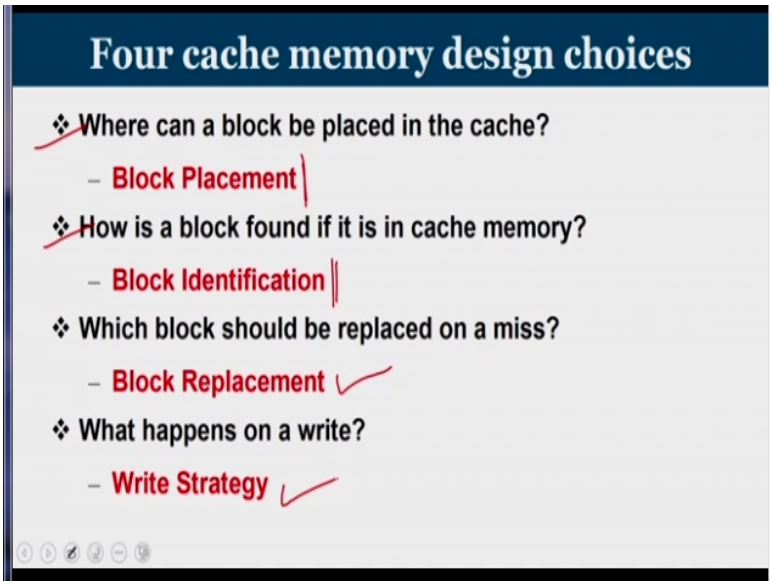
Advanced Computer Architecture
Prof. Dr. John Jose
Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology-Guwahati

Lecture-19
Block Replacement Techniques and Write Strategy

Hello everybody, today in lecture number 14 our discussion agenda is on block replacement techniques and write strategy on cache memories. In the last lecture the concept of cache memory was introduced, we have seen some of the design aspects of cache memory and we have seen in depth how block from main memory are placed in the caches. Similarly when CPU gives an address the technique to find out whether it is present in the cache or not was also explored.

We have seen the 3 different types of cache memory mapping techniques or block placement techniques like direct mapped cache fully associative caches and set associative caches. Now let us move into the next design aspect that is about block replacement techniques.

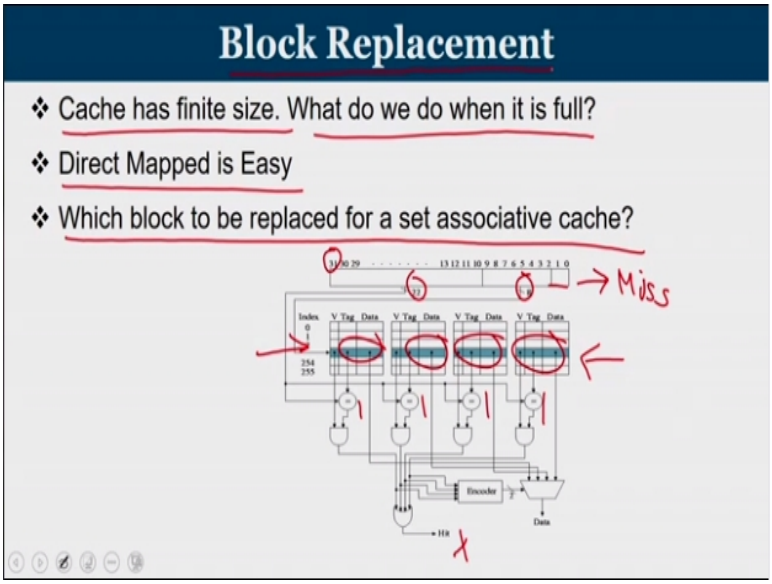
(Refer Slide Time: 01:26)



In the last lecture we have seen that there are 4 cache memory design choices out of it is the first 2 we have already seen where can a block be placed in the cache which should be define by the block placement techniques and how a block is found if it is present in the cache it will be dealt by the block identification technique. Now we are moving into the next 2 which block should be

replaced when there is a cache miss or not that is addressed by block replacement techniques. And how are you going to deal with whenever there is a write operation that is to be done on the cache defined by the write strategy.

(Refer Slide Time: 02:04)



Coming to block replacement, generally our cache has finite size and a program that we are going to execute will be generally larger than the size of the cache. So during the execution of the program we may have to bring in new elements into cache and take out already and take out elements there are already there in the cache. So what do we do when a cache is full, now think of a case if the cache is direct mapped hope you know what we mean by direct mapped technique.

For every block in main memory they are exist a predefined location in the cache, so when a new address is coming upon searching in the cache if we find that, that is a miss. Then the block when it is been brought from the main memory as a predefined location. So whichever block that is already existing in the predefined location has to be eventually taken out. So there is no choice as far as a direct mapping is concern wherever is the mapping location to that location we how to bring and whoever is there he has to be replaced.

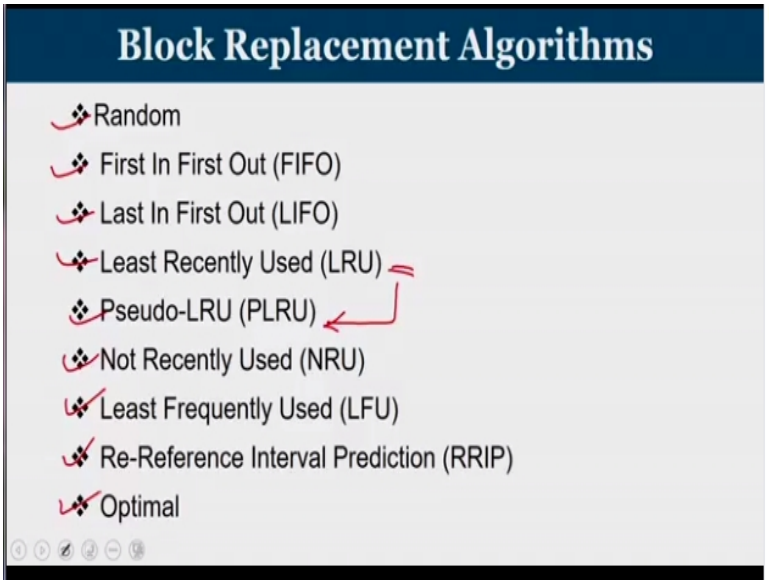
That is why block replacement technique is very simple and straight forward and there is no choice as far as the design is concerned regarding which block is to be replaced in the case of the

direct mapped cache. Now which block is to be replaced when the cache is going to be set associative, so consider the case that you have CPU which is going to give you a 32 bit address. This 32 bit address is divided into we can see that the 22 bit tag and 8 bit index and the remaining is going to be offset.

Now using this 8 bit index you are going to index into one of the 256 sets, let us imagine whatever is a shaded color that is 1 particular set and you are going to perform tag comparison on each of them. This is a 4 way associative cache but unfortunately we have found that none of the tag matching comparison is going to give a hit. So there is no hit that happen that means the given address as encountered in a miss.

Now if it is a miss we go all the way to main memory we are going to bring a block of data and where is this block of where data going reside this block of data has to reside in the same set because the set number is prefixed from the address. Now in the set all the 4 ways are full meaning the valid bit is already 1 and it is occupied by some other data. So out of these 4 blocks that is residing in the set which is to be replaced and that is all about cache block replacement technique.

(Refer Slide Time: 04:48)



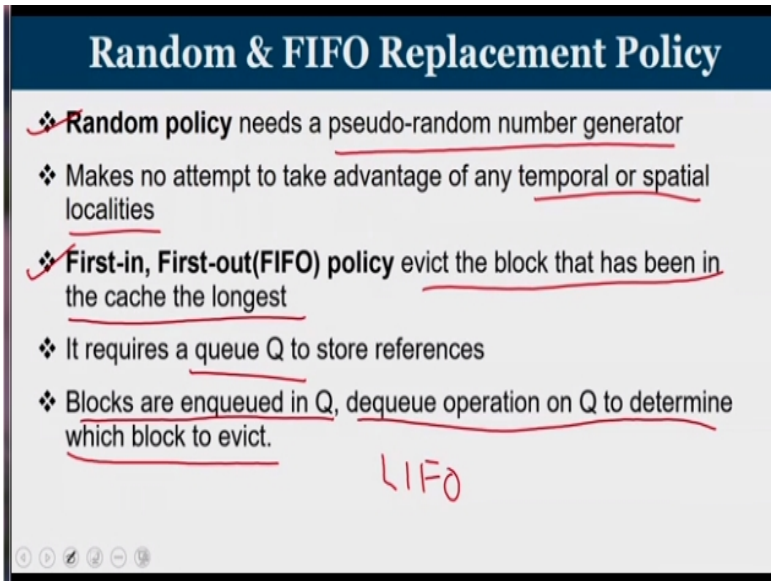
There are different block replacement algorithms first one is random then we have first in first out, last in first out, least recently used which is a popular one. And then Pseudo LRU that is the

one that is practically implemented it is derived from LRU but for practical purpose we are using Pseudo LRU. Then we have NRU not recently used then least frequently used and then some of the modern techniques which use the re-reference interval prediction and then we have the optimal algorithm.

Let us try to understand what is the context in which we are going to work, just to summarize the block replacement algorithm. Block replacement algorithm happens only in the context of set associative cache whenever there is a miss occurring on a given set index out of the all available blocks that are there in that particular set one has to be picked. So the block replacement algorithm is purely restricted to what is the block or which is the block that we are going to pick for replacement.

So the rest of the discussion is all on in a given set which of the block need to be the victim for replacement.

