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

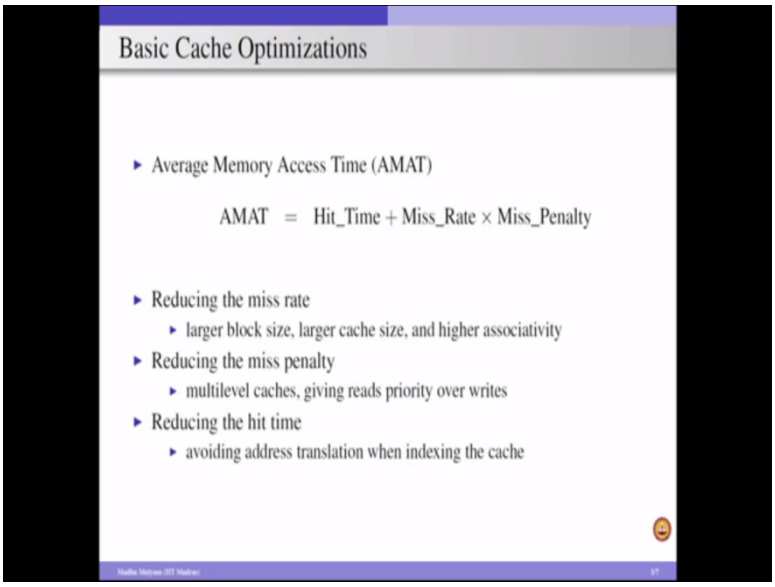
Module – 03
Lecture - 08
Memory Hierarchy Design (Part 3)

So, in the last module we discussed the basics of cache memory. And now in this module we are going to look at basic cache optimizations. So, we know that once we have cache memory the overall performance can be improved. In other words, the cache memory can reduce the average memory access time. So, if you look at the formula to calculate the average memory access time, it can be given as,

$$Average\ memory\ access\ time = Hit\ time + Miss\ rate * Miss\ Penalty$$

So, the hit time is the time it takes to service a request, if the request is hit in the cache. And if the request is a miss, then we are going to incur a miss penalty because we need to get the data from the main memory or the lower level caches. And the miss rate is defined as the total number of misses incurred per the overall memory access. So, given this average memory access time formula, now to optimize the caches, in other words to improve the AMAT, we have to either reduce the hit time or reduce the miss rate or reduce the miss penalty. So, we look at the techniques which will reduce each of these components, reducing the miss rate.

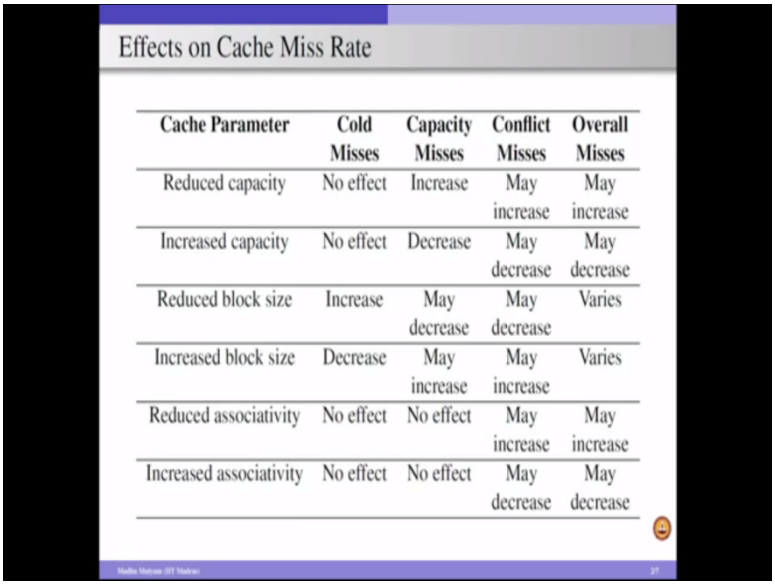
(Refer Slide Time: 01:42)



So, cache miss rate can be reduced by increasing the block size or making the cache bigger or increasing the associativity. And we will see what happens to different types of cache misses when we increase the block size associativity or the cache size. We already discussed in the previous module the cache miss classification. The caches can have compulsory misses, capacity misses and the conflict misses. The second one is to reduce the miss penalty. So, in order to reduce the miss penalty, we can go for multi-level cache hierarchy. Rather than one level of cache maybe we can go for two levels or three levels of cache memory between processor and the main memory.

