Advanced Computer Architecture
Dr. John Jose
Department of Computer Science & Engineering
Indian Institute of Technology Guwahati

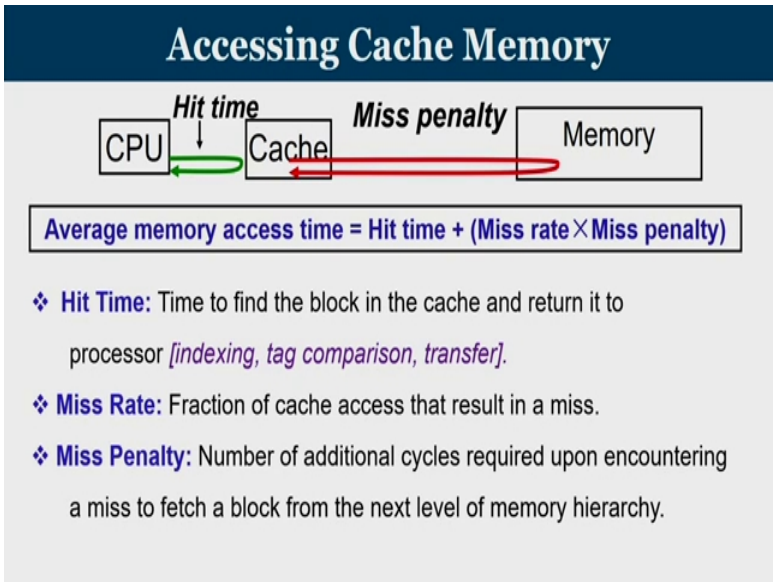
Lecture No. – 16
Advanced Cache Optimization Techniques

Welcome everybody to lecture number 16. In lecture 16, our focus of discussion is on advanced cache optimization techniques. In the last lecture, we had a quick introduction of what is an optimization technique in cache. As you know, cache is working along with the processor to deliver instructions and data in time.

Reducing average memory access time is a very important and critical parameter as far as efficiency of a processor is concerned. We have learned a few techniques in the last lecture, like how can we reduce average memory access time by coming up with techniques that can reduce miss rate, miss penalty, and hit time.

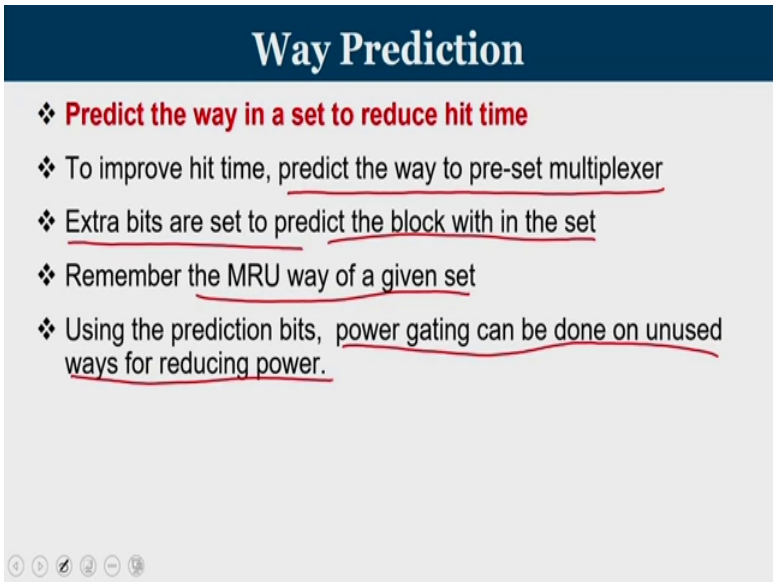
Today, we will learn a few advanced cache optimization techniques. They are not that simple and trivial in nature. A little bit of extra circuitry is built on cache such that these three parameters of miss rate, miss penalty, and hit time are manipulated. A quick recap of what is advanced cache optimization. Prior to that, let me introduce you to what is average memory access time in caches.

(Refer Slide Time: 01:58)



We have seen that hit time and miss penalty are two important times that govern average memory access time, with miss rate also coming into picture. And hit time is the time to find the block in the cache and return it to processor, whereas miss rate is a fraction of cache access that results in a miss, and miss penalty is the number of additional cycles required upon encountering a miss to fetch a block from next level in the memory hierarchy.