(Refer Slide Time: 06:15)



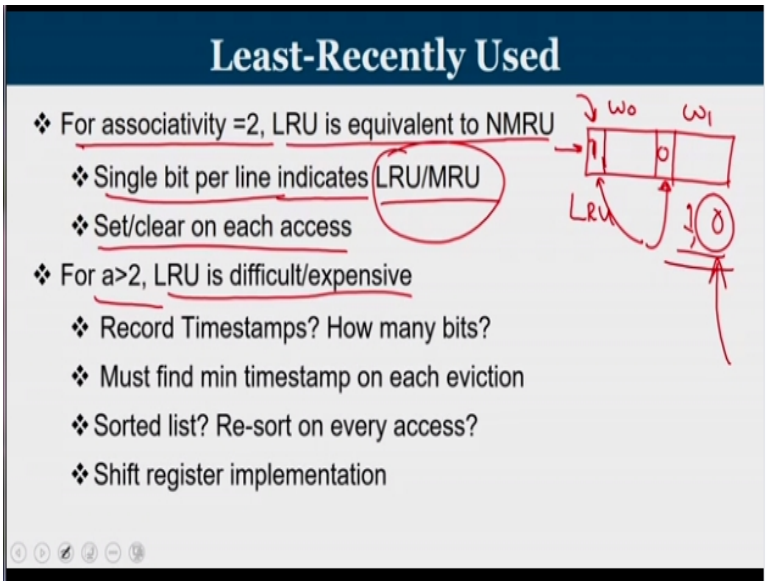
The first policy is called random policy which will generate a pseudo random number. So here we are not going to explore whether any temporal or spatial locality in a block or not out of the available block in a given set pick somebody by random. So whether one of the block is heavily use the re-referenced many times one is rarely used these aspects are not looked into. But

implementation wise random is very simple then we have the first in first out policy which evict the block that has been there in the cache for the longest time.

So to implement that you know that we require a Q, so every time whenever a new block is coming to the cache in a given set, the Q is been maintained. And blocks are enqueued in the Q, dequeue operation on the Q to determine which block is to evicted. So whenever there is a block replacement that is been needed look into the Q and find out which of the block has reach the cache first and that is been taken out.

Here also we know that temporal and spatial locality aspects or re-reference of any particular block, usage of a particular block is not looked into, we are only looking on that aspect of who has reached first. Similarly we can find out what is there in LIFO policy as well last in first out the block that has reach the cache last is going to be the victim.

(Refer Slide Time: 07:41)



We now move to least the recently used algorithm when we have an associativity = 2 least recently used is equivalent to not most recently used. So think of a case that let us say I am talking about a 2 way associative cache this is your way 0 and this is the way 1 we are adding 1 extra bit there is a single bit per line which is called a LRU bit. So whenever I am going to access a block let us say I am going to access a block I put this as 1, this is eventually 0.

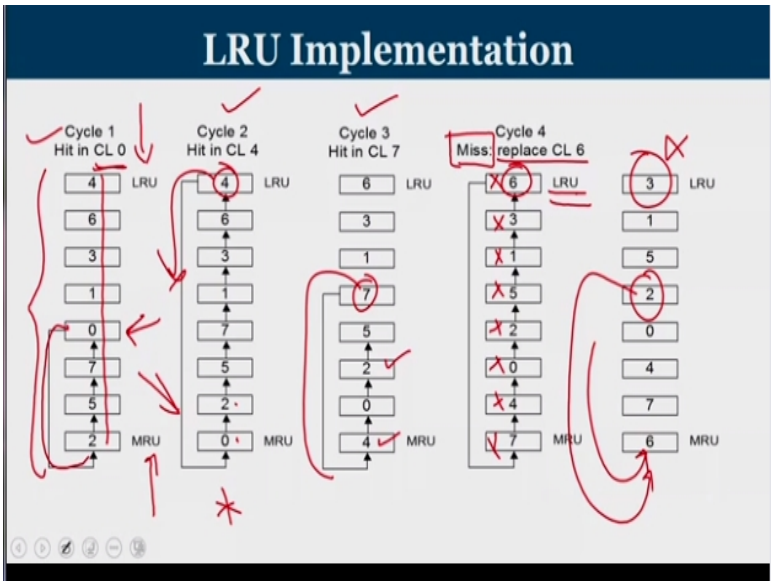
Now think of a case that every access that you make in to this particular set access will be either here or it will be here whichever block you are going to access you make that as 1 and other one has to be made to 0. If you implement this policy then at a time of replacement you look into both the blocks whichever is having 0 that was not recently used. The other one which is having 1 in this LRU bit that is the least recently used one.

So this is very pretty simple and straight forward we are using extra bit and this bit is nothing but LRU or MRU most recently used one. So the LRU bit how it works when you have 2 blocks whenever I am going to access a block I am going to make the LRU bit as 1. That means it is recently used and other one is put as 0 that is not recently used. So for every access we are going to set this bit or clear the corresponding bit whichever is acting 0 that is a least recently used and that has to be evicted out.

Now the question is when associativity is slightly bigger than 2 let us say 4 way associative or 8 way associative the single bits scheme will not help. So LRU is little difficult and expensive, then we have to record the time stamp what was the time at which is (()) (09:51) is record was referenced. And then you must sort out it is timestamp at the time of eviction there are different timestamps for each of this ways and you have to sort them and find out which is having the minimum one.

So sorting and re-sorting that is having it is own overhead and it can be done when we shift register implementation as well.

(Refer Slide Time: 10:11)



So let us have an illustrative example by which we will know how LRU is been implemented. So consider an 8 way associative cache where in the number written inside this squares is the way number, consider an 8 way associative cache where the number is the way number. And what you see on the bottom side is the most recently used way and on the upper side is the least recently used way, this is the way by which an LRU is been implemented.

Now this is for 1 set, similar such kind of a link-list is there for every set in the cache memory. To what this particular diagram shows on the left hand side is way number 2 was most recently used before that it was way number 5 then 7, then 0, then 1, 3, 6 and 4. So out of all the 8 ways 4 is the 1 that was least recently used. Now consider a case that you got a hit in cache line 0, so 0 is the 1 where in we got a hit.

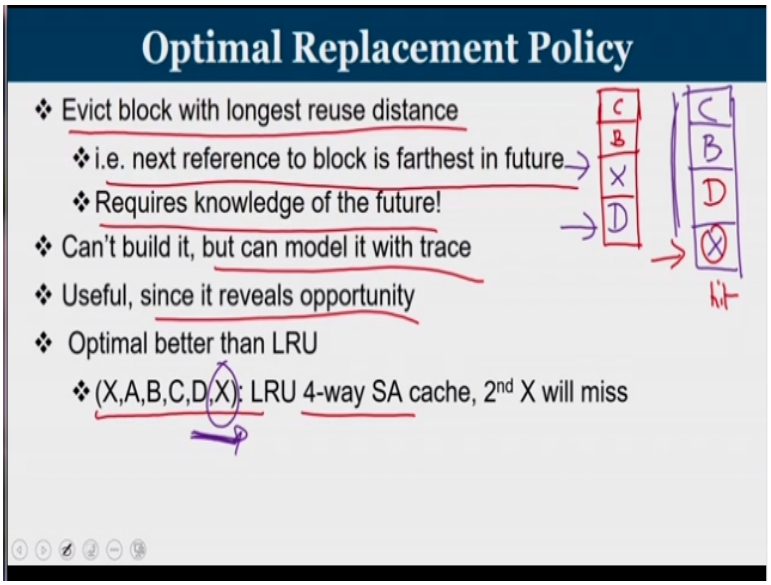
Once we get a hit 0 will become the most recently used, so 0 is promoted from there all the way to 2 that is what you can see here. So this one is the updated list after you are going to get a hit in 0, so 0 will become most recently used followed by 2 and then the chain is maintained. Now in cycle number 2 you got a hit in cache line 4 you know that cache line 4 was the least recently used, now it got a hit.

So 4 will go to the front of the list that is what we can see 4 and the 0 which was previously there in the front of the list and then 2 like that. Now I got a hit in cache line number 7, so 7 will get

promoted, so 7 will come followed by 4, 0, 2, 5, 1, 3, 6. Now I am going to have a miss when I have a miss it is none of it is not present in any of the ways, so we have to replace which is a block that we are going to replace.

We are going to replace the LRU cache block and which is a LRU cache block, way number 6 that is why it is called replacing of cache line 6 and then 6 comes the most recently used position. So whenever there is a miss you are going to replace the upper one and whenever you get a hit any of them let us say I got a hit in 2 the 2 get promoted to the front of the link list. So by maintaining a link list for each of the set we can have the LRU implementation without using any counter. Because it is only ordering is the relative ordering is important not absolute value of the timing.

(Refer Slide Time: 13:02)



Now what is the optimal replacement policy, we have to evict block with longest the reuse distance. That is we need to find out which is the block that is farthest in the future. The next reference to the block that is farthest in the future has to be replaced and it requires knowledge of the future. So when processor is going to give addresses at that point it does not know what are the future addresses that is going to be accessed.

Since we do not know what is a pattern in the future optimal replacement policy is very difficult to implement. So ideally it looks very fine the good theoretical concept if we know what are the

future memory address then at the time of replacement I have to replace that block which is going to be access farthest in the future. So generally optimal replacement policy is the best one that you can achieve and all others are the practical ones.

So ideally a purpose of a block replacement algorithm is to achieve as close a performance which is near to your optimal replacement policy. We can build optimal replacement policy but can model it with if you have a trace available and it is very useful since it gives you lot of opportunity to know what are the future trends and optimal is always better than LRU. So consider the case that you have a sequence X, A, B, C, D, X and let us say these are the block numbers which are all mapped to the same set on a 4 way set associative cache.

