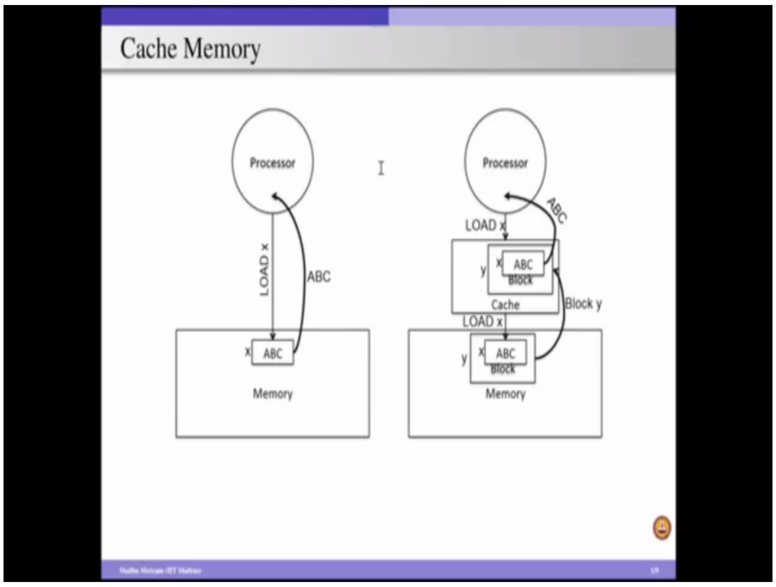
Computer Architecture

Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module – 03
Lecture - 07
Memory Hierarchy Design (Part 2)

So, in this module we discuss the basics of cache memory, which include how the cache is organized and what happens when there is a miss, how to select a victim block in the cache. And what happens when we want to write block to the cache. And also we look at the issues related to how to access the cache memory and so on. So before that we start with an example why we actually require a cache memory.

(Refer Slide Time: 00:47)


Cache Memory

Consider a scenario where a system without a cache memory. So, when processor wants to read the data which is stored at an address x in the memory. So, it sends a request, load request to the memory and the memory supplies the data stored in the address x. Let us assume that the data stored in that particular address is ABC. We know that memory access takes lot of time hundreds of cycles. So, this operation will be very costly.

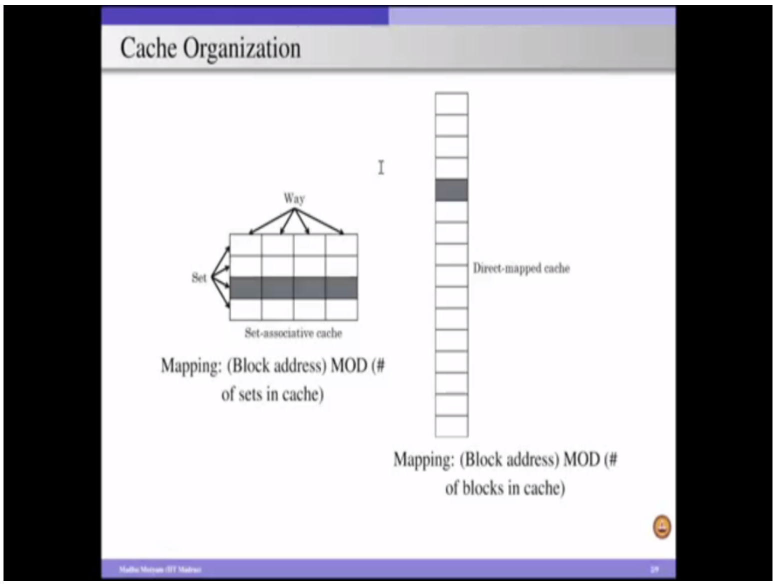
And every time if you are going to the memory and spending hundreds of cycles, our overall performance will be degraded significantly. To minimize the performance penalty what we can do is we can consider a faster memory between the processor and the main memory which is called as a cache memory. We send the request from the processor to load the data which is stored at an address x. We first search in the cache and the cache is initially empty. So, there is no data, we will go to the memory and get the data, but because we are spending anyway hundreds of cycles to get the data from the memory, rather than reading only the required word of data, we read a collection of data items which are needed to this address x.

So, we bring this chunk of data from the memory and store it in the cache memory. So, that in the future, if processor requires any data in this particular chunk, the cache memory can service. So, effectively the cache memories improve the overall performance of the system. DRAM cell takes more time to access compared to an access from an SRAM cell. So, that is the reason typically why cache memories are designed using SRAM technology. The DRAM memory density is much higher where it typically takes one transistor and a capacitor for one bit of information compared to 4 to 10 transistors in the case of SRAM cache memory.