(Refer Slide Time: 02:23)



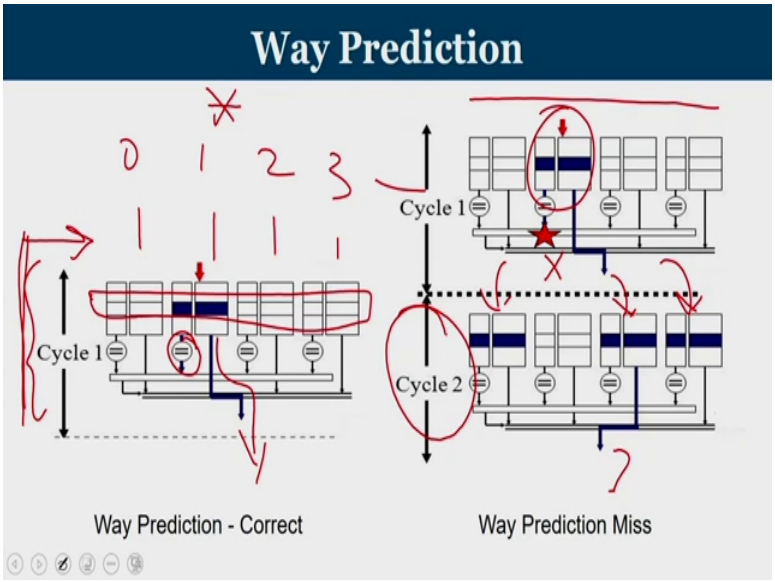
So, we are going to learn about a new technique known as way prediction, predict the way in a set to reduce the hit time. Let us try to understand what way prediction is all about. We know that direct mapped caches have the lowest hit time because only one way of a given set is being operated on, and if it is a hit, the corresponding data is extracted.

When you go to set associative cache, even though the tag comparison is parallel, at the end the place where hit occurred you have to transfer data only from that particular cache line. So what we do in way prediction technique is we can inherit the property of a direct mapped cache in terms of hit time and, at the same time, the underlying cache is actually a 2-way or a 4-way associative cache.

So, we are going to have a technique by which we pre-select the way to one of the particular way in a given set with the help of a multiplexer, so that we are going to use few extra bits here and these extra bit will predict which among a block in a given set there is maximum probability for a hit.

So, what we do is remember the most recently used way of a given set that is one way of looking at it and it is used for prediction and using these bits you can even do power gating. Power gating is a technique wherein we are going to temporarily keep certain logic circuits in a low power mode.

(Refer Slide Time: 03:54)



So, consider the case that you are going to work with a 4-way associative cache. So, these are the four different lines in a given set. Let us say our focus of attention is in which particular set and we are predicting that. This is way 0, 1, 2, and 3. Let us say I am predicting that way 1, there is a possibility of a hit. So, the tag comparison will work only for way 1. That is what the tag comparator circuit tell.

And think if it is a hit, then without looking into any other we are extracting this value. So, it acts like a direct mapped cache. At the same time, we are going to get the benefit of a multi-way associative cache. What is the benefit of multi-way associative cache? We could store more number of lines in a given cache set. So, if the prediction is correct, then our hit time is very less, as that of order of a direct mapped cache.

Now, if prediction goes wrong, this is the case where prediction goes wrong, that is on the right side. We tried in this way as per the prediction information, but it resulted in a miss. That does not mean that this particular address is a miss in the cache. In the next cycle, the remaining three ways have been parallelly searched to find out a hit or not. So, we are adding a few extra bits to facilitate this prediction.

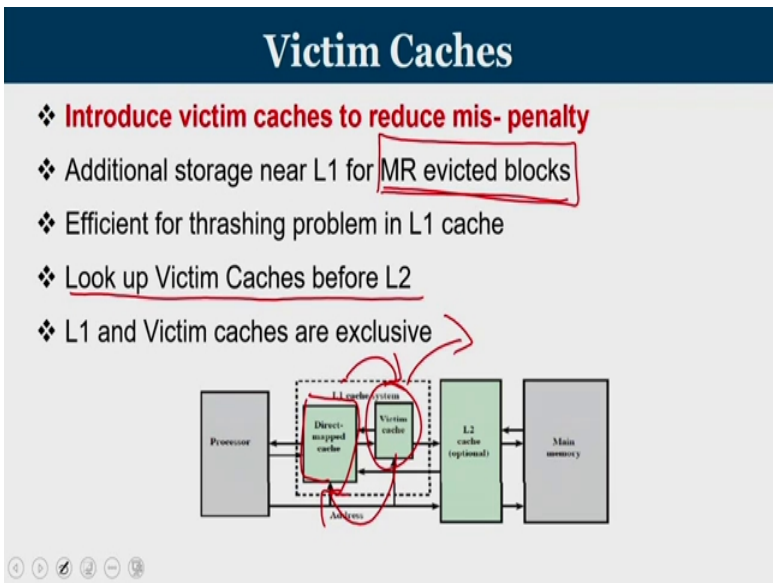
If production is good, or if production goes correct, then, with one cycle we can complete this information. If prediction is wrong, we are using one more cycle, that is an extra cycle. There is a penalty that happens if the prediction happens wrong. So, where prediction will help, for example, if you are going to work inside a loop, what happens if you are going to work inside a loop, think of a case that one particular loop is traversing across 10 different sets.

Now, in all the sets there are certain blocks or a cache line which is used. Now, if you are going to repeat inside a loop, for example, way number 0 of set 1, way number 3 of set 2, way number 2 or set 3, like that, so the same pattern is being repeated as this loop is going to run for its multiple iterations. So, the remaining ways I am not going to use.