And we will see, what is the impact of this on the overall miss penalty, or if we give priority for reads we can improve the overall performance of the processor and we will see how to implement such technique and finally, to reduce the hit time. So, we avoid address translation. Especially, in the case of systems with virtual memory support, processor generates virtual address and that virtual address needs to be translated to physical address and this translation is going to take time. And how can we eliminate this address translation from the overall access time so, that we will discuss at the end. So, we start with the techniques to reduce the miss rate. If I reduce the capacity so, the size of the cache is reduces then what happens?

(Refer Slide Time: 03:36)



So, we know that there are three types of misses the cold or compulsory misses, capacity misses and conflict misses. We already discussed in the previous module that the cold misses

are not dependent on the size of the cache. Even when you have 16 kb cache versus 128 kb cache your cold misses are the same in both the cases, but whereas, in the case of capacity misses, as I reduce the size of the cache automatically the capacity miss rate is going to increase because you have smaller size of the cache.

So, it keeps only limited amount of data in the cache. So, as a result applications may have increased capacity misses. In the case of conflict misses may increase the miss rate because of the reduced capacity. So, let us assume that the block size is constant in two caches one is with 64kb and the other one is 128kb when I go from 128kb cache to a 64kb cache as the block size is same.

So, we are going to have reduced number of cache blocks and because of that the application, if it is accessing random data which is located at different locations in the memory. So, there is a chance of increasing the conflict miss rate. So, in overall if I reduce the size of the cache the overall miss rate may increase. If I consider the opposite to this which is like increasing the capacity of the cache again there is no impact on the compulsory misses, but the capacity misses as expected is going to reduce.

In the case of conflict misses as I increase the cache size the number of blocks the cache can accommodate is going to increase and which in turn may reduce the overall conflict miss rate. Effectively the overall miss rate may decrease if I increase the size of the cache. So, this is about increasing or decreasing the size of the cache. Now we will look at the block size. If we reduce the block size for example, if I consider 64 byte block versus a 32 block byte. So, in the case of 64 byte block, I am going to bring for example, 16 words of data from the memory where each word is of 4 bytes. In the case of 32 bytes I am going to bring only 8 words. So, we know according to the definition of compulsory misses a miss that occurs when first reference to a word and the cache is empty.

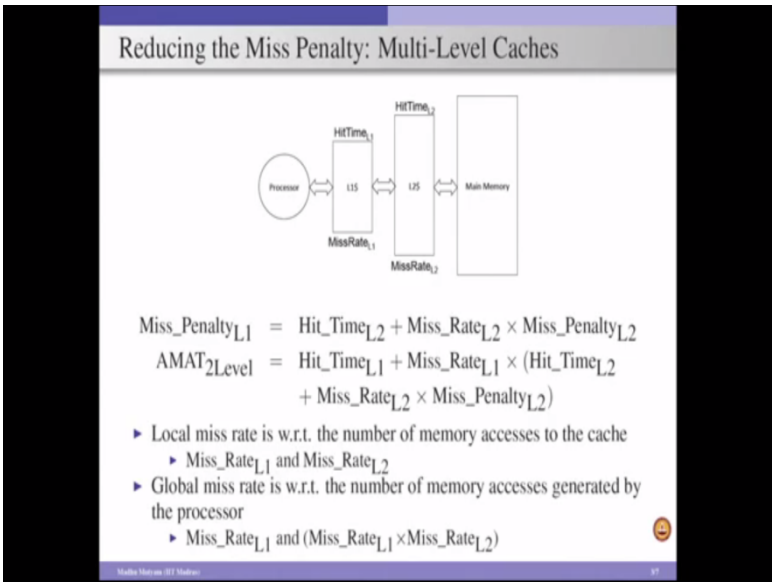
So, when I bring in the data, so I am servicing the first compulsory miss, but after that if I have a 64 byte block, I am going to bring another 15 words along with this missed word. Whereas, if I consider a 32 byte block I am going to bring only 7 words along with this missed word. So, as a result if I reduce the block size there is a chance of increasing the cold miss rate. And in the case of capacity misses, if I keep the size of the cache same, but the block size is reduced. So, as a result the number of blocks stored in the cache is going to increase. And as a result the overall capacity misses may decrease.