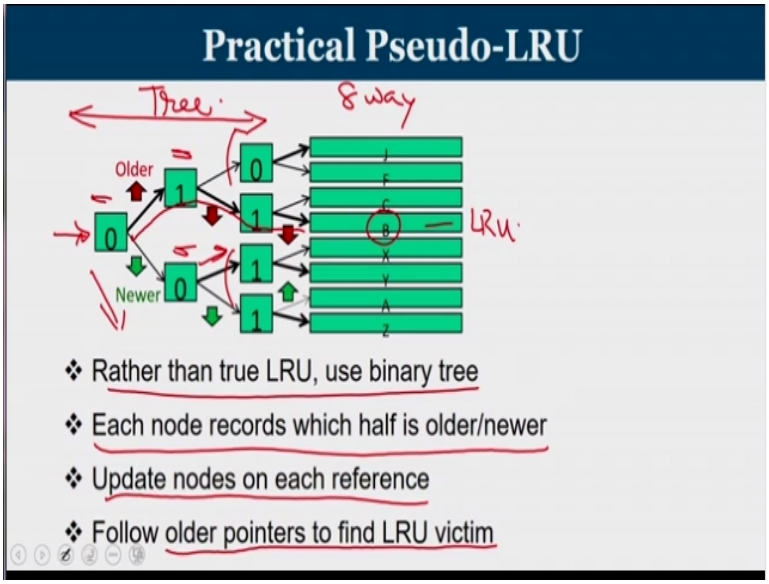
So if you look at the 4 way set associative cache first let us say these are the 4 ways we have X and then we have A, then we have B, we have C. These are the 4 ones in a 4 way associative cache, so X is residing A is residing, B is residing and C is residing. Now in this case all the 4 ways are full and if you are going for LRU policy out of we can know that X is the least recently used one.

So when D comes I am going to replace X and put my new D there, so D is going to be in the position of X. But we know that the very next sequence is X, so whatever we how just thrown out that is going to be brought again. So in that case A is least recently used, so the new X will be populated here, essentially D and X the resulted in a miss. All others will result in miss but D and X will result in a conflict miss.

Now the same thing if we are going to work with your LRU, your optimal algorithm then we know that X is kept A, B and C this is the initial 4 configuration. Now when there is a miss that is going to happen to the D we look into out of the 4 that is already existing in the cache block or in the cache set which is the one that is going to be used in the near future. So we know that X is going to be used in near future provided if you know the future trace then X should not be taken out.

So we are going to take out A in this case because we know that X is going to be accessed in the near future. So replacing X will not be a good idea, so what I do is I put D here. So D will encounter a miss but when it comes to X it is going to be a hit again. In this example we can see that optimal algorithm is better than least recently used algorithm.

(Refer Slide Time: 17:00)



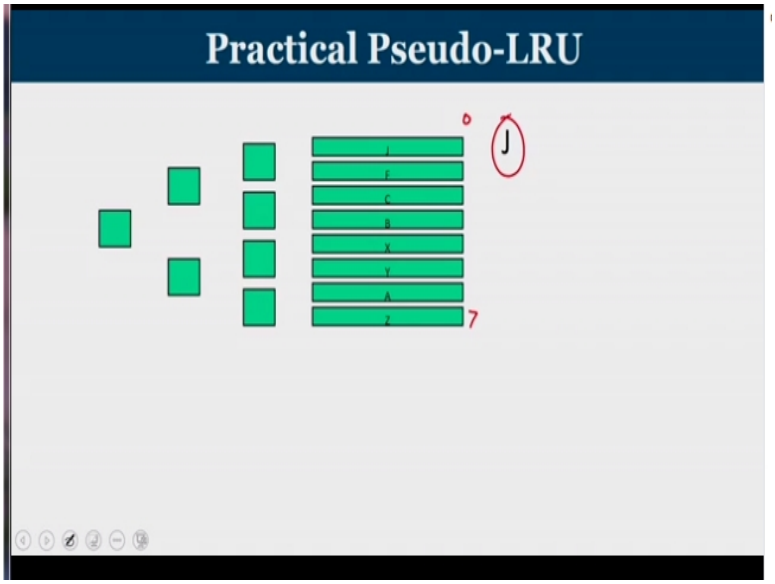
Practical implementation of LRU, it involves maintaining a link list and manipulating the list and the links of the link list has to be adjusted based upon hits and misses. LRU is a very good algorithm but it is all overhead is slightly on the higher end because of manipulation of link list on every set. Researchers are proposed a new one which is called a Pseudo LRU wherein it is not a pure LRU algorithm least recently used algorithm, it is a practical approach which is relatively very close to the LRU implementation.

Let me draw your attention to an illustrative example in understanding how pseudo LRU works. So we are maintaining a tree that is what we have see here there is a tree for every set, now rather than true LRU we are using a binary tree each node records. So in the tree let us say in this case it is an 8 way associative cache, so this tree has 7 nodes that is a root, 2 children. And then each of the children has 2 more children, each of this node records either a 0 or a 1.

So each node records which half is older, so when I put 0 that means that upper half is older and the lower half is newer. Similarly here we have 0 means upper half is older and lower half is

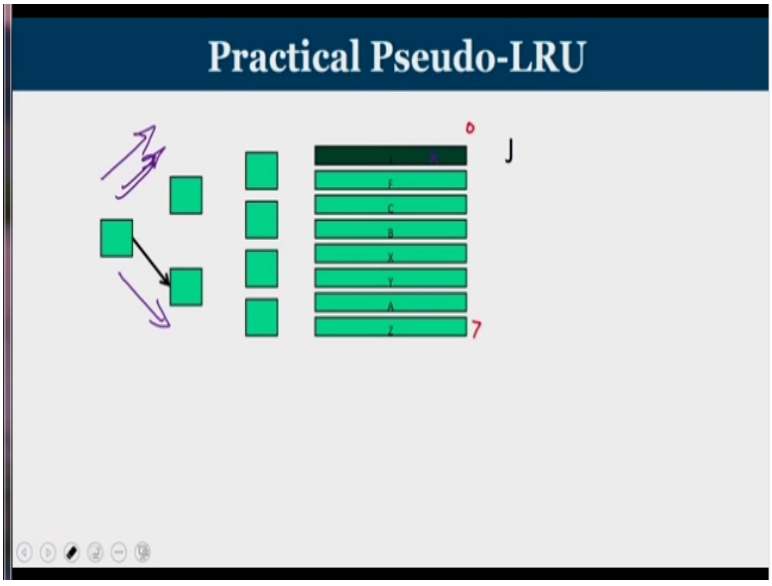
newer. So we can see that this is a bold arrow, this is a path bold arrow, so if you look into this particular line of this set is the older one. So you update the nodes on each of the reference and you follow the older pointers to find the LRU victim, so this is basically your LRU in this context.

(Refer Slide Time: 18:59)



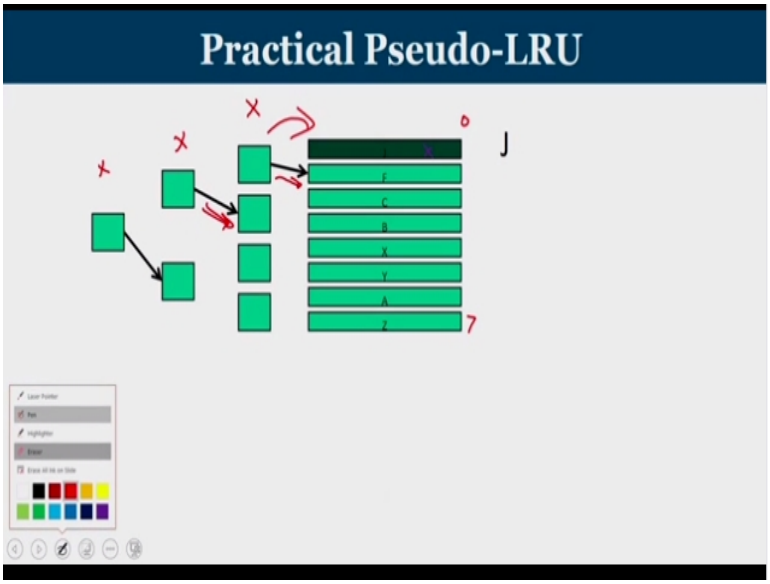
Now let me do an illustrative example, so consider the case that these are the ways let us say way 0 all the way up to way 7, so the 8 cache lines in a given set are been shown here. Now let this J, F, C, B, X, Y, A and Z are the contents of these cache blocks of the given set. Now we are going to get an access wherein CPU once J.

(Refer Slide Time: 19:35)



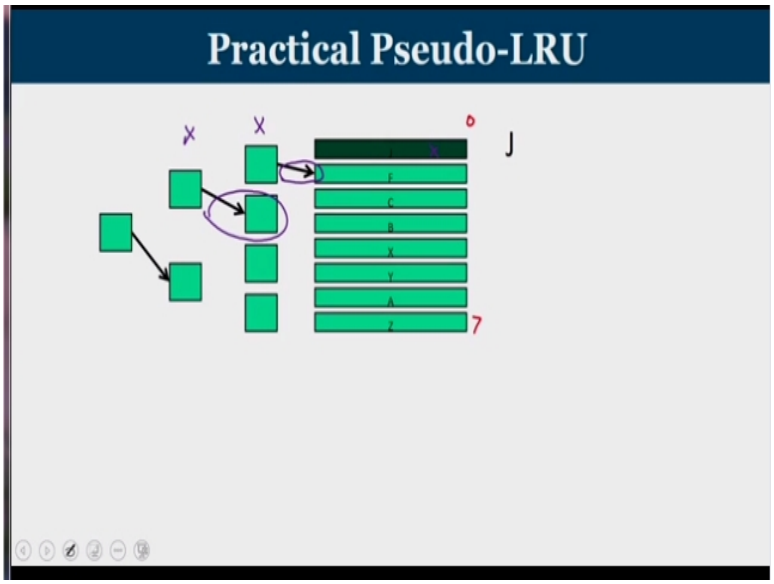
We know that J is a hit but where is J located, J is located here, so to reach J I am going to access J from the root to access J, I have to go to upper side, that means the lower side is going to be my LRU path. So J is what I have to access, so as far as the root is concerned after accessing J the lower half is the least recently used because I am going to use the upper half, so lower half is the least recently used path.