Wherever it is, even though it is a 4-way associative cache, in the natural case in all the four ways the tag comparison happens, and only wherever there is a hit the data is being extracted. But, by using certain extra bits we can tell what was the most recently used in its last iteration and that can be used in predicting what will be the possible access in the next iteration as well so that the remaining ways can be temporarily kept in a shut-off mode.

And that is what is known as power gating. So, if you go for a 4-way associative cache, the hit time maybe more, but now the hit time is reduced because you behave as if it is a direct memory cache. If the prediction is wrong, then you require an extra cycle to perform the tag comparison in the remaining ways.

(Refer Slide Time: 06:44)



Now, we will go to victim caches. Victim caches are used to reduce the miss penalty. Let us try to understand what is the context of a victim cache. It is an additional storage near L1 for the most recently evicted blocks. It is very efficient for thrashing problem in L1 cache and it acts as a look up before the L2 cache, and L1 and victim caches are exclusive. See from this diagram.

This is your L1 cache and there is one more storage unit, it is an extended L1, maybe we can tell that. The evicted blocks from L1 they are kept in the victim cache. So, when there is a miss that happens in your conventional L1, you go and search in victim cache prior to have a look up in L2. If you get somebody from victim cache, you promote him to the L1 cache and throw out somebody else.

So, the most recently evicted block that is what is being mentioned here, most recently evicted blocks are kept in the victim cache, it is a FIFO queue. So, when somebody who is present in the victim cache is not re-referenced again, then eventually it may be pulled out from victim cache and then you are going to write back into L2. Let us try to understand what is the context of making use of victim caches.

We know that L1 has a limited capacity. Let us say the incoming address sequence from the processor is in such a way that all of them cannot be accommodated in L1, then that will lead to replacement. Think of a case you are going to just flush out a block, it has been evicted now because of an incoming miss. What happens in this miss, I am going to bring a new block to everyone.

And because of this operation one block has to go out. Now, imagine that in the near future there is a request to this block which is evicted out. So, somebody who is just thrown out is needed in the future. So, in the conventional cache what happens is you go to L2 cache and bring that block. Now, that may actually throw out somebody else, and just imagine that whoever is thrown out they are being demanded in the near future.

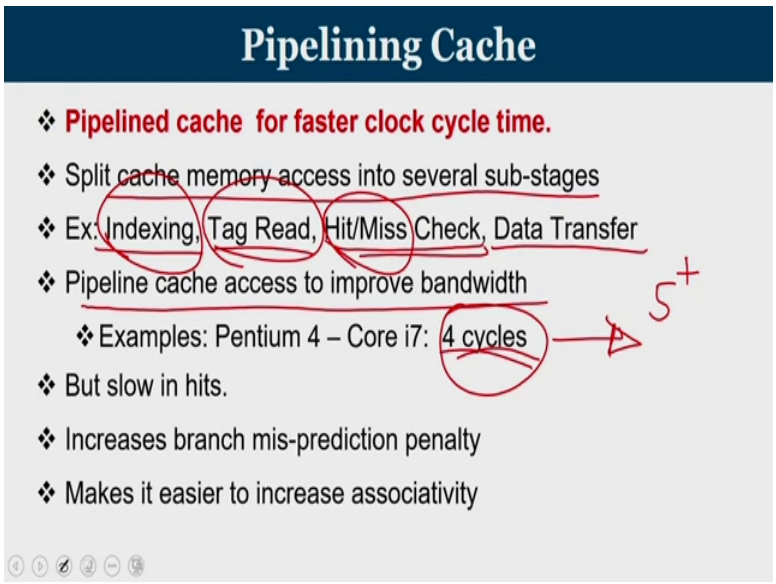
So, why cannot I have a small unit outside L1 where all the thrown away blocks are being kept. It is a small FIFO queue which can accommodate, say, four blocks or eight blocks like that. So, when there is a miss in L1 you have to understand that it might be recently evicted

out from L1 in that context, it may be present in the victim cache, and order of access of victim cache is lower than the access time for L2 cache.

So, this victim block acts as a flush out entry for L1 cache. So, searching in victim cache sometimes if it is a hit, it is much beneficial than going and taking the block from L2 cache. In this way, victim cache is going to reduce the miss penalty of L1. Normally, when there is a miss in L1 you go all the way to L2 and bring it.

Now, when there is a miss in L1 there is a probability that you can find it from the victim cache and if there is a miss in victim cache as well, then you go to L2 and then bring it. In this way, when there is a thrashing problem, what do you mean by thrashing problem, whenever all the access are going to result in a miss because they are not present or the working size is larger than the number of blocks, then we call it as thrashing. To mitigate thrashing victim caches are going to be helpful.