So, because of that reason the size of the cache is also considered typically smaller. And another point is, the access time of the cache also depends on the size of the cache. So, if the bigger size cache is considered then automatically it is going to take more amount of time to supply the data to the processor. Keeping all these reasons, typically we consider the cache memories of reasonable size compared to the size of the main memory. In addition to that because the cache memory typically placed on the chip and the chip real estate is very costly.

So, we cannot dedicate too much space on the chip for these cache memories and so on. And because we are bringing chunk of data from the memory, to minimize the number of accesses to the memory as long as cache supplies the data required by the processor. So, what is the size of the block we have to consider? Typically, we consider a block size either as 32 bytes or 64 bytes. If you consider the smaller block size the applications may not exploit the spatial locality properly. At the same time, if the block size is very large then the number of blocks stored in the cache memory is limited and as a result the overall conflicts miss rate may increase. So, that is the reason why typically the current processors are considering a block size of 32 bytes or 64 bytes. So, having discussed, why we require a cache memory.

(Refer Slide Time: 04:43)



Cache Organization

Mapping: (Block address) MOD (# of sets in cache)

Mapping: (Block address) MOD (# of blocks in cache)

Now, we will look into the organization of the cache memory. As I mentioned previously, we bring a block of data from the main memory to the cache. So, effectively the data transfer happens at a block level granularity between this multiple levels of the caches or from the memory also. Effectively, we can view a cache as a collection of blocks. Now, how do we organize these blocks in the cache? There are three ways in which we can organize the cache memory.

One is a set associative cache where the total numbers of blocks are arranged in collection of sets and each set is accommodating a fixed number of blocks. So, effectively a cache is divided into sets and the ways. In this particular example, we consider 4 sets and each set is accommodating 4 ways. And that is the reason why this is called as 4 way set associative cache. So, once we have this type of organization, now how do we map a cache block to a particular location in the cache.

So, for that we use this mapping formula. Take the block address and apply mod operation with the number of sets in this cache. For example, if my block address is 20 and because here the number of sets is 4, so automatically this block address 20 is stored in set zero. And once we fix a particular set for a given block, within the set the block can be placed anywhere either we can place the block in way zero, way one, way two or way three. So, the advantage with the set associative cache is, if multiple blocks are mapped to the same set, still we can service the request by accommodating all these blocks in a particular set, as long as the number of blocks we accommodate is not more than the total number of ways of that particular cache.

If you want to accommodate five blocks in a set, we can at most accommodate four. So, the, to insert the fifth block we have to evict one of these existing 4 blocks. So, for that we use cache replacement policies that we discuss after a couple of points. So having discussed the set associative cache, now we consider two extremes of the set associative cache. The first one is we restrict the number of ways per set. Let us assume that each set accommodates only one way. Now, in this particular example, we have 16 blocks which are arranged in 4 sets and each set is having 4 ways.

Now, when I restrict the number of ways per set to be one, then automatically I am going to have total 16 sets and each set is having only one way. Effectively it can accommodate only one block per set and this organization is called as Direct mapped cache. So, in this direct mapped cache, to map a particular block to a particular location, a fixed location in the cache, all we have to do is we apply the mod operation on the block address with the total number of blocks in the cache.

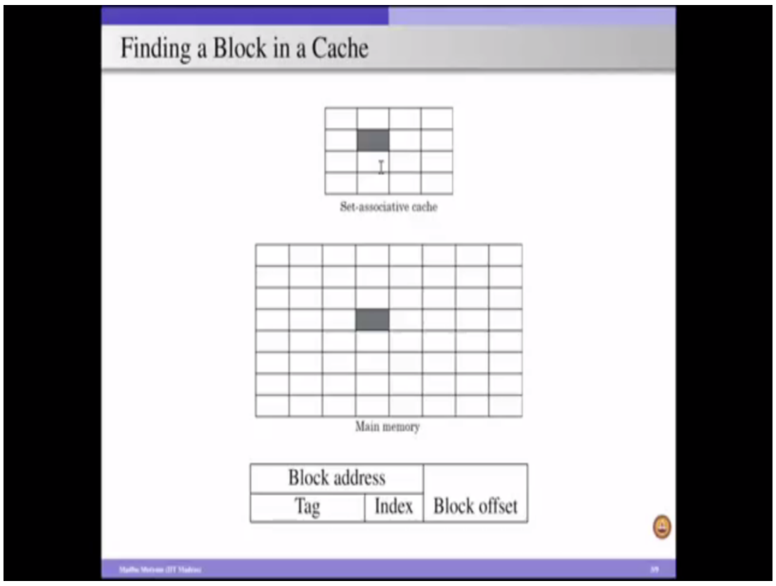
Again, in the previous example, if I want to store a block address 20 into this direct mapped cache, so 20 mod 16 because there are 16 blocks in the cache. So, which is effectively, we are going to store that at block 4. So, starting with block zero, block two, block three and so on. So, we are going to put this, the twentieth block, in block 4. So, compared to set associative cache, here because we have only one location per set if application requires multiple blocks mapped to the same location, then we are going to get too many number of misses.