(Refer Slide Time: 20:20)



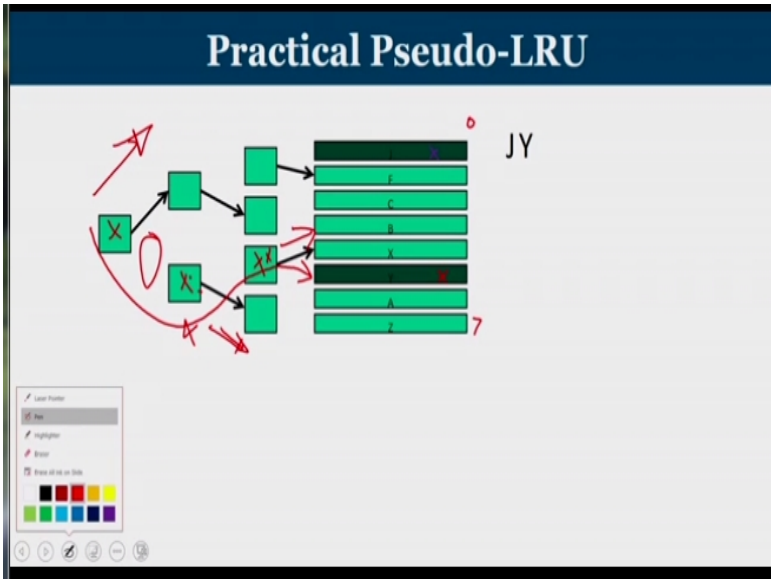
That is why a arrow, so here the bold arrow always point to the LRU path least recently used site. Then upon reaching J from here I have to reach this node but to access J I have to still go further. So this side each of the node has to mark here also the lower side is the LRU path and when you go to the next level to access J I have to go here. So naturally the other side is the LRU path.

(Refer Slide Time: 20:54)



So after accessing J this is the bold arrow whatever you see will tell you what is the LRU path after accessing J. So as for as root is concerned the lower half of the root is LRU as for as a next child is concerned. Like in this case this child is if this child is concern then this the LRU path as far as this child is concern then this is the LRU path.

(Refer Slide Time: 21:21)

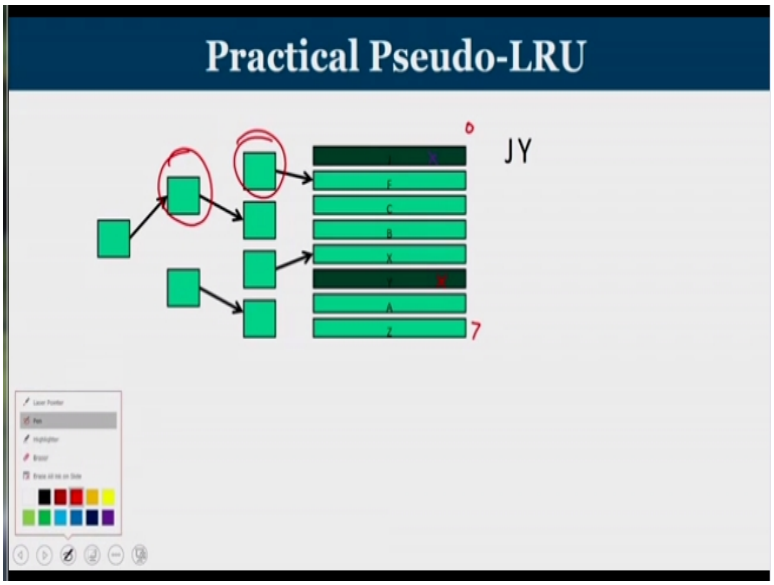


So I hope with the illustration it is clear what will happen after accessing J let us continue further now. The very next request is to Y, where is Y, Y is here, to reach Y from the root this is actually the path to reach Y. So these are the nodes that are getting affected. So the pointers of these 3 nodes in the tree will be affected upon reaching Y. So the root the moment I access Y then the upper side of the root will become least recently used.

So because I am going to travel in this 5, so the pointer in the root this one will get flipped, so that is Y that you can see and the root value will get flipped. And what about here, here if you look into to access Y how to go to the upper child to when I go to upper child then naturally this is the place where the bold arrow comes because that is a side which is going to be least recently used and similarly here upon reaching this particular node to go to Y it is again the lower child.

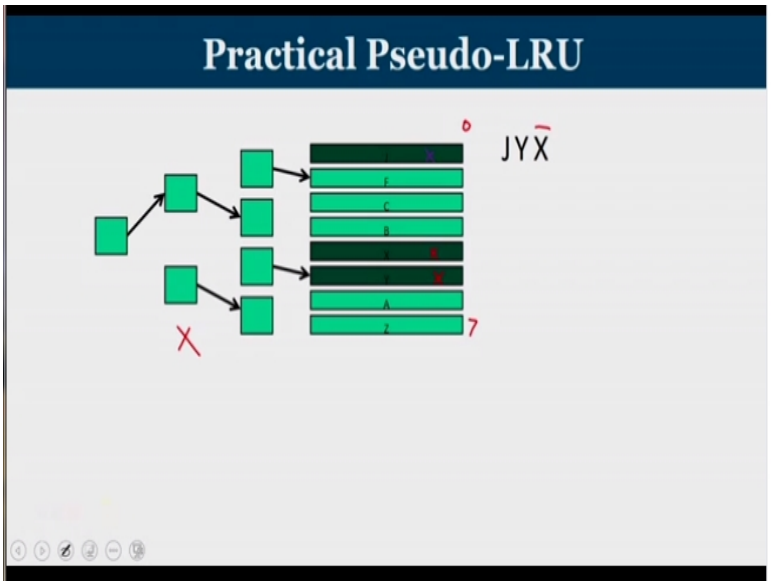
So naturally the upper child is going to be the least recently used, so this is the path that we have after accessing of the block which carries a data Y.

(Refer Slide Time: 22:44)



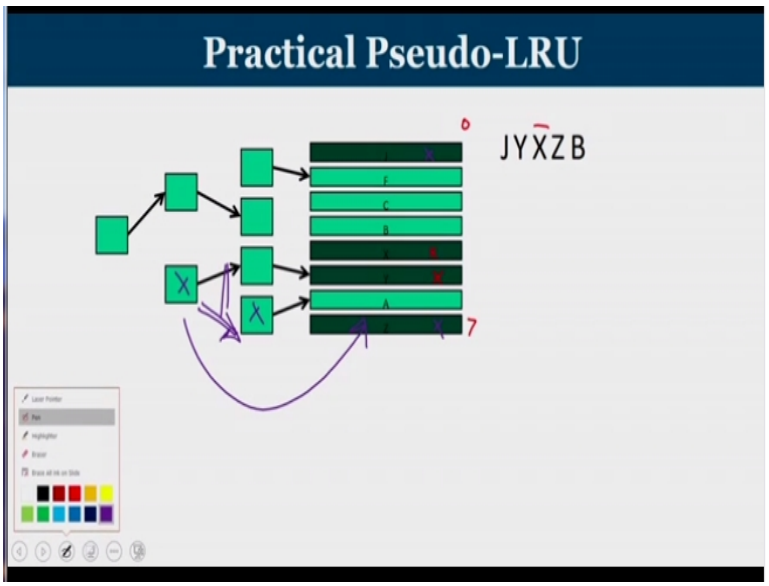
Remember during exercise this node is not affected, this node is also not affected whatever values that they had that will be retained as such.

(Refer Slide Time: 22:59)



Now let us continue further the next one is X, so where you have X you know that X is somewhere here. So an X accessing X you kindly carefully observe the animation X is what you want, so to reach X the arrow in the root is not going to get affected still the upper half will be the least recently used, no change, when it comes to the second also. Here also we do not have any change because X is on the upper half when you come to the last one it get flipped back to this. So this is the status after accessing X.