(Refer Slide Time: 10:08)



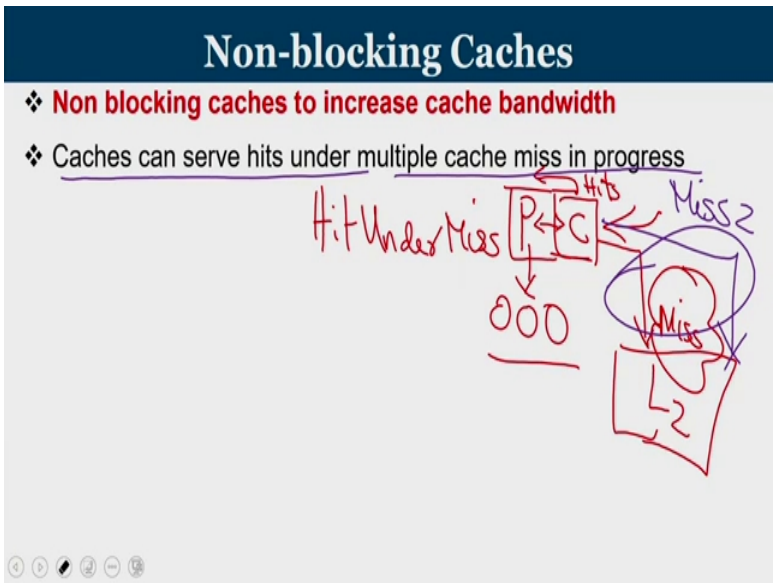
The next technique is pipelining the cache. You know that as far as the cache is concerned, there are different sub-operations. Split the cache memory access into several sub-stages, like you index using the index width. You have to find out what is a set number using a decoder. And then, the next operation is read the tag that is present in the corresponding set, and then compare the tag that is called hit or a miss check, comparison of the tag that is stored in the cache line with the tag of an incoming address.

And if it is a hit, then you are going to perform data transfer. Now, we are going to pipeline the cache access to improve the bandwidth. So, when you are performing the hit or miss check for one set of address you are reading the cache for the next address and your indexing into the cache for the third address, just like the instruction pipeline principle that we have learned.

Here, we are subdividing the entire cache operation into four or five sub-operation, independent sub-operations, and we can do it in a pipelined manner. Pentium 4 to Core i7 they basically use 4 cycles of cache access, meaning, the entire operation of cache is divided into four sub-components, but it is going to be very slow as far hit is concerned and it will increase branch mis-prediction penalty.

Generally, you are going to have your branch determined in the very second cycle. Now the four cycles itself is gone for cache access. So, ideally, checking of the branch will happen only in 5th clock cycle or even larger than that. By the time already for instructions are in pipeline. So, if it is a mis-prediction, it is going to have more amount of penalty and it makes it easier to increase the associativity also, because then you have more number of cycles and you are not in the critical path.

(Refer Slide Time: 11:59)



Next, we move on to non-blocking caches. Non-blocking caches are used to increase the cache bandwidth. Cache can serve hits under multiple cache miss in progress. So, consider the case that you have a processor and you have a cache. Now, the processor is going to

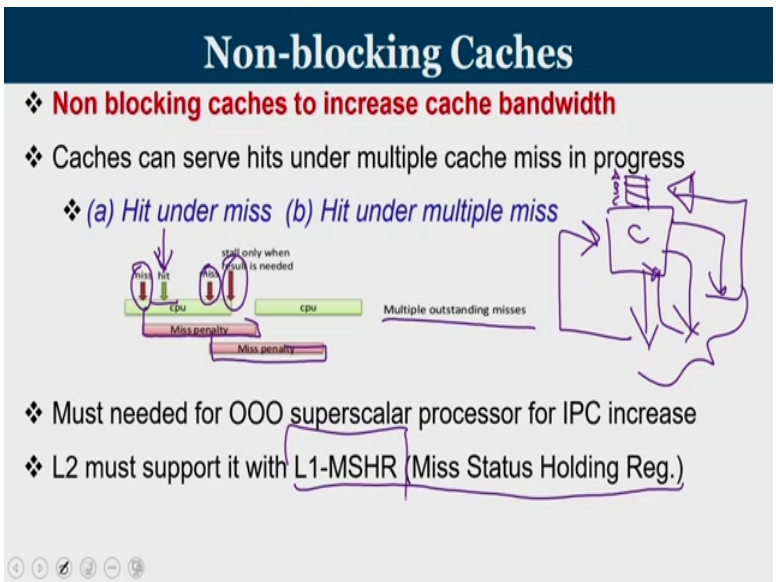
interact with the cache. Imagine that you encounter with a miss. So, this miss is being processed in the next level of memory, let us say it is an L2 cache.

When the miss is in progress will the processor stall or processor will continue? How can processor continue if the processor is out of order execution, when it cannot run the current instruction due to an instruction miss? It looks up in the remaining instruction if any of them is independent of the current instruction that resulted in the miss. If we have an instruction that can be run or it is non-dependent on a missing instruction, then we call it as out of order execution, processor runs it.