So, it is actually not efficient in terms of the cache miss rate, but it is efficient in terms of the search time. If we want to locate a particular block, all we have to do is apply just mod operation and we will get the block directly, whether the block is present or not we will get the information, but whereas, in the case of set associative caches, when we apply the mod operation we just get only the set. And after getting the set, we have to search in that set all the ways in the set we have to search to see whether our requested data is available in cache or not.

So, which is going to take slightly more time compared to the search time in the direct mapped cache. Now, we consider the other extreme where we consider a single set in the cache, but this set can accommodate all the blocks of the cache. Again considering this example where 16 blocks are there in the cache and if I organize in a way where we consider only one set and this set is accommodating 16 blocks, then we are going to get the other organization which is called as fully associative cache. And this fully associative cache, the block can be placed anywhere whether you can place that on the way zero or way one, way two or at the end.

So, anywhere you can place the block in the cache. So, as a result it will be efficient in terms of utilizing the cache space, but it is inefficient in terms of searching for a block because when we keep a block anywhere and if you want to search for that block we have to search all the blocks in the cache, to know whether a block exist in the cache or not. So, that is the reason why search time will be very complex in this particular case and also it consumes significant energy as we have to search every location in the cache. But the current processors are typically using either direct mapped caches or set associative caches, because of their advantages in terms of reducing the cache miss rate and simple search. So having discussed this cache organization, now we will see how to locate a block in the cache?

(Refer Slide Time: 11:42)



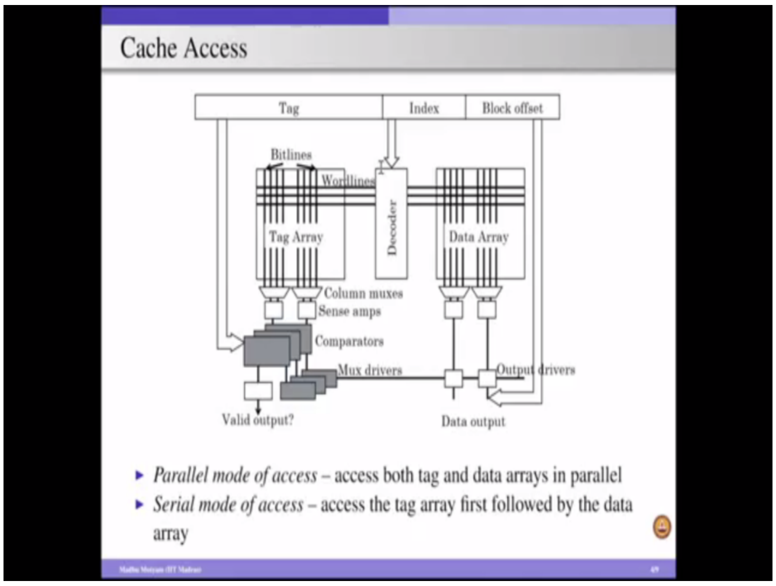
As I mentioned earlier, the cache memory size is very small compared to the main memory size. And also the granularity at which the data transfer happens between the cache and the main memory is at the block level granularity. So, multiple blocks of the memory can be mapped to a single location in the cache. So, how do we locate a particular address block in the cache? So, in this particular example, we considered a set associative cache, but this can be applied for direct mapped cache or fully associative cache. Having the address available from the processor we can divide that address into three fields.

LSB few bits are going to specify the block offset, next few bits are going to specify the index and also called the set index. MSB remaining bits are going to specify the tag of the particular block. So, having divided the given address into three fields, in the set associative cache, we use the set index bits to locate a particular set and within that set we search all the blocks present in that set using the address generated by the processor with the addresses associated with the blocks present in that particular selected set. If there is a match that indicates that or block is present in the set. If there is no match then the cache is not having this particular requested data.