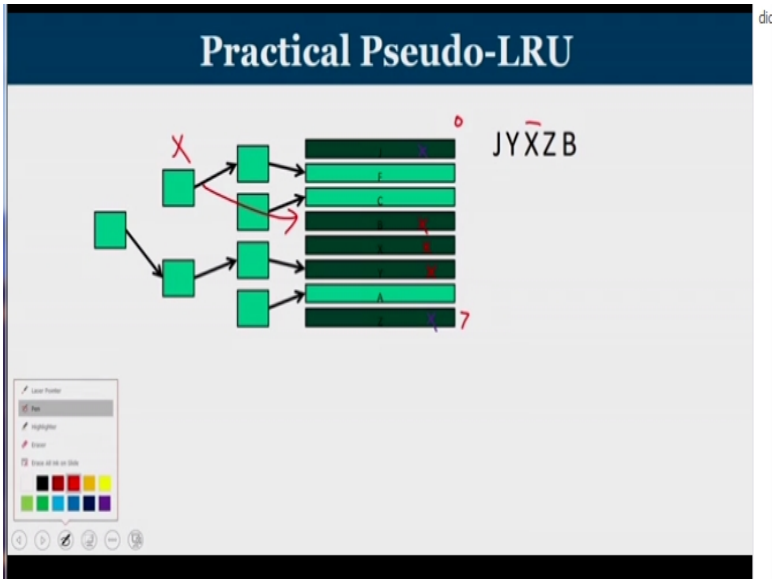
(Refer Slide Time: 23:40)



Continuing further next we have Z where is Z, Z is there here, to access Z, you know that is it is on the bottom half root there is no change. Because still the upper side is the LRU, this wherein till now it was showing that this side is the least recently used, if you are going to access Z then

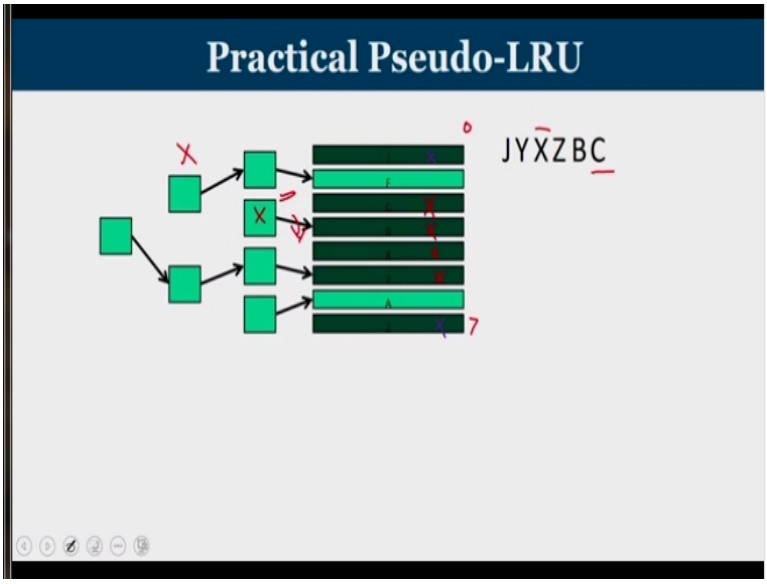
this will be flipped. And for this child Z is on the lower half, so LRU path will be there only opposite side that is on the upper half.

(Refer Slide Time: 24:20)



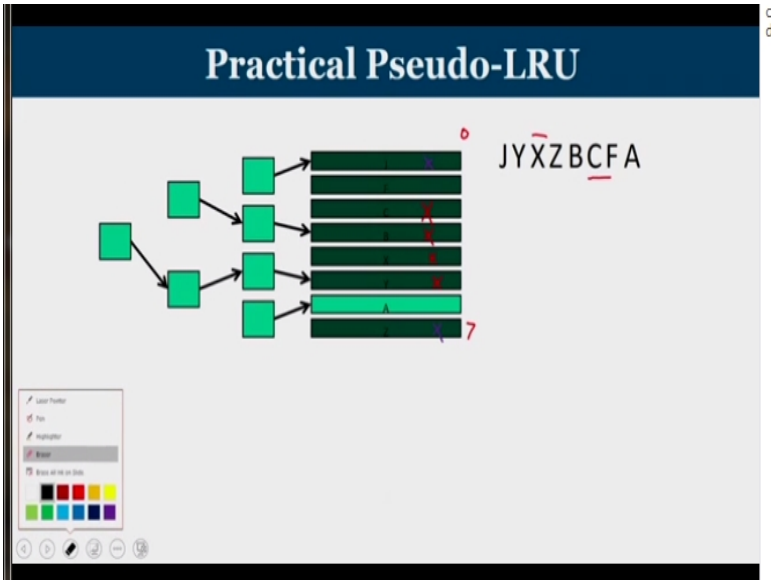
So this is what we get for accessing Z, now we have B, B is here, now B is on the upper half of the root. So naturally now flipping happens in the root level and then we are going to talk about this, it is also get flipped because B is in this path, so the arrows are getting flipped. Remember arrows always indicate the least recently used path, so when we are going to reach upon a block the arrow has to be flip in opposite direction, similarly here also the arrow get flipped like this.

(Refer Slide Time: 25:01)



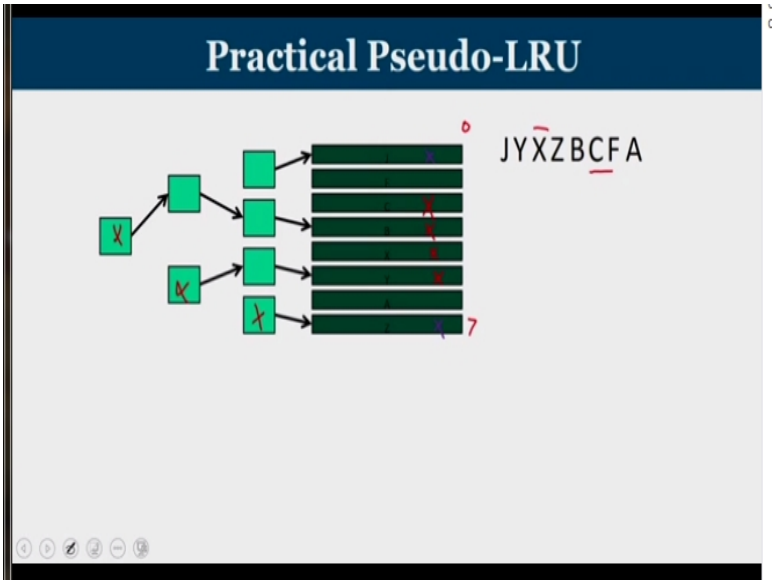
So this illustration will help you in better understanding of how things are been moved, continuing further next I have C, C is here so we can see that there is no change as for as root is concerned, no change. For this child no change and for this child since I am going to access in this region, the arrow will be flipped down.

(Refer Slide Time: 25:32)



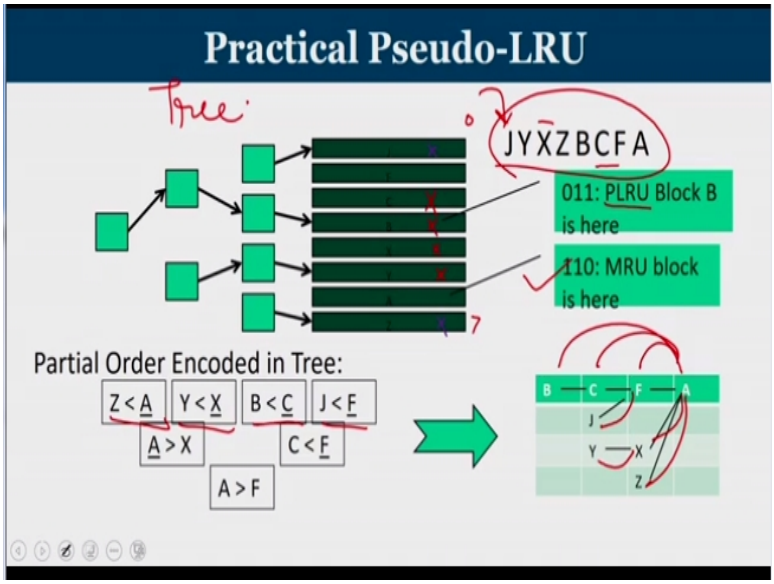
So at after accessing C these are the positions of the Pseudo LRU tree, continuing further next is F no change in the root. First child will get flipped, next child also will get flipped, so it goes up and then we have A which is there in the lower half since A is on the lower half.

(Refer Slide Time: 25:60)



So these are the 3 tree which is getting flipped we can see that for A root will get flipped, the first child will not be having any change, the last child will be flipped again. So this is what we have now after a trace is being given.

(Refer Slide Time: 26:19)

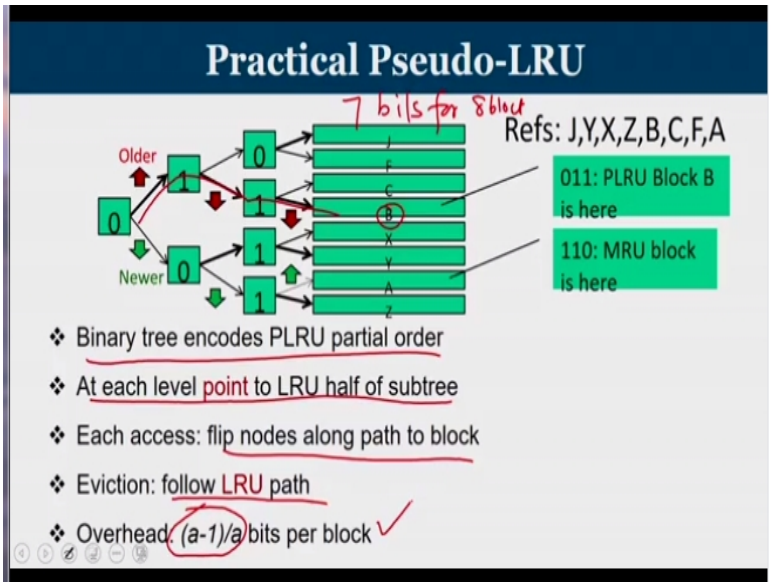


Now let us try to understand what we did here these are the traces that we got for each access into the cache the tree is been updated, the values of trees been updated. So it is basically a 7 bit value, the 7 bit value can completely manipulate this one. Now if we look into further this is the most recently used block and if you travel through the arrows alone that is the least recently used one or it is also known as Pseudo LRU.

So it is basically a partial order encoded in tree we can see that Z is so A is the recently used one. Similarly X is the other one, B when compare to C and J when compare to F. Now if you apply, if you look into that you can show that it is based upon how many number of block arrows that we have in reaching them. Continuing these continue together A is preferred over X and F is preferred our C, so both A and F are the winners and at the end A is the winner.