So, still your cache can supply the hits. This scenario is called already a miss is under progress, we are entertaining hits, and that scenario is called hit under miss. So, cache can actually supply instructions that are hits when a miss is already in progress. Now, imagine that rather than a hit, let us say the subsequent access also resulted in a miss, that is the second miss, even that can also be supported. That is called miss under a miss. You are going to permit one more miss.

So, you need to have a support mechanism which will provide you parallel cache misses. This is called caches can serve hits under multiple cache miss in progress. So, this types of cache is known as non-blocking cache because once a miss is in progress we are no longer blocking the cache, no longer blocking the processor. So, for an out of order processor, non-blocking cache is really needed in order to deliver the performance.

(Refer Slide Time: 14:04)



So, there are two types, first is called hit under miss, and then the second one is called hit under multiple miss. So, you can just imagine the CPU is working. In that case, there is a miss that occurred, but CPU still running. There is one more miss that is occurring. So, this is the miss penalty of the first miss. The second miss is having another penalty. So, you will reach a point where you cannot further run.

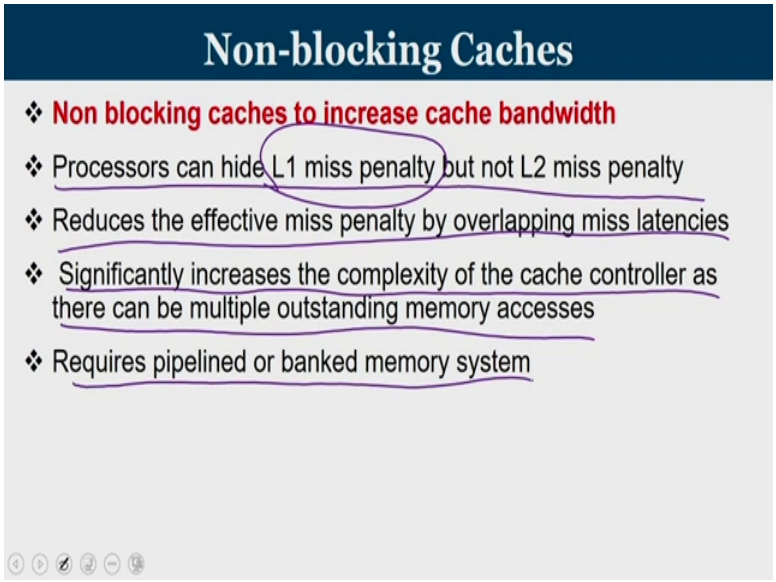
That is the time at which CPU stalls. In-between whenever there are hits those are also being serviced. So, you have multiple outstanding misses in this case and this kind of caches are known as non-blocking caches. It is much needed for out of order superscalar processors for IPC increase, and L2 must support it with L1-MSHR, it is Miss Status Holding Register.

So, the scenario is when you have a cache memory from which multiple misses are processed together, it is called outstanding miss, and this miss data can come back to cache at different timings. We will learn when we learn about multicore processor why they are coming back a different timings. So, this cache memory needs to have a logbook where you need to retell this miss happened at this particular time to this address.

So, whenever something is returning back to the cache controller, they should know whether it is the first miss A, second miss B, or third miss C, which is the data that is coming up. Such a logbook is called the L1 MSHR, it is called miss status holding register. So, miss status holding register is nothing but we have register in which for every miss the entry is being recorded and whenever the miss is going to come back the entry is being removed.

So, if the MHSR size is going to be 8, that means my cache controller can support outstanding 8 miss at a time. So, once the MHSR is full, then the processor is stalled because there are already 8 outstanding misses, until one of them returns processor cannot work forward. So, MSHR is a very important control unit or rather storage which help the processor for out of order execution.

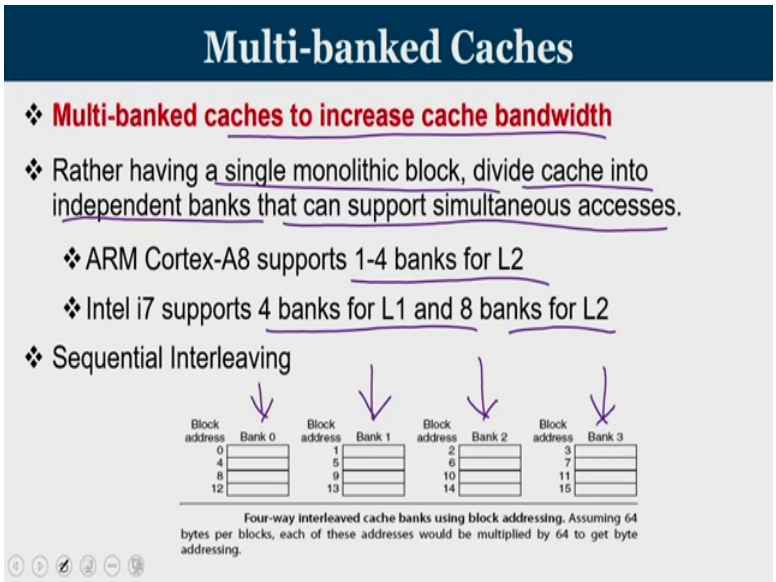
(Refer Slide Time: 16:21)



Now, non-blocking catches, the advantage is processors can hide L1 miss penalty, but sometimes it may not be possible for us to hide L2 miss penalty, but they are very high. Why processor can hide L1 miss penalty because processor is still running, there can be multiple L1 misses parallelly processed, so each miss one feel that much impulse.