In the case of direct mapped cache because each set is storing only one way or one block of data. So, effectively we are going to use both the tag and the index together and that is called as a block address and we apply mod operation on this block address with the total number of blocks present in the cache to locate a particular block. And similarly, in the case of fully associative cache also because we can store anywhere.

So, we have to search entire cache with this block address to locate the requested block whether it is present or not. So, we mention that processor generates an address, takes an address and uses the tag bits, uses the set index bits and go to a particular set and search for a particular block within that particular set. But to understand all these things we actually need to understand how the actual cache operation happens inside the cache. So, for that we need to know the cache micro architecture in detail.

(Refer Slide Time: 14:34)



Cache Access

- *Parallel mode of access* – access both tag and data arrays in parallel
- *Serial mode of access* – access the tag array first followed by the data array

If we look at the cache microarchitecture, it consists of data array and the tag array. The actual data is stored in the data array whereas, the tag array stores the meta data associated with this data. The meta data consists of the tags, valid bit, the dirty bit and all other necessary information. And we also have a decoder which is called as a set decoder and we have multiplexers, sense amplifiers, comparators and drivers. Now, when a processor generates an address we partition that address into three fields tag, index and the block offset.

We take the index bits and supply that to the set decoder and set decoder will select one of the sets effectively this output of the decoder is connected to the word lines in the cache and based on the set index bits, one of the word lines will be activated and also we activate the bit lines. The intersection of the word line and the bit line will give you an access to an SRAM cell, which will supply one bit of information. So, when the word line is activated and all the bit lines are activated, if we read the data from all the intersection points and this data goes through the column access sense amplifiers. The sense amplifiers will ensure that no full testing happens on these bit lines.

So, because once there is no full testing happens, your dynamic energy consumption will be minimized. And once we read the data, we give the data whatever we read from the tag array to the comparator as one in input and the other input we supply from our processor generated address from the tag field. And if there is a match, that indicates that our requested data is present in the cache. Then we read the corresponding data from the data array. Finally, we read a block of data because our tag is actually associated with the cache block. Once we read the entire block, we have to supply the requested word to the processor because the processor always requests word of data, it will not request the entire block of data.

And remember the block size is 64 bytes or 32 bytes and processor may request 32 bits or 64 bits of data. So, to select a particular word in the selected block, we apply block offset and then supply the requested word of data to the processor. So, as a result, there are lots of a sequence of operations happen, starting from the processor generated the address all the way up to supplying the data to the processor. And in this we can either perform reading from the tag array and data array in parallel or we can perform the reading from the tag array first, followed by reading from the data array.

So, if I perform the operation parallelly in both the tag array and the data array, then this is called the parallel mode of access which is efficient from the point of access latency because our two operations are overlapped.

$$Overall\ Access\ Time = max(Tag\ array\ access\ time, Data\ array\ access\ time) + Access\ of\ time\ of\ other\ peripheral\ components\ \text{in}\ the\ cache\ memory$$

Though it is efficient from the access time point of view, because it is accessing all the ways in the cache, it is not efficient from the energy point of view.

Consider a scenario, if the cache is having four ways per set, in other words it is a 4 way set associative cache if we consider we know that a requested block can present at most in one way in a set. So, that indicates that, when we are accessing a 4 way set associative cache parallely, we are reading 4 tag blocks, 4 data blocks simultaneously and searching all these 4 tag blocks with the tag of the supplied address. And if at all if there is a match only one tag block is going to give the hit. So, that effectively wastes the energy due to accessing the three data blocks. So, from the energy point of view parallel mode of access is not efficient, but from the access latency point of view the parallel mode is efficient. So, other alternative is first access the tag array and compare tag given in the address with the tags available from the tag array reading.