Similarly, in the case of conflict misses, because the number of blocks stored in the cache is increased by reducing the block size, the conflict miss rate also can decrease. So, effectively the overall misses varies. The same time, if I for example, increase the block size the compulsory misses are going to reduce, but in the case of capacity misses it may increase. Similarly, the conflict misses also may increase with the increased block size because as the block size increase the number of blocks stored in the cache is reduced. And if the application is exhibiting or accessing the data which blocks to different blocks in that scenario because the number of blocks stored in the cache is reduced now as a result these misses may increase.

Finally, if I alter the associativity of the cache, if I reduce the associativity there is no impact on the cold misses and there is no impact on the capacity misses because of the size of the cache is still same. And in the case of the conflict misses as the associativity is reduced the conflict miss rate may increase because the conflict miss rate is defined with respect to the misses happened with a set. So, as the associativity is reduced so we have less options to keep new block into the cache so we have to evict the existing data. So, as a result the overall misses may increase and when I increase the associativity the conflict misses may decrease, the capacity misses has no impact and similarly, the cold misses also there is no impact with respective to the associativity.

So, as a result like, so in order to reduce the cache miss rate if we just opt for one of these things either reducing the size or increasing the size of the cache, reducing or increasing the block size or changing the associativity there is an impact on the other type of cache misses when we are trying to reduce the miss rate of one type of a misses. So, we have to look at all these things and based on that we have to take a decision. But again among all these three types of misses the cold misses contribute very least number of misses to the overall misses and conflict misses are predominant. So, we have to look at the conflict misses and accordingly we can take the decision.

(Refer Slide Time: 10:50)



To reduce the miss penalty, we can go for multiple levels of cache hierarchy. So, we keep multiple caches between our first level of cache and the memory. So, that whenever there is a miss in the first level of cache rather than going to memory we may get a hit in the next level of the cache. In this particular example we consider two levels of caches followed by main memory. So, we know that,

$$Miss\_Penalty_{L1} = Hit\_time_{L2} + Miss\_Rate_{L2} * Miss\_Penalty$$

So, when there is a miss in the L1 cache now we are going to get the data from L2 as long as L2 supplies the data, if L2 cannot supply the data, we have to go to the main memory.

$$Miss\_Penalty_{L1} = Hit\_time_{L2} + Miss\_Rate_{L2} * Miss\_Penalty_{L2}$$

So, now we substitute this miss penalty of L1 in our original AMAT equation. So, AMAT for two level cache is,

$$AMAT_{2\,Level} = Hit\_time_{L1} + Miss\_Rate_{L1} * Miss\_penalty_{L1}$$

$$AMAT_{2\,Level} = Hit\_Time_{L1} + Miss\_Rate_{L1} * \left(Hit\_Time_{L2} + Miss\_Rate_{L2} * Miss\_Penalty_{L2} \right)$$