It reduces the effective miss penalty by overlapping the miss latency. It significantly increases the complexity of cache controller, there can be multiple outstanding memory access and these memory accesses will return back out of order. It requires pipelined or banked memory system to support.

(Refer Slide Time: 16:59)



Now, we move on to the next optimization that is called multi-banked cache. Multi-banked cache will increase the cache bandwidth, it means the number of blocks that they can supply

from the cache to the processor. Rather than considering your cache as a single monolithic block we can divide the cache into independent banks that can support simultaneous access. So, ARM Cortex-A8 supports 1 to 4 banks for L2.

Intel i7 supports 4 banks for L1 and 8 banks for L2. And this is typically done with the help of sequential interleaving. So, look at the diagram that is given in the slide. Imagine that you have 16 sets in a cache. Set 0, 4, 8, and 12 they belong to bank 0 and 1, 5, 9, and 13 they belong to bank 1. Similarly, the balance of the set also has been divided.

So, rather than taking all your 16 sets and keeping it in a single place, I divide the entire sets into four, set 0, set 1, set 2, and set 3, and again set 4, set 5, set 6, and set 7. So, since you have separate controls circuitry in all these cases, whenever there is an access to set 0, when it is in progress, if you know that, because of a spatial locality there is very high probability that the next access is to set 1.

Then, the control circuitry in set 1 can do some pre-processing such that the moment access in set 0 is complete set 1 can take over. So, the initial processing that is to be done on set 1 can be done in background while we are processing set 0. It is as good as, like, imagine you are going to read a textbook, I am going to read this, even before the reading is over my hands are already ready such that I can turn the pages.

So, it is not like my hands are moving at the end to the textbook such that I can tilt the page. So, background you are doing some work which will help you start reading from the next page early. So, as and when you are completed your hands are ready and then you move on. So, this is called, basically, the overlapping work that you do or you can look in different way. Imagine that you are having a textbook.

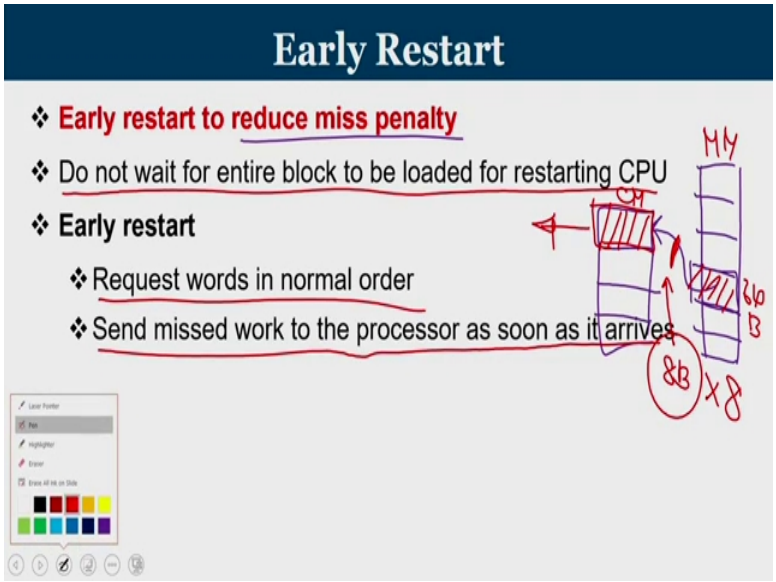
Now, you are reading sequentially, page 0, page 1, page 2, page 3 like that. Assume that all the even numbered pages are in one book. So, you are going to divide the textbook into two, section A which consists of only even numbered pages, and section B which consists of only odd numbered of pages.

So, if you wanted to read you read book A, then B, then again come back to A, again go back to B, like that. So, while you are reading the first chunk, let us say, page number 0, page

number 1 is already kept ready. So, as on when 0 is over you move to page 1, you are reading page 1.

When you are reading page 1, I can keep page 2 ready. So, as and when page 1 is over you move to page 2. So, rather than considering it as a single unit we divide the entire book into two sections and then parallelly doing a lot of operations in background such that we have a smooth transition when you move from one block to another. This whole idea is called multibanked caches.