So A is the most recently used one that is a takeaway at the end of the whole story, meaning I can represent them using a tree, A is most recently used then F, C and B. But F is over C and J, A is preferred over X and Z meaning X is preferred over Y and like that this partial tree is going to give you benefits.

(Refer Slide Time: 28:11)

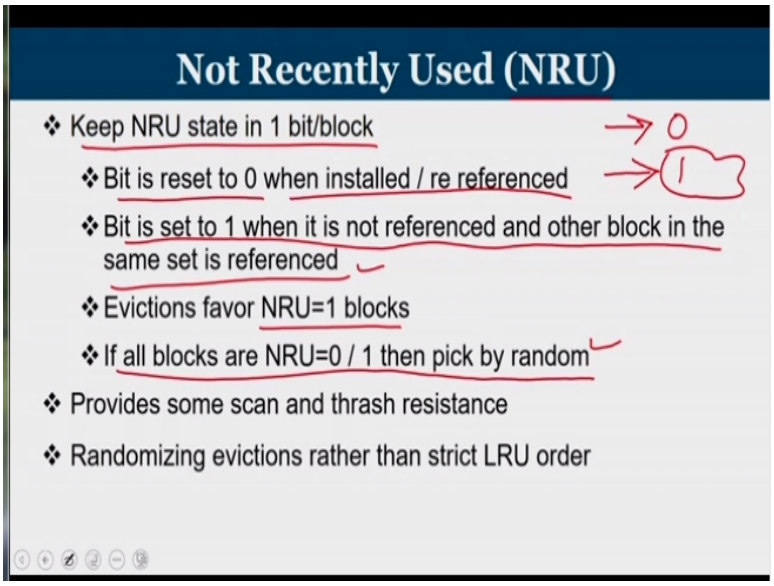


So what we have done here is a binary tree encodes a Pseudo LRU partial order, at each level the tree is pointing to LRU half of the subtree. In each access we have to flip the nodes along the path to the block and upon eviction you have to follow the LRU path mean just travel through the arrow, these are the bold arrows. So you will be able to reach the block that will become the victim and as for as overhead is concern.

Here we require 7 bits for handling 8 blocks, so the overhead is if it is an A way associative cache A – 1, that is 7 divided by 8 divided by bits per block that is a overhead.

7/8 bits per block

(Refer Slide Time: 28:59)



So we have learned what is Pseudo LRU, Pseudo LRU is a rather simpler approach as for as implementation overhead is concerned, it may not give you the perfect answer but it gives you a near approximate answer as for as LRU block identification is concerned. Moving further into yet another block replacement algorithms, not recently used sometimes rather than least recently used we are only bother about not recently used.

So we can represent whether somebody is not recently used or not using a single bit information, this bit is reset to 0 when you reference it or a block is brought all other are already once whoever is access that is made 0. The bit is set to 1 when it is not referenced and the other block in the same set is referenced. So when you reference or bring somebody put 0, all others value will be having 1.

Now when there is an eviction you look at all the values or all the blocks that are having 1 and pick 1 among them. Evictions will always favor

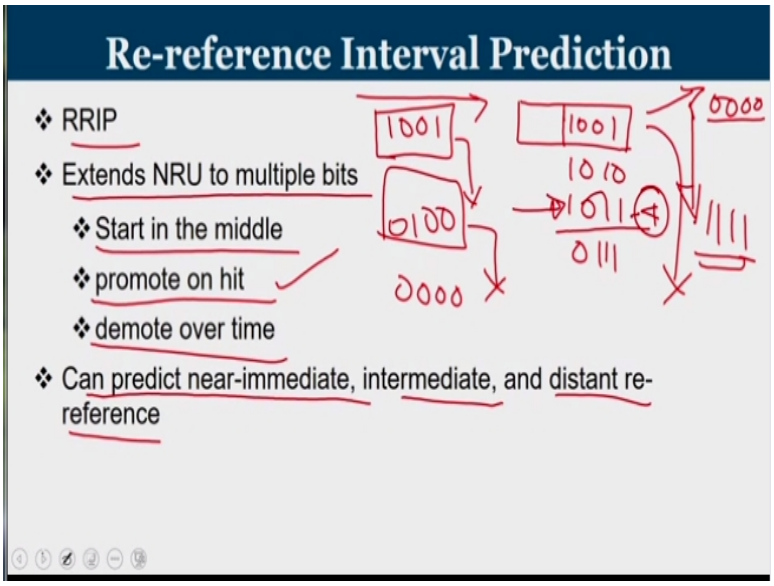
NRU = 1

blocks if all blocks are

NRU = 0

and 1 then pick somebody by random. It provides some scan and thrash resistance and it will help us to randomize eviction rather than strictly following LRU order.

(Refer Slide Time: 30:21)



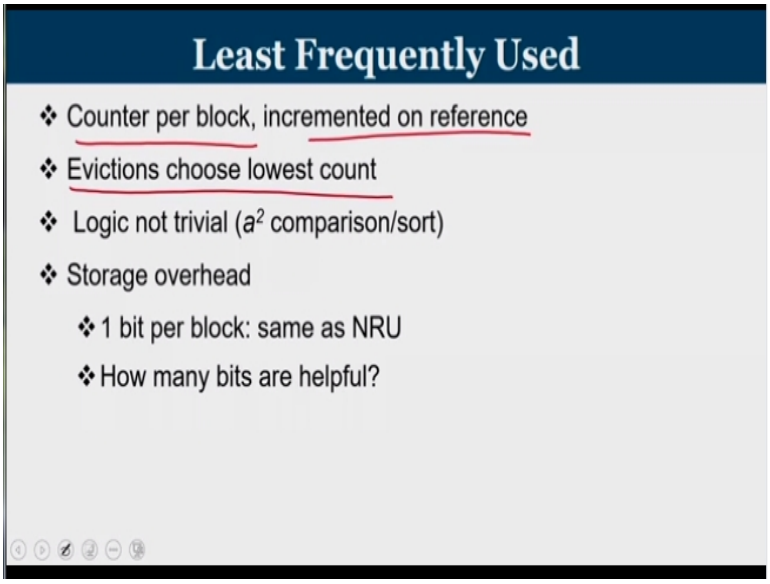
We now look into another block replacement technique which is called re-reference interval prediction. So it actually extends NRU to multiple bits, so for each block for example let us say each of the block is associated with a 4 bit value. Let us say start with a value 8, so this 4 bit value can range from 0000 all the way up to 1111. Here the 11 indicates a very heavily used block and 0000 indicates a very lightly used block, when a block is been brought into cache you start the middle value, let us say I start with 8, you promote on hit.

So when I am going to access it again when there is a re-reference that happens this value will become 9 and then it will become 10, then it will become 11 like that. That is promotion generally happens on a hit, now when will demotion happens. Let us say at the end of every 16 cycles I am going to bring down the value by let us say - 4, so this value is already 7 when I subtract 4 the value will become 7.

Now think of a case, I was 11 at that point of time and then a constant value was subtracted. Now imagine a scenario after bringing into the cache with an initial value 8, I was never re-referenced. So in this case when I subtract 4 from it, it eventually become 4, now when I have already 4 at the end of a next time a poke again I am subtracted to 0. So the blocks which are not referenced will eventually bring their value down and all other blocks the value will go high.

So this by this we can predict a near - immediate, immediate and distant re-reference. So based upon re-reference your value is updated if you are not referenced at all then value is demoted over the time.

(Refer Slide Time: 32:21)



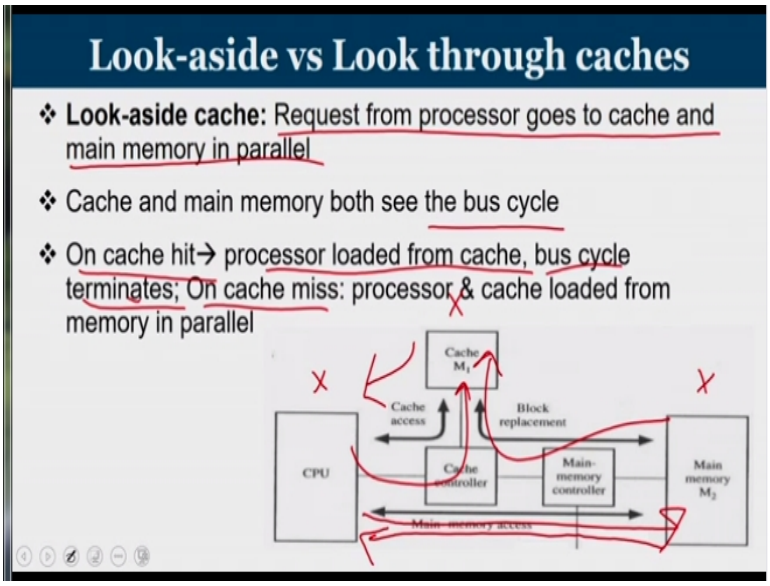
Now so far we have looked into aspect of recency of usage, now let us try to understand what is called frequency of usage. So our purpose in the case of least frequently used is to find out a number of times a block was re-referenced after it was brought into the cache. So if you wanted to find a count, the number of times that a block was a reference, you require a counter. So every cache line is associated with a small register whose value is been updated or incremented for every access.

So a counter is there per block and it is incremented on reference and you how to choose the lowest in the counter during evictions and the logic is not reveal you require roughly

$$A^2$$

comparisons. So since it is absolute number to find out what is minimum we have to sort them and sorting is having it is own overhead.