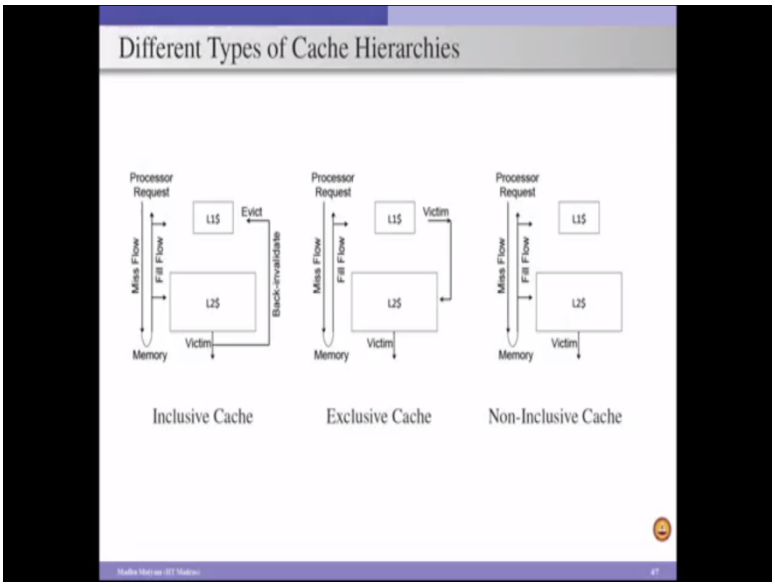
So, remember that here we consider the miss rate of L1, miss rate of L2, but we know that the definition of miss rate is the total number of misses incurred by a particular cache to the total number of memory requests coming to that particular cache.

So, as a result for the first level of cache that is L1 cache, all the memory request issued by the processor will come to L1. So, in that case our miss rate of L1 is the actual miss rate seen by the processor at the level 1 cache. The total number of memory accesses seen by L2 is nothing but the total number of misses generated by L1. So, effectively,

$$Local\_Miss\_Rate_{L2} = \frac{Total\ number\ of\ misses\ incurred\ by\ L2}{Total\ number\ of\ misses\ generated\ by\ L1}$$

So, we have to define the other term which is a global miss rate seen by the processor and in the case of level one cache our local miss rate or global miss rate is same because is the first level of cache where all the memory request generated by the processor will be seen by the L1 cache, but in the case of L2 the number of memory requests seen by L2 is nothing but the number of memory requests misses happen in the L1 cache. It is not the number of memory requests issued by the processor. So, effectively the global miss rate of L2 is nothing but the miss rate of L1 times the miss rate of L2. Now once you have multiple levels of caches. So, how do we organize these caches? So, a straight forward way is considering inclusive caches. In the case of inclusive cache the data that is there in L1 must be there in level two cache that is L2 cache.

(Refer Slide Time: 14:24)



So that means L1 is inclusive of L2. And in this case, whenever, if I evict a block from L2 cache, may be because of a cache replacement policy applied at L2 cache, so to keep the inclusive property correct. So, we have to invalidate the corresponding block from the L1

cache that is shown in the figure by using back invalidation. So, we have to invalidate a block in L1, if the same block is evicted from the L2 cache. Whenever there is a miss happens for a processor memory request in the L1 cache so we go to L2 and if even L2 also incurs a miss for the same memory request, then we will go to the memory. And when we get the data from memory we keep the block in L2 and supply the block to L1 and finally, supply the data to the processor.

So, that is what is shown in the figure using a miss flow. So, when we are filling the data from memory we first fill in L2 and then fill in L1 and supply the required word to the processor. This is simple to implement, but the disadvantage with the inclusive cache is your overall capacity is equal to the capacity of L2 cache only because the L1 is inclusive of L2. To increase the cache capacity we can go for other extreme which is called as exclusive cache where the data stored in L1 is completely exclusive to the data in L2, effectively L1, L2 are mutually exclusive.

So, when there is a miss in the cache in the L1 cache, we will go to the L2 and if there is a miss in L2 also we get the data from the memory, but when we are filling the data from memory we directly go to L1 and keep the block in the L1 cache alone. And whenever we evict a block from L1 we keep the block in the L2. The reason is when we evict a block from L1 we do not know whether processor may require the data in future or not. So, for the safe side we will keep all the evicted blocks from the L1 into L2. So, that if processor requests the data from evicted block in future L2 may supply the data. So, that way we can reduce the overall miss penalty.