(Refer Slide Time: 20:30)

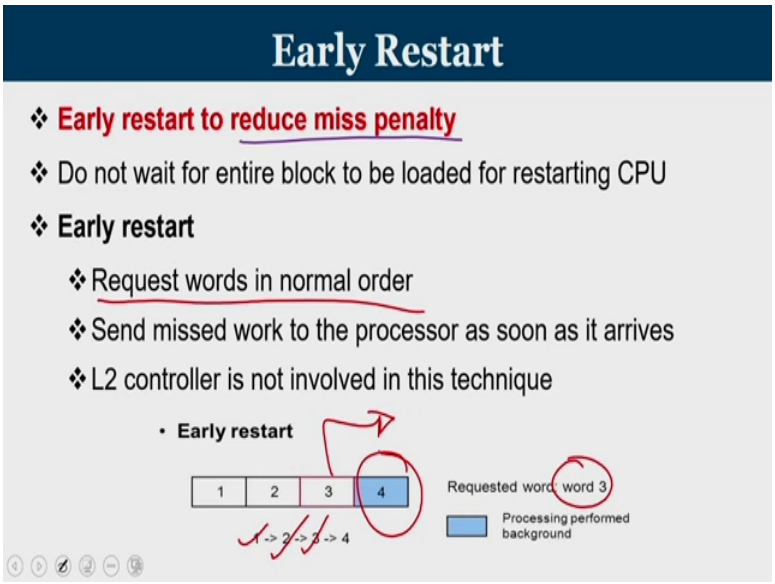


Now, we move on to the next optimization technique which is called early restart. Early restart is used to reduce the miss penalty. We do not wait for the entire block to be loaded for restarting the CPU. So, generally, if you look, this is my main memory structure, let us imagine these are the blocks in the main memory and let us say this is my cache. Now, when there is a miss in the cache, then the main memory contents that is one particular block from main memory is loaded into the cache.

Now, the contents of this main memory location which is there cannot transfer all these contents in one stretch. It is limited to the bandwidth of this particular bus that connects from main memory into the cache memory. Imagine your block is 64 bytes and your bus is only 8 bytes. So, the 64 bytes which is a block of data from main memory it requires eight consecutive time segments to transfer them because in one stretch I can transfer only 8 bytes.

So, we require eight such stretch to completely fill up the cache block. Only when a cache block is completely filled, then only we can resume. That is the conventional way. So, what we do in early restart is we do not wait for the entire block to be loaded for restarting the CPU. So, that is called early restart. What you do is you request words in the normal order and send the missed word to the processor as soon as it arrives.

(Refer Slide Time: 22:22)



So, L2 controller is not involved in this technique. Let us try to understand what is early restart. Processor is giving an address and your cache will check on it. You will find that it is a miss. Once you know it is a miss, then you go to L2 cache and from the L2 cache you bring word by word. As and when the word that processor requested arrives your L1 cache controller will forward it to processor so that processor should not wait for it, processor can start the work.

In the background the subsequent words are being copied. This technique is called early restart. In this case, your L2 cache controller is not involved because L2 cache controller will read it as a normal miss, it supplies all the words in sequence, but there is a slight modification that is required in the L1 cache controller.

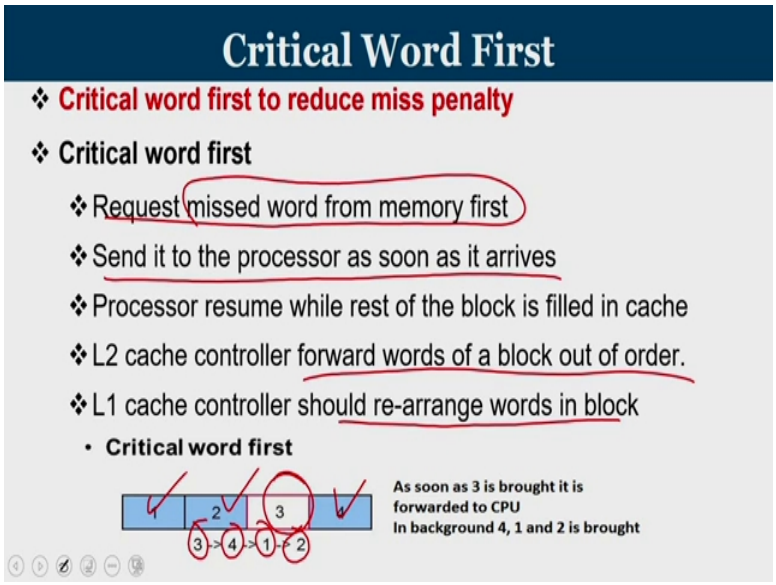
What L1 cache controller do is you need to have a controlled circuitry which will understand what is the critical word that you are talking about or the word that processor wants as and when the word arrives. So, whenever each word arrives you need to have a mechanism is this the word that has been requested by the processor. If it is not, just try to store it in the L1 cache and subsequent word is coming.

