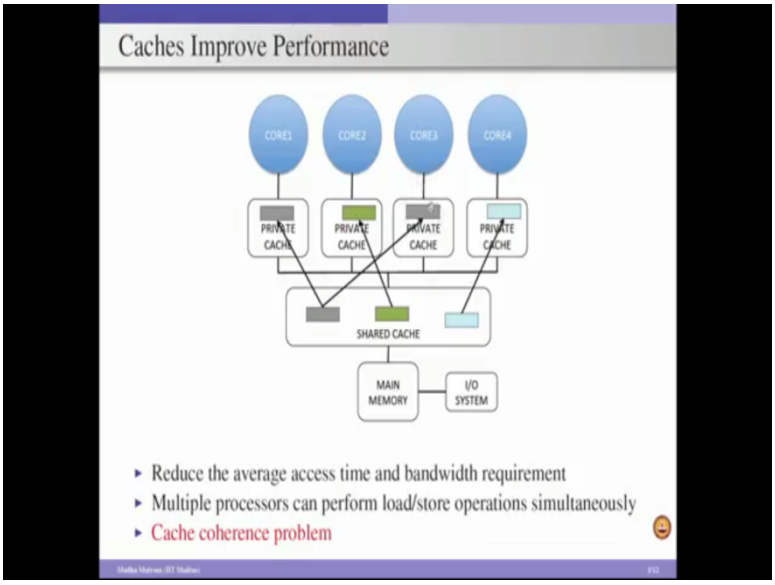
Computer Architecture
Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module – 08
Lecture – 29
Cache Coherence

So, in the last module, we discussed the need for multicore systems and we explained why, we have to go from single core system to multi core systems. And now in this module, we are going to discuss one of the main issues associated with this multicore system that is the cache coherency problem. And we first discuss what is a cache coherency problem and then we will discuss how to deal with this cache coherence problem associated with this multicore processors.

(Refer Slide Time: 00:45)



So, in the multi cores systems, we have multiple cores and each core will have one or two levels of cache hierarchy, as a private cache and then there will be a shared cache and this shared cache will be shared by all the cores associated with that multi core system. And if we see the overall design, we can clearly see here we have collection of cores, and each core has set of private caches and then there is a shared cache there is a main memory and the I/O system.

In this particular example we consider two levels of cache hierarchy, the level one cache which is private to each of the cores. And