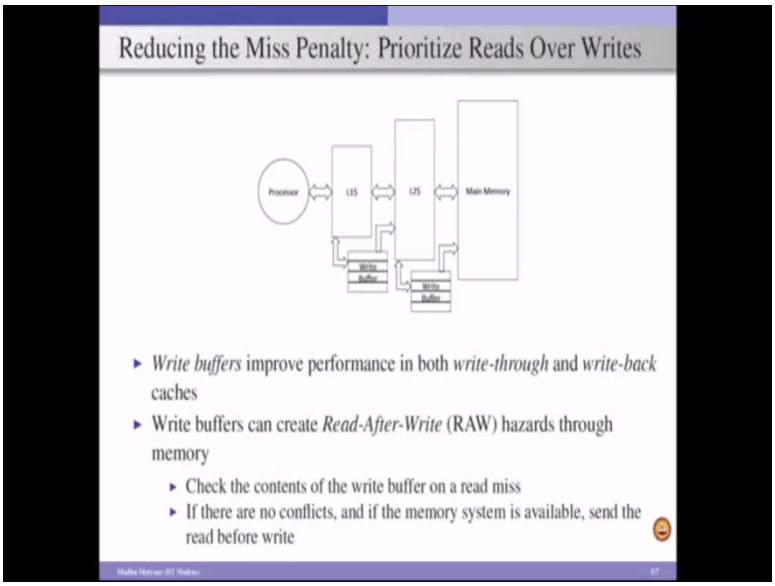
So, this improves the overall cache capacity, but the problem with this exclusive cache is whenever we want to implement cache coherency. So, do not worry about the definition of cache coherency we are going to discuss the cache coherency when we come to the multicore architecture in the last unit of this course. When we want to invalidate a cache block we have to search in both L1 and L2 caches if you are considering the cache hierarchy in an exclusive fashion. But whereas, in the case of inclusive caches because we know that all the L1 data is there in L2, all we have to do is just search in L2 cache and invalidate to locate a block we have to search both in L1 and L2 as L1, L2 are mutually exclusive in the case of exclusive caches.

So, as a result, from the implementation point of view the exclusive caches are costly, but from the capacity point of view the exclusive caches are efficient. In other words the capacity of the exclusive caches is equal to the sum of the capacity of L1 cache and L2 cache, but the current day processors are actually implementing an intermediate to these two extremes which is called as non-inclusive caches. So, this non-inclusive cache says that data of L1 may or may not present in L2.

So, whenever processor incurs miss in the L1 cache, we go to the L2 cache, if L2 is also missing then we go to the memory but when we are bringing the data from memory we keep the block both in L2 and L1, but now when L2 cache evicts a particular block we do not have to back invalidate the corresponding block in L1 we can just invalidate in L2 itself. So, as a result it violates inclusive property. But the advantage with these non-inclusive caches is it can increase the capacity overall capacity of the cache hierarchy as compared to the inclusive caches, but the drawback with this is still when we want to implement the cache coherency to invalidate a particular block we have to search both L2 and L1. Or otherwise we have to come up with mechanisms such as like extra bit is allocated for each of the block in the L2.

And this extra bit may say whether the block is present in the L1 or L2. We can do couple of the optimizations to reduce the penalty associated with searching in both levels of caches especially in the case of cache coherency implementations. So, now we will look at the techniques to minimize the miss penalty. So, we know that the processor can generate a read request or a write request. And generally the reads are performance critical because processor will wait for the data to be available to continue further, but whereas, in the case of a write operation processor can write and it can continue with subsequent operations. So, as a result we need to give priority for reads over writes.

(Refer Slide Time: 20:43)



But in the normal design of a system with multiple levels of cache hierarchy, when a processor generates a write request and the write will be written to the L1 cache. And when we are evicting a block from the L1 cache, for example and the block is dirty. Now, what is going to happen is we have to write it to L2 to ensure that this write operation is completed. Especially in the case of write back caches whenever we evict a block from L1, if the block is dirty, to maintain the consistency we have to write the data to the lower level of caches.