When you come to know that the requested word has reached, along with storing in the cache you are going to forward it to processor as well. So, in the conventional way, processor will wait until the entire words of a given block is loaded in the corresponding cache block, whereas in this case, as and when the word arrives. So, if the word happened to be the first word in a given cache block, early restart will give you very good performance.

If the word happened to be the last word in a given block, then we cannot do much about it. It is same as early restart under conventional cache. When you look on benchmark for applications we can find out how much percentage of your word access will actually result in this kind of performance.

So, think of a case that your requested word is 3. So the requested word is 3. So, you are bringing 1, then 2, then 3 is coming. So, 3 is applied to the processor and 4 is reaching the cache in the background. So, processor is already reading the three or working with word 3 in the background, word 4 is being kept.

(Refer Slide Time: 24:42)

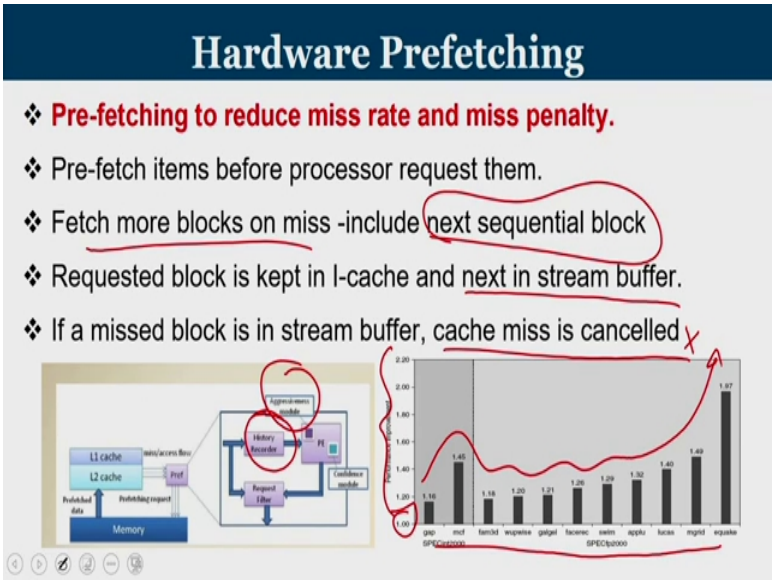


Yet another optimization is called critical word first. It is an extension of early restart. This also reduces miss penalty. In critical word first, request the missed word from the memory first. That is very important. In the case of an early restart, your L2 cache controller still supply the data in order when it comes to criticalr word first, whichever is requested that is going to reach first.

So, send it to processor as soon as it arrives. Processor resumes while rest of the block is filled in the cache. An L2 cache controller forward words of a block out of order. So, L2 cast controller also is involved in this. L1 cache controller should rearrange these words in the corresponding block.

The same example, you know that this is the critical word. So, first to send 3, followed by 4, then go to 1 and 2. So, bringing a 4, 1, and 2, that happens in the background. So, critical word first will add on performance because of the fact that the word that is requested by the processor is being brought first.

(Refer Slide Time: 25:46)



We next move on to hardware prefetching. Hardware prefetching is going to reduce miss rate as well as miss penalty. Let us try to understand what is prefetching. It is a very active area of research in cache memory community. We know that because of spatial locality there is a high probability that the adjacent addresses of a currently requested address will be in demand.

So, if I am working on L, L + delta L will be requested in the near future. So, when I encounter with a miss, the requested block is being brought to cache. At the same time, another block also which can be the adjacent block that is also brought in parallel. So the requested block is being supplied, processor is resuming.

In the meantime, rather than processor issuing one more new cache miss, we are predicting what is going to be the next block that will be demanded by the processor, and that block is

being brought parallelly such that as on when it misses, may be it is already present there, or it is in the near vicinity or it is actually traveling towards your L1 cache. This technique is called prefetching, bringing something before it is demanded.

If the predicted block is present in the L1 cache, it reduces miss rate. If the predicted block is kept in some buffer near L1 cache, then your miss penalty is reduced. To prefetch items before processor requests them, fetch more blocks on a miss, that is what has been needed, and sometimes you bring the next sequential block and sometimes you predict, it may be an offset value next nth block **(27:24)**.

So, depending upon the access pattern we can predict what is going to be the next block, and the requested block is kept in I-cache and the next block, that is the block that has been predicted, it is kept in a small buffer known as stream buffer. If a missed block is in stream buffer, then cache miss is actually canceled.

It is already there and there is no need for further cache miss. So, you need to have some kind of a history recording mechanism and then somebody has to predict by looking into it, and then you are going to bring what has been needed. So, this shows that for certain benchmarks, these are all different SPEC benchmarks, what is the performance improvement if you are going to enable hardware prefetching.

You can see that the conventional one the performance is one. When you enable prefetching, you are able to get speed up of up to 2. So, this shows that prefetching can really improve average memory access time. So, with this we come to the end of the cache optimization techniques. In the last class, we learned some elementary optimization techniques and today's class we learned slightly advanced concepts. Thank you.