And if there is a hit then only access the data array. If I do this effectively I can minimize the access energy wastage due to accessing the other blocks in the data array. Though it saves the access energy from the access latency point of view because these operations are serialized and

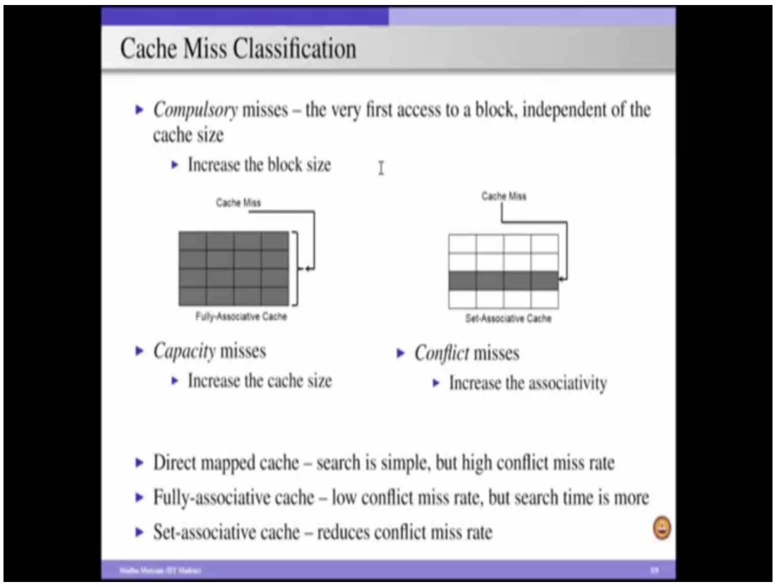
$$Overall\ Access\ Time = Tag\ array\ access\ time + Data\ array\ access\ time + Access\ time\ of\ all\ other\ components\ \text{in}\ cache$$

which actually increases the overall access time. Now, so where do we apply this parallel mode of access, and where do we apply the serial mode of access.

Typically the caches which are very close to the processor require the data as fast as possible. In those cases we apply parallel mode of access. And whereas the caches which are very far from the processor, we are actually applying 8 way associative or 16 way associative caches in those levels of the caches, where it is not necessary to access the tag and the data arrays parallely. So, in such cases, we can apply a serial mode of access. In other words, typically for L1 cache we can apply parallel mode of access whereas for L2 and L3 caches we can apply serial mode of access.

Now, we discuss the cache miss classification, when a processor generates an address to read the data from a particular location in the cache, the cache may have the data or it may not have the data. If the data is not there in the cache then we consider it as a cache miss. And these cache misses can be classified into three groups, the first one is called as the compulsory misses or cold misses.

(Refer Slide Time: 22:00)



Cache Miss Classification

- *Compulsory* misses – the very first access to a block, independent of the cache size
  - Increase the block size

Cache Miss
Fully-Associative Cache
Cache Miss
Set-Associative Cache

- *Capacity* misses
  - Increase the cache size
- *Conflict* misses
  - Increase the associativity

- Direct mapped cache – search is simple, but high conflict miss rate
- Fully-associative cache – low conflict miss rate, but search time is more
- Set-associative cache – reduces conflict miss rate

And these are the misses that happen for the very first access to a block. Initially the cache is empty when the processor generates an address, you cannot find anything in the cache, we incur a miss because there is no data in the cache and this miss is called as a compulsory miss. The number of compulsory misses is independent of the cache size. Even if you consider 32kb cache, 64kb cache and 128kb cache these compulsory misses will be the same. Only one way you can reduce this compulsory misses and that is by increasing the block size, but remember if you increase the block size the total number of blocks stored in the cache will be reduced and as a result the application especially in the case of data accesses may not exploit the locality properly.

And which in turn can increase the overall conflict miss rates. The second class of cache misses is the capacity miss. For example, consider a fully associative cache where you can keep a cache block anywhere in the cache. Now, assume that a scenario where the entire fully associative cache is full with valid blocks of data. If processor requested the data which is not there in the cache, now we need to bring in the new data from new block of data from the lower level caches and we supply that to the cache, but now this cache is full. So, we evict some block from this fully associative cache and keep the new block into this particular location.

And now after that if processor requests a data to the previously evicted block. So, we incur a miss and this miss is happened because we were not able to keep that particular block in this cache due to the lack of space or lack of capacity and this is called as the capacity miss. A miss occurs because there is no space in the cache. So, one way to reduce the number of capacity misses is by increasing the cache size rather than considering 32 kb of cache is I consider 64 kb of cache, my cache capacity is doubled. So, as a result the capacity misses can be reduced significantly. And the third class of cache misses is called as conflict misses.

Consider a set associative cache, where we know that the blocks are mapped to a particular set. Let us assume that processor generates an address which is mapped to the third set in this particular example set associative cache. And also assume that this set is having all valid blocks. Now, the new data cannot be there in the cache and cache incurs a miss and when we bring in the data from the lower level of the caches, we evict one of the blocks from this particular set and store this new block in that. And after that if processor generates request to the evicted block we incur a miss. That miss is called a conflict miss.