In this particular example, we have to write this dirty block from L1 to L2. Unless we perform that write operation to L2, we cannot proceed further in the normal cases, this is with respect to write back caches, if we consider the cache hierarchy is designed in such a way that write through cache is implemented. Then whenever a processor is writing a word of data to L1 cache according to the write through mechanism we have to write it to the lower level of cache hierarchy also. That means whenever we are performing a single write operation we have to write to L1, L2 and to the memory in this particular example. But writing to L2 is going to take more time. Similarly, writing to memory is going to take much more time. So, as a result if you are implementing the write through policy in the normal cases, then every write is going to take significant amount of time.

And as the write is not performed processor is not going to continue with the subsequent requests and that is going to degrade the performance significantly. So, what we can do is we can provide write buffers associated with each of the caches. And whenever we are evicting a

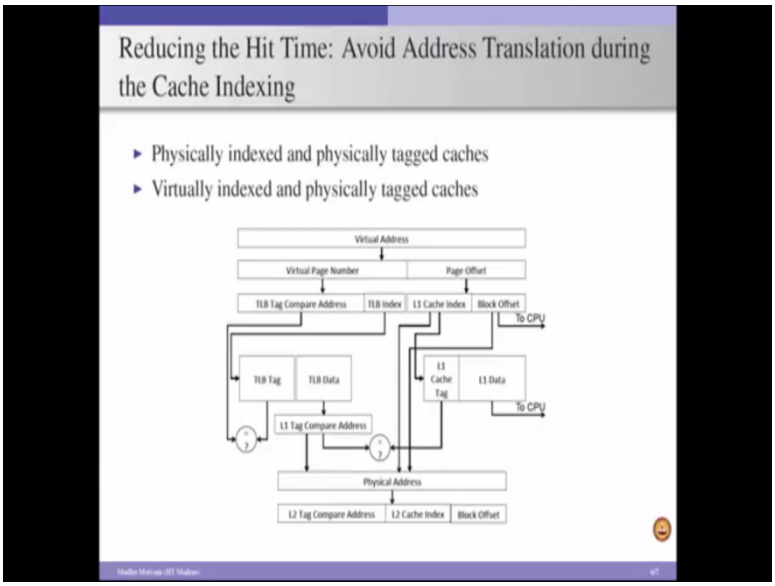
block, a dirty block from L1 we can write the dirty block to the write buffer. Once we write the dirty block to the write buffer then processor can resume execution of subsequent requests performing the other operations and so on and also cache is free with performing the other operations.

Similarly, in the case of write through caches where a processor is writing a word to L1 cache, the same word will be written to the write buffer and the processor will not wait for writing to the L2 and the memory. Because as long as we write to the write buffer, from the processor point of view that write operation is said to be completed. So, that processor can execute subsequent request to this write request. This way we can improve the overall performance. So, effectively write buffers improve performance both in the case of write through as well as write back caches, but write buffers can create problems especially read after write hazard.

If processor generates a read request to a block which is recently modified and evicted from L1 and this modified dirty data is stored in the write buffer between L1 and L2 and there is a read request to this evicted block. So, this read request will incur a miss in the L1 cache and if you are not going to look at the write buffer which is associated with the L1 cache and directly go to the L2 cache, L2 cache may supply the data, but the data is stale. So, this is called as ‘Read after Write’ hazard. In order to overcome this problem whenever there is a read request generated by the processor and which incurs a miss in the L1 cache, we have to search in the write buffer with that particular address.

Check all the contents of the write buffer on a read miss in the L1. Similarly, if there is a read miss happens at the L2 we have to search the write buffer contents associated with the L2 cache and so on. And if there is no conflict in the write buffer we can supply this read miss request to the lower level caches. So, that the lower level caches can supply the correct data, but if there is a match with the write buffer contents then the write buffer supplies the data to the processor for that particular read request.