(Refer Slide Time: 33:18)



So we learned about different types of block replacement algorithm starting from random first in first out, last in first out, LRU, Pseudo LRU, optimal, NMRU, LFU, RRIP. These are the different types of block replacement algorithms that we have learn today.

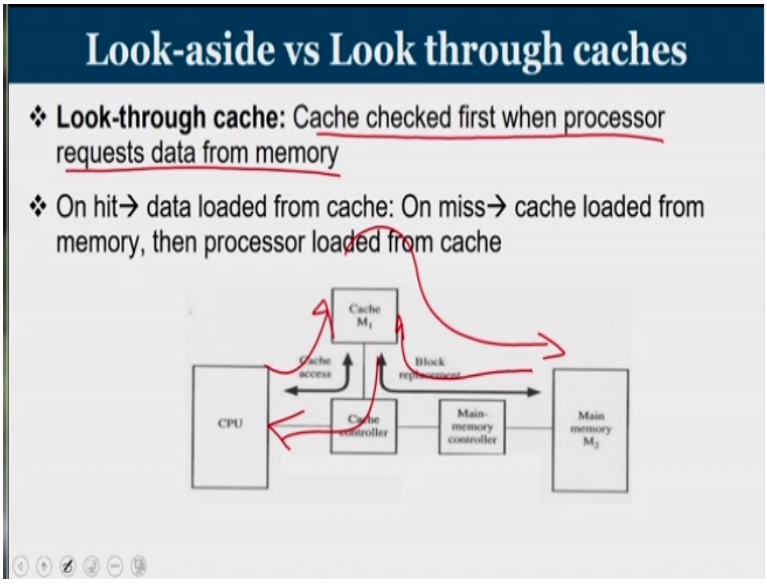
Now let us look into some other properties of cache they are look aside cache and look through cache. So look aside cache you request from processor goes to cache and main memory in parallel. So this is your processor you have the cache and you have the main memory, so whenever I am going to put a request a cache controller will take it to the cache memory as well as it going to main memory in parallel, what is a advantage. Cache and main memory both see the bus cycle if there is a cache hit processor is loaded from the cache this is what happens. The bus cycle which has gone to initiate things in the memory is been cancelled.

But if there is a cache miss you need not trigger main memory already main memory is been informed this is the address that you need. So when there is a parallel look up in both cache and in main memory if the cache gets a hit, cache will return the word if cache get a miss within sometime the main memory is going to respond in the word, what is the advantage. Main memory will supply the word upon a miss and the same time it is going to feed in to the cache.

Since the lookup is parallel the moment he come to know it is a cache miss I am not initiating anymore request to main memory. So the request from main memory has already gone, so this

parallel lookup will save time when there is a miss but if the majority of the hits the lot of memory operations has to be cancel because since I got a hit already whatever happens in the main memory it has to be cancelled.

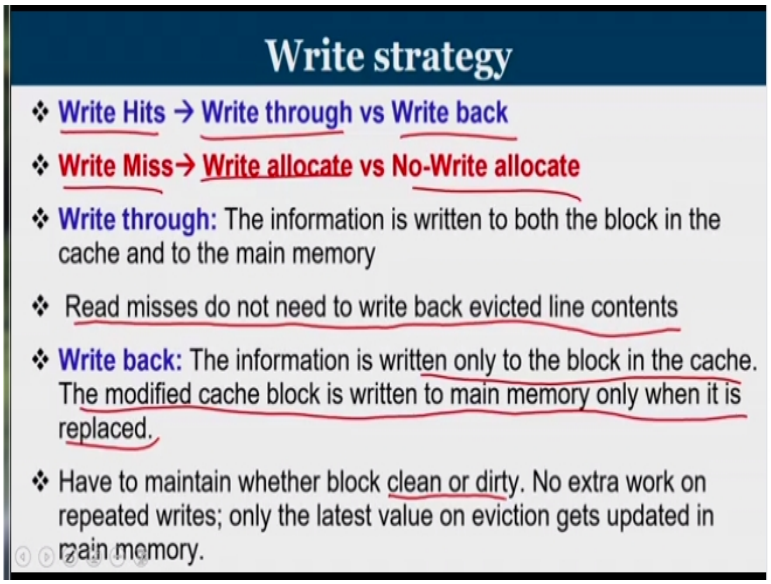
(Refer Slide Time: 35:21)



The other version is called look through cache, cache is checked first when processor request a data from the memory that is what you see you look into the cache. If it is a miss then cache will trigger what is there in main memory, blocker is replace in the cache and then the cache will supply the corresponding word. So on a hit data is loaded from the cache on a miss cache is loaded from the main memory and then the processor is loaded from the cache.

So here there is parallel lookup of main memory is not there during a miss then only your miss penalty is going to stop. So the basic difference is in the case of a look aside cache a miss penalty is less because of the parallel lookup in cache and in main memory. But in look through cache we are not involving the main memory we are hoping that the cache is going to return the word but your miss penalty is going to be higher than look aside cache.

(Refer Slide Time: 36:14)



Now we move on to write strategy, so where you access cache memory, we access cache either to perform a read operation means I have to take some value from the cache or it can be for a write operation. Now if it is a read operation I can have 2 outcomes read hit or read miss we have seen what happens in the case replacement another. The other version is called write operation and that also can have 2 versions write hits which works exactly same like read hits.

But there are something called write misses, so we will deal with what will we do on write during write hits and write miss. First is based upon what we do on write hits caches are divided into 2 they are write back cache and write through cache. So based on write hits you have write through versus the write back cache. And based on write miss caches are of write allocate caches and no-write allocate caches.

The what is write through cache, the information is written to both the block in the cache and to the main memory. So whenever you write something in the cache then and there itself main memory is also updated thereby both the memory copy as well as the cache copy is consistent. But during read misses you need not write back the evicted cache line, so think of a case you have a processor, you have a cache and you have a main memory any write that you perform on a block it is updated in the main memory as well.

Now think of a case had some block replacement happen should I simply overwrite the block or should I write the updates. Because any write that is happening on that particular block already it

is updated in main memory, so main memory copy and cache copy are exactly identical. So when there is a replacement that is needed the victim block need not be updated in main memory, that is what it is been mentioned.

During read misses there is no need to write back the evicted cache lines, coming to the second category of cache which is called write back cache. The information is written only to the block in the cache, so the main memory and cache will have different copies of the same block. The modified cache block is written to main memory only when it is replaced, so here the strategy is processor, cache and main memory, processor is going to write only to cache.

So whatever changes that is happening it is strictly local to the cache, the main memory is having a totally different value of the given block. So they are now not consistent they are not identical copy but as long as processor is going to write and read from the cache processor always get the latest value but memory is having a different value it is called a stale value. Now when this block where the writing is happened when it is going to be evicted out during a cache block replacement.

That is a time when we have to make sure that the cache block is been may consistent with the memory block. So during a write back operation the contents of a cache is been updated to main memory. Now do I need to do this process for every eviction, no we need to do this updates only on those blocks which are changed or modified after bringing into the cache. Let us say there is a block that is there in the cache I have not performed any write on it.
So eventually when it is been going out of the cache, cache block and main memory both are same there is no need to overwrite the main memory. So how will you know an extra bit is been used which is called a dirty bit or a modified bit. So anytime processor writes on a write back cache this bit is been set meaning that particular block is modified. So when you are going to evict out this block have to update main memory.

So we have to maintain whether a block is clean or dirty, so no extra work is done on repeated writes. So when you perform n number of writes on a cache block in the case of a write through cache each of this write main memory is updated. So when you perform 100 writes on a cache

block the main memory is not updated at the end of the whole process when the cache block is evicted the result of the 100th write the final version only is getting updated.

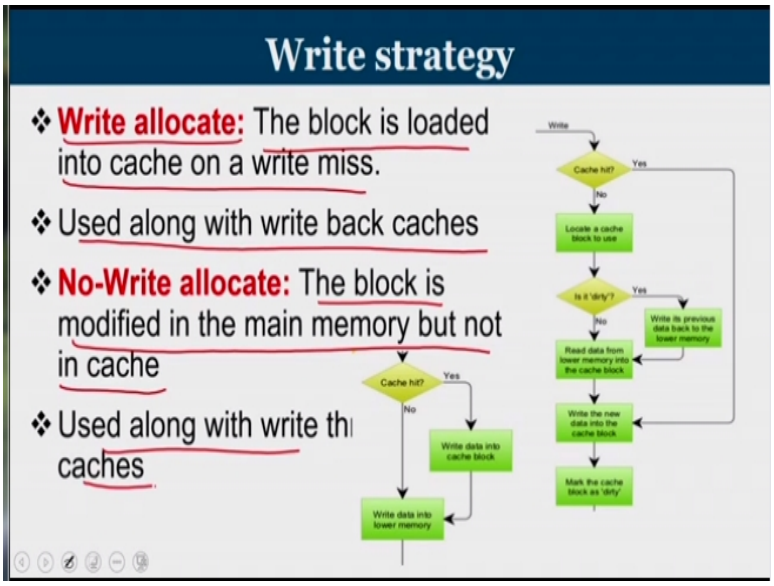
So think of a case that you are going to increment the value of a,

$$a = a + 1$$

inside a loop. In the case of a write through cache when the value of a initially which was 10 let us say it becomes 11 main memory is updated, it becomes 12 main memory is updated, it becomes 13 main memory is updated. But if it is a write back cache from 11 to 12 to 13 to 14 all updates happen only in the cache and when the block that contains it is value it is updated value is been evicted out the dirty bit will tell that this is a dirty data or the cache block is modified.

So let us say then final value of A was 100 only 100 will be reflected in the main memory all intermittent changes are strictly local to the cache. That is a beauty or that is a comparison of a write back and the write through cache.