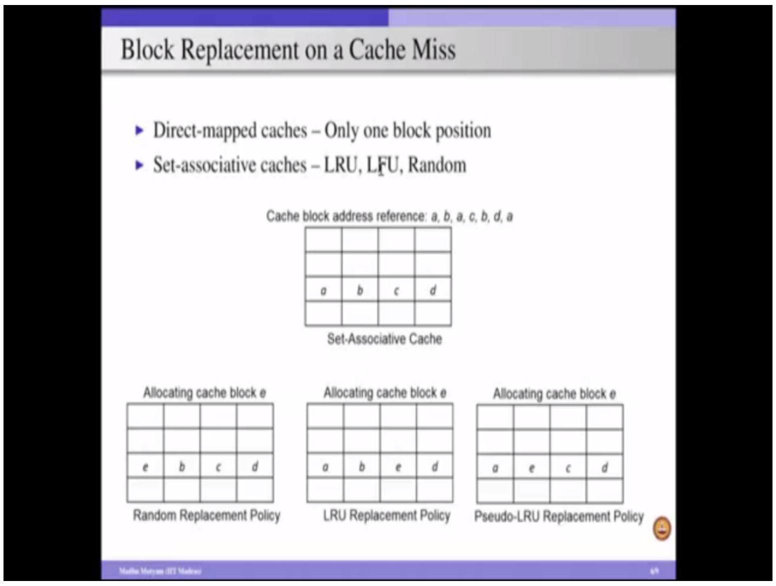
This says that, block is not found in the cache because there is a conflict in the particular set, even though the other sets have empty spaces where a block can be placed, but because of the set associative property blocks are always mapped to a particular set. So, as a result we incur a miss. This says that misses happen because of lack of space in a given set that is called as conflict miss. To reduce the conflict miss rate, all we have to do is increase the associativity of the cache, rather than considering a 4 way set associative cache, if I consider 8 way set associative cache I can reduce the overall conflict miss rate.

So, among all these types of cache misses, the compulsory misses are having very low fraction of overall misses whereas, the conflict misses have the highest percentage of misses. So, in other words the conflict misses are predominant and we have to optimize our caches such a way that the conflict miss rate is reduced to improve the overall performance. And when we consider direct mapped caches, we know that the search is simple, but the conflict miss rate is very high because in the direct mapped cache each set can accommodate only one block of space.

On the other hand, fully associative caches in which we can keep blocks anywhere in the cache. So, which incurs very low conflict miss rate, but the search time is very high. And set associative caches reduce the conflict miss rate significantly compared to the direct mapped caches. And also it has low search time compared to fully associative caches and because of all these advantages typically set associative caches are predominantly used followed by the direct mapped caches in the current processors. So, when I say there is a miss in the cache, now we have to bring in new block of data from the lower level of the caches and keep that in particular cache. So, if I consider a set associative cache where a set will have multiple blocks. If it is a 4 way associative cache, set can accommodate 4 blocks out of this 4 blocks which block to be evicted to make a room for the new block that is brought from the lower level of the caches or the memory.

So, to do that, we have to consider cache replacement policies. In the case of direct mapped caches, since each set accommodates only one block, there is no question of which block to be evicted because only one location, you have to evict that particular location and store the new block into that location.

(Refer Slide Time: 28:36)


Block Replacement on a Cache Miss

- Direct-mapped caches – Only one block position
- Set-associative caches – LRU, LFU, Random

But whereas, in the case of set associative caches, variety of techniques proposed in the literature, but in this discussion we are going to consider three policies which are random replacement policy, least recently used policy and pseudo least recently used policy. So, consider a set associative cache with 4 sets and each set is having 4 ways and also consider a cache block address reference pattern a, b, a, c, b, d, a and all these addresses are mapped to the third set in the cache.

And so when processor generates an address for a, we are keeping the corresponding block in the first way of the third set. When the processor requests the address location b, we are going to keep the block in the second way and we will continue. And after the processor generates an address for d, then we keep d in the fourth way of the set. And the state of the set associative cache after referencing the address location d is like this, where the third set has 4 blocks with the data corresponding to a, b, c and d.