(Refer Slide Time: 25:26)



And finally, we look at a technique to reduce the hit time. So, when we have a system supporting virtual memory processor actually generates virtual address for a load or a store. And this virtual address needs to be converted into a physical address and because the physical address is the actual location, the actual address specified in the memory, so we have to convert virtual address into physical address and to do that efficiently in the hardware we are going to have Translation Look Aside Buffers TLBs.

We search in the TLB and translate virtual address into physical address and once we have the physical address we partition this physical address into three segments. One is a block offset, the second one is set index and the third one is the tag associated with that particular block especially in the case of set associative caches. And using these three fits we search in the cache and finally, we supply the data if the cache has that particular data, but doing all these TLB look up, is actually is going to increase your overall hit time.

Even if the cache is supplying the data we have to translate that before actually accessing the cache. So, effectively these TLB look up is in the critical path to supply the data from the cache to the processor. So, how do we reduce this or how do we eliminate this TLB look up. So, in the normal cases typically cache is designed as physically indexed and physically tagged, that is only after address is translated from virtual address to the physical address we actually search in the cache, first by looking at the set index, index into the cache and then compare with the physical tag.

So, the caches which are designed using this principle are called as PIPT caches, physically indexed and physically tagged caches. But because this has a problem of increased hit time, in order to reduce this we can go for virtual caches, where tag look up and the indexing into the cache can be done through the virtual address, but if we do completely for the virtual address, there are some issues such as protection as well as aliasing problems and so on.

So, that is the reason why caches are not completely designed with virtual address search. So, we can consider an intermediate technique which is called as virtually indexed and physically tagged. Our indexing into the cache can happen with the virtual address, but tagging, tag comparison can happen with the physical tag. So, we will just give an example, where how this virtually indexed and physically tagging happens in a cache, which is designed as VIPT, virtually indexed and physically tagged caches. Given a virtual address generated by the processor, we partition that into two fields, one is the page offset and the other one is virtual page number.

From the page offset we further divide that. So, one component is going to give you block offset, the second component is going to give you the L1 cache index. Effectively this is the set index into this L1 cache and virtual page number is divided into TLB index and TLB tag compare. And our TLB actually contains the virtual tag as well as the physical tag. So, when we index both into the tag array and the data array of the TLB, we read the TLB tag and this TLB tag is compared with the tag that we obtain from the virtual address that is a TLB tag compare address.

And if it is matching then we read the data from the data TLB portion and that actually gives the physical tag. Remember while we are performing all these operations on TLB we can also perform operations on the actual L1 cache by indexing into the L1 cache using the L1 cache index and we read the L1 data. So, once we perform these two operations simultaneously, one is performing the operations on the TLB cache, the other one is performing the operations on the L1 cache.

So, we get the physical tag from the TLB data array similarly, the physical tag from the L1 tag array. And these two will be compared and if there is a match that indicates that the data is available for the virtual address generated by the processor and we supply the data from the L1 data array to the processor. If there is a miss then we have to go to the next level cache and before going to the next level cache we have to come up with the physical address. So,

we will get the physical address by combining this L1 tag address that is obtained from the TLB data portion.

And we combine that with the L1 cache index and the block offset obtained from the page offset of the virtual address generated by the processor. And once we have this physical address we further partition this into three component, one is L2 tag compare address and L2 cache index and the block offset. And once we have this we will go the next level cache and perform the operations. Remember that this virtual indexing can happen only for the L1 caches. We are not going to perform this for the L2 and L3 caches and so on. And effectively in a multi-level cache hierarchy L2 and the further levels of caches are designed using physical index and physical tag, but whereas, L1 cache can be designed using virtually indexed and physically tagged.

As we eliminated the tag lookup, TLB lookup from the critical path of the cache access so that we can improve the overall performance. So, with these basic optimization techniques I am going to conclude this module and in the next module I am going to discuss the advanced optimization techniques that can be applied on cache memory.

Thank you.