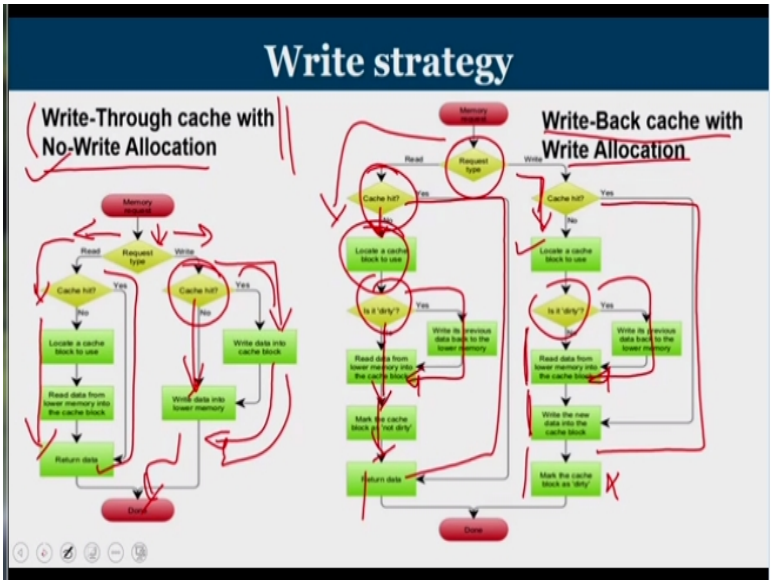
(Refer Slide Time: 41:18)



Now coming to write strategy you have a write allocate caches, so when there is a write miss the block is loaded into the cache on a write miss. So when you encounter a write miss you are going to have, so when the block is encountering a write miss bring the block from the main memory assign a space in the cache. So during the process you may evict out somebody and then you perform the write and generally this write allocate caches are used along with write back cache.

Similarly for no write allocate cache the block is modified in the main memory but not in the cache you directly go and write in the main memory, it is used along with write through caches.

(Refer Slide Time: 42:03)



So one normal combination is write through cache with no write allocation and then write back cache with the write allocation. Let us try to understand, so when there is a request that is coming first you check whether it is a read operation or a write operation. If it is a hit let us say if it is a cache hit, yes then write the data into the corresponding cache block, write the data into the main memory why it is so because it is a write through cache and then it is finished.

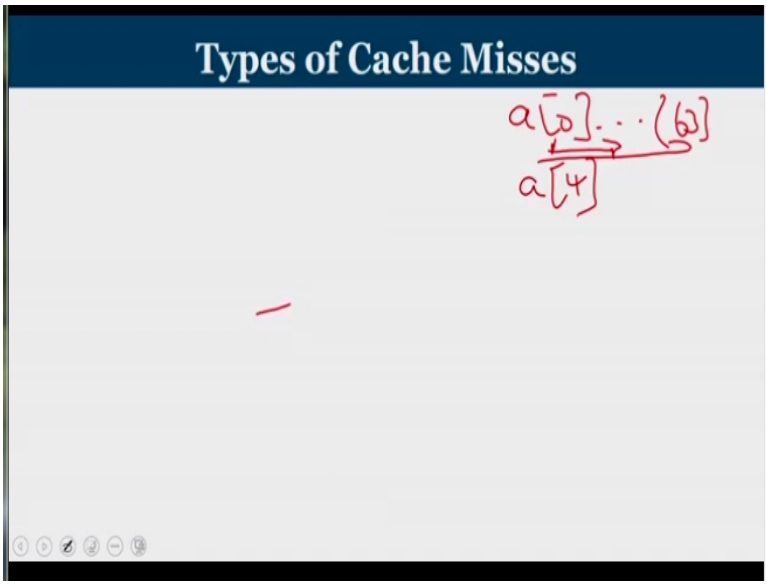
Since it is a no write allocates if you come to know it is a cache hit then you go and write if it is a cache hit no means it is a cache miss then you write directly in the main memory. Now if the operation is read you see whether it is a cache hit or not yes you return a data, no you locate a cache block to use, read data from lower memory into the cache block that is what you do in a read miss and then you return the data.

This is exactly what happens in a write through cache with no write allocate. Now it is a write back cache with write allocation, first you see what is the request type let us say it is a write operation see if it is a hit, yes it is a hit write the new data into the cache block and then tell there it is dirty. If it is a write miss first you have to find out locate the cache block to use see whether

it is dirty if it is dirty you write the data back to memory, read the data from lower memory into cache and then write data onto be make it dirty.

On the other side if it is a read operation check whether it is a hit or not if it is a read hit, yes you written the data. If it is a read miss you have to find out the block that is to be replaced is it dirty if so write into main memory and then read the corresponding contents, mark the cache block as not dirty because now it is a newly block that has come and then you return the data.

(Refer Slide Time: 44:10)



So we have seen different types of cache now, look aside cache where main memory and cache is parallely looked into. Look through cache, where first to look into the cache if there is a miss then only you go to cache and then based upon the writes the write hits, I can have 2 types of cache write back cache where all updates happen in the cache only. And write through cache where cache and the next level of memory is having the same copy.

Now during write miss should I find space in the cache bring a block and then write only there that is called write allocate cache or it can be write no allocate cache as well where you are you have the freedom to go directly writing in the main memory. We know that while giving a request to cache it can either result in a hit or it can result in a miss. Now once you get a miss we have to understand why the miss happened that is called analyzing the misses.

So first we have to understand what are the possible ways in which you can get a miss, so misses are classified into 3. They are compulsory miss, capacity miss and conflict miss, when you give an address for the very first time, it may not be present in the cache. And that is what is called compulsory miss, when you have a huge array let us say a of 0 up to a of 63. So consider an array a of 0 up to 63, now in this case an access to a of 0 will be surely a miss.

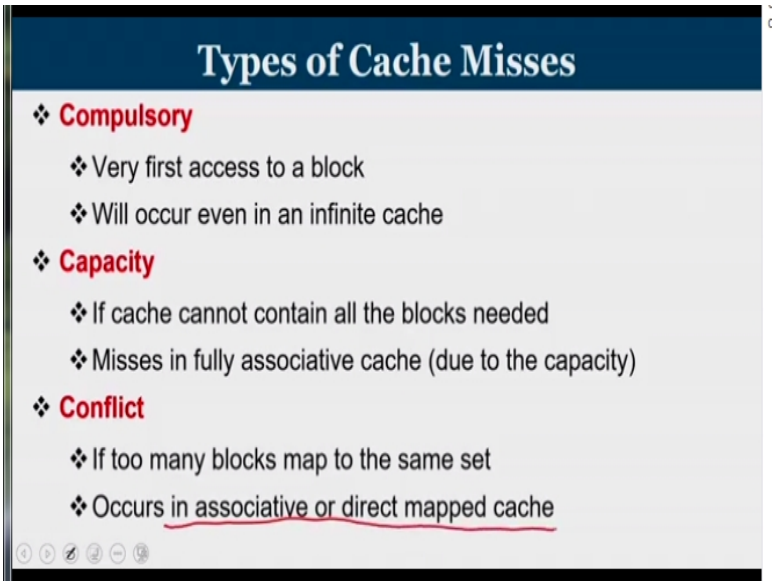
So you go to main memory bring a block of data, now here the block is important the block may contain a of 1, a of 2, a of 3 maybe 4 words you are going to bring. So a of 1 to a of 3 will not result in a miss and then when you are going to access a of 4 again I get a miss. So during encountering a miss for a of 4, 4, 5, 6 and 7 are brought. So you get a miss in 8 like that the compulsory misses are going to be there every time.

Now there is another category of miss that is known as capacity miss, consider a case where fully associative cache. So you surely you will have compulsory miss let us say I need a program or I am going to access an array which is of 100 words. Now I have 4 cache blocks, each can accommodate 4 words, so total 16 words can exist in the cache. Consider the cache is fully associative, so you bring a of 0, a of 0 to a of 4 is there, so a of 0 to a of 3 is there, a of 4 to a of 7, 8 to 11, 12 to 15.

So the 16 words are now existing when you go to 17 the new one, 17th word cache is already full. Since I am using fully associative there is no restriction on where to map, I have stored in all possible places that I could still I encounter a miss at a of 17. And that I may have to replace and that is what is known as a capacity miss because the cache is already full. The third category is called conflict miss, I brought somebody a, then I request somebody called c and that is say you imagine c is mapping to a.

Then I take off a and put c there, when a is needed again it is not a compulsory miss, it is already come once now I am bringing it once more. Because it was thrown out by somebody and this is called conflict miss.

(Refer Slide Time: 48:03)



So compulsory miss is a very first access to a block that is called compulsory miss, it will occur even in an infinite cache. Capacity miss is if a cache cannot contain all the blocks it needed generally happen in fully associative cache that is called capacity miss. And conflict miss if too many blocks map to the same set then even though certain blocks which are brought it will encounter a miss in the near future because after bringing in they are thrown out by another block mapping to the same set, so this occurs in associative and direct mapped cache.

(Refer Slide Time: 48:39)

With this we come to the end of this lecture just try to summarize what we have learn today. We started with different cache block replacement techniques and then we learned about different types of caches based upon what do you on during a read and then write operation. And then we

learned different categories of misses like capacity miss, compulsory miss and conflict miss. There are also tutorial videos in this week wherein how you solve numerical problems associated with cache memory related techniques.

So solving such kind of questions will give you better clarity on the concept that we have learned in the last couple of videos. I have also request you to go through as many as numerical problems given at the end of the textbook to get more confidence on the subject, wishing you a good learning, thank you.