When the processor generates an address for ‘a’ again because ‘a’ is already there. So, it supplies a data from the cache and it is treated as a hit. Now, consider a scenario when a processor generates an address location ‘e’ and that corresponding to a particular block now how do we store? So, if I consider random replacement policy, because the block which consists of address location ‘e’ is not there in the current cache. So, we have to evict one of these 4 blocks. And when we want to evict, so, if I consider a random replacement policy we can pick a block at random by generating a pseudo random number.

And in this particular case, we consider the first block is evicted and the block containing the address ‘e’ is stored in this particular location. If I consider least recently used policy, I always select a cache block which is least recently used in the past, among all these blocks. So, block containing the address ‘c’ is least recently used and we evict that particular block and keep the block containing the address ‘e’ into that. Mainly the least recently used policy is working on a principle that, if a block is not accessed recently in the past there is a high chance that block will not be accessed in the near future also, with that principle LRU policy is designed.

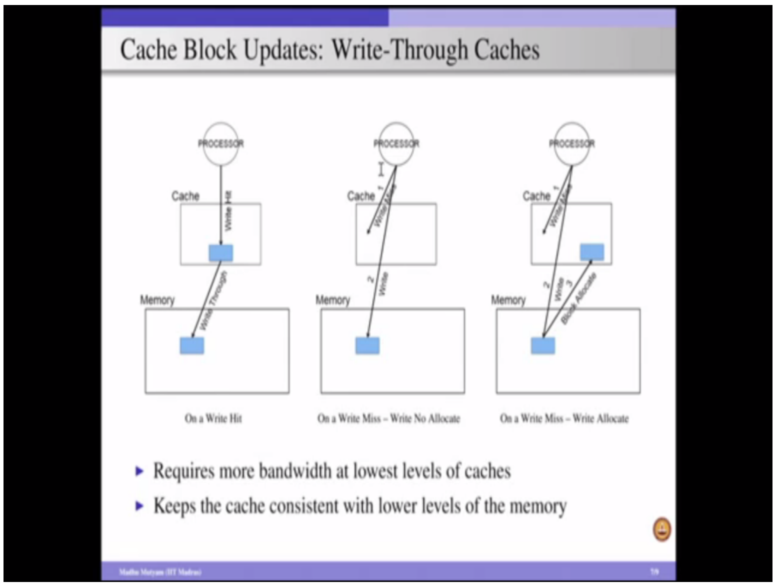
And it always selects a block which is not recently accessed in the past, but how do we count the time, how do we know which block is not least recently accessed or which block is recently accessed. To do that we associate a ‘Log N’ bit saturation counter for each of the blocks, where ‘N’ is the associativity of the cache. In this particular example, we consider 4 way set associative cache. So, we consider a two bit saturation counter for each of the blocks. And whenever a block is referenced, the counter is reset to zero and all other counters associated with the other blocks in the set are incremented by one. And whenever we want to select a victim we always come through all the counters and whichever counter is giving the maximum value that block will be selected as a victim.

So, the LRU policy achieves good cache hit rate, but the problem with this policy is as the associativity increases the maintenance of the overhead due to maintaining this replacement policy increases. If we consider a 16 way associative cache we have to consider 4 bit saturation counter for each of the blocks effectively 16, 4 bit saturation counters we have to consider and that is going to incur significant overhead. And also every access we have to update the counter values. To minimize the overhead associated with this LRU replacement policy, we can come up with an approximated LRU policy which is called as pseudo LRU, pseudo least recently used policy.

Where rather than considering ‘Log N’ bit saturation counter, for each block we consider a one bit flag. And whenever a block is accessed the flag will be set and we will continue setting the flags of access blocks. And once all the blocks in a set having their flag bit set then we reset all the bits. And whenever we want to select a victim we either scan from left to right or right to left and select the first block whose flag bit is a reset. So, this actually incurs only one bit of overhead per block compared to ‘Log N’ bit overhead for an n way set of set of cache if we are using an LRU policy. But compared to the pseudo LRU, the LRU policy will achieve better cache hit rate.

The current processors for example, Intel Sandibridge and others so typically use the pseudo LRU policies. There are several variants of pseudo LRU policies and so on. So, finally, we are going to discuss the things that related to dealing with the writes in the cache. So, when we want to write a particular word to the cache, we can either write the word of data to a one level of cache or to the multiple levels of the cache in the hierarchy. So, effectively we have two options. If you are writing only to one particular the cache then that is called as a write back cache. And if you are writing to all the levels of the cache hierarchy then that is called as the write through policy.

