refer Time: 46:46) loads. But, it is a simpler solution. You do not have to keep track of dependences.

A slightly more complicated solution would actually maintain the dependency of each load. So starting from the load, you can think of it as a root. It will actually store the value to bunch of instructions to form a tree. You keep track of the tree. So, that is what todays Intel processors actually do. They keep track of the tree at every load to the buffer. So, that actually minimizes your ((Refer Time: 47:13).

(Refer Slide Time: 47:18)



So, that is precisely what the solution is. So, this is called speculative memory disambiguation. So, essentially we are doing one form of speculation. Why did you issue the load? Because we are speculating that this load would not have any problem; would not even conflict.

So, computer architects are very optimistic about whatever they do so. Here again, we are being optimist and we are saying there will not be any problem on issues. The good news is that it is correct most of the time. That is why it walks. Otherwise, of course ((Refer Time: 47:50)).

So, assumes that there will be no conflicting store. If the speculation is correct, you have issued the load much earlier and you have to allow the dependents to also execute it much earlier. Ok alright.

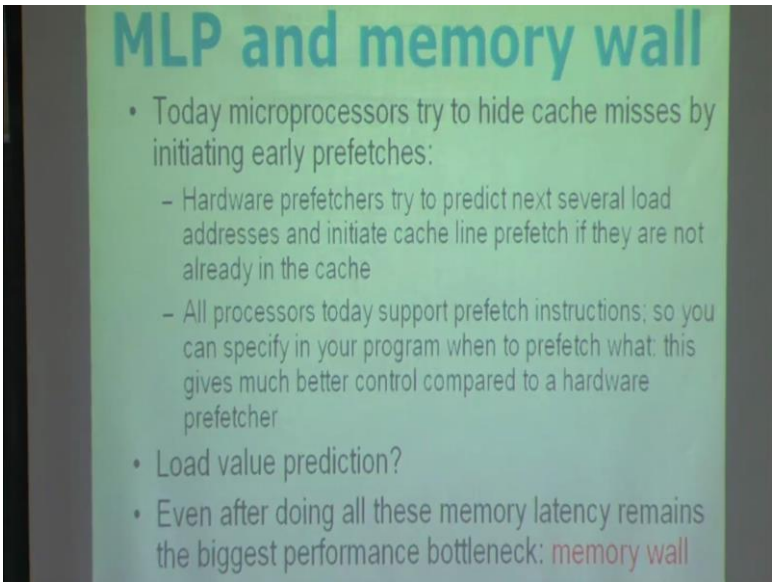
If there is a conflicting store, you have to squash the load and all the dependents that have consumed the load value and re-execute them systematically. Turns out that the speculation is correct most of the time. And, often this is called a blind speculation because you are not really using any property of the load. You are oblivious about the history of this load because in the past, this load might have conflicting in the same store. Ok alright. But, here you are not caring about that. You are saying that I will do a blind speculation; I will issue this load whatever may be its history. Ok alright.

So, you can improve upon that. So, that is what todays processors do. They use simple memory dependence predictors, which predict if a load is going to conflict with a pending store based on that load's past behavior.

So, we can actually make association between the pair of loads and stores. You can store the association in the table, then figure out in future that this load actually is going to conflict with any of the stores that are waiting here. If it is, then you would not actually issue this one. You will wait until all the stores have… or at least till the predicted conflict in store has gone. Ok alright.

So, this is what the processors do today. I already discussed about that actual predictor implementations. If you want to read about that let me give you some papers. So, they look very much light branch predictors. What they have to be little smarter; because here we are talking about establishing the association of the pair allowed in a store, essentially. But, still the queries are binary query. So, guess what answer will come. Ok alright. Just like a branch predictor. You take a load and ask tell me if this, if it is safe to issue this one.

(Refer Slide Time: 50:00)



So, today microprocessors try to hide cache misses by initiating early prefetches. That is another solution. Hardware prefetches try to predict next several load addresses and initiate cache line prefetch, if they are not already in the cache. So, I hope your noticing that there are many labels of actual predictors that go into a processor. That varies actually.

So, this is again another pattern of predictor that minds the pattern of addresses that are accessed by the cache and tries to predict the future. Then, tell me what address are going to come in future? Can I fix them now? So, by the time it is needed, during the cache. Alright

All processors today support prefetch instructions also. So, you can actually specify in your program when to prefetch what. So, the compiler can actually insert these instructions that actually take places. This gives much better control compared to a hardware prefetcher because here you can actually see the whole program and also you can maintain the semantic of the program. For example, you can see that it is an array access. So, you know that it is going to be very predictive. You can access in sequential error locations. The addresses are extremely predictive. Whereas this guy, the hardware prefetch is a finite state machine. Sitting there, monitoring addresses and just making your prediction; a plain prediction. So, there is a bigger chance that it may be wrong. But, of course there is an advantage that it can see all the addresses, but the compiler cannot. For example, ((Refer Time: 51:22) addresses only visible to the compiler. Ok alright.

Another solution that the researchers have explored is load value prediction. That is, I am loading a particular piece of data right here, can I predict what this load is going to become; the value. Ok right. So, that also has been explored.

So, much difficult problem; because you are not going to predict the 32 bit or 64 bit value. So, entropy is expected to be much higher. It is not very easy and accuracy is very low. Which is why, the industry has not yet adopted this particular solution. But, there is a large ((Refer Time: 52: 04) how to predict the load values. And, the good thing is that the programs often load constants. Lots of constants are loaded often. And, the most loaded constant is zero. So, these values can predict extremely ((Refer Time: 52:21).

Even after doing all these things, the memory latency remains the biggest performance bottleneck and that is called the memory block. So, it is a very hot research topic and probably will remain a hot research topic for a long time because there is no easy solution visible in future. Processors are expected to get faster; memory is not getting faster at the same rate. So, the speed gap is only increased gradually. So, a ninety percent stall time may become ninety five percent stall time in future. So, I am going to stop here. The next time we will go deeper into caches.