(Refer Slide Time: 35:29)



Cache Block Updates: Write-Through Caches

- Requires more bandwidth at lowest levels of caches
- Keeps the cache consistent with lower levels of the memory

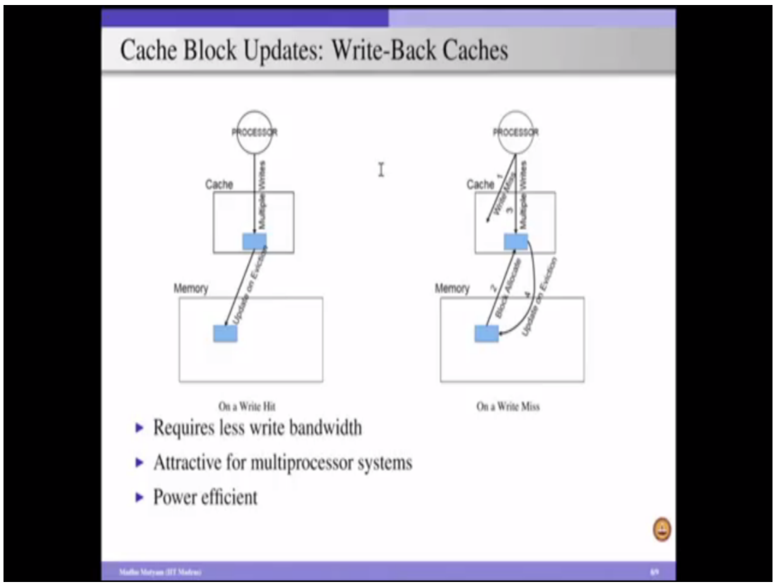
Madhu Mutyam (IIT Madras) 7/9

So, effectively we can design a cache by considering a write through policy or write back policy. So, when we consider a write through policy, processor generates for example, a write request to a particular block, it will be written to the cache as well as it will written to the lower level of the caches or the memory. In this particular example, we consider two level memory hierarchy where L1 cache and the memory is considered. So, whenever we write to a block in the cache we also write it to the memory. What happens if there is a miss? On a write miss we have two options, one is we can either write the data directly to the memory or we allocate the block in the cache from the memory and then write the data to the cache as well as to the memory.

So, if I consider the first policy that is the block is not allocated on a cache miss and writing directly to the memory which is called as a write no allocate. The write allocate policy, on a cache miss, first the data is written to the block in the memory and after writing to the block the entire block is brought into cache. If there are no reads to the recently written block then we can always go for write no allocate policy, but if there are multiple reads in the future for a recently written block, then we have to go for write allocate policy.

So, whether write through policy is efficient or not? Remember for every write you are writing to the lower levels of the memory or the cache which is effectively takes significant band width requirement. And that is not actually good especially when we consider a multiple core scenario, where multiple cores when they are trying to write to their caches they are also trying to the write to the lower of caches and there will be a significant bandwidth requirement. And also writing to all the levels of the caches incurs significant energy. So, from the energy and the bandwidth point of view write through caches are not efficient, but the main advantage with the write through caches is it keeps the lower level caches coherent with respect to the higher level caches. So, cache coherence maintenance as simple.

(Refer Slide Time: 38:08)



Cache Block Updates: Write-Back Caches

- Requires less write bandwidth
- Attractive for multiprocessor systems
- Power efficient

The other case is write back. In this whenever processor is generating a write request and if the write request is a hit in the cache, it will be written to the block and without writing this data to the memory. Effectively a cache block observes all the write request from the processor and this block will be written to the memory only when it is evicted. Since, cache block is updated with the processor writes. So, it is inconsistent with the memory block. To say that the cache block is updated or inconsistent with respect to the memory we maintain a one bit of information with the cache block that bit is called as a dirty bit.

And this dirty bit is set whenever processor writes to that particular block and when the cache replacement policy selects a block as a victim block and if the block is a dirty automatically we have to write this dirty block to the memory. On a miss when a processor generates a request which is a miss in the cache. So, when there is a miss, we have to go to the memory and get the block of data and allocate the block or store the block in the cache and after that we perform our write operations to the cache block. This is similar to write allocate policy of the write through cache that we discussed previously. So, obviously because the write back policy observes all processor writes.

So, it minimizes the bandwidth requirement and also it minimizes the overall energy consumption by not accessing the memory or the level of caches. And it is very attractive for multi-processor systems, mainly because the bandwidth requirement is not so high. So, typically we can consider write through policies mostly at the higher level caches, but for the lower level caches we go for write back caches. So, with this I am concluding this module.

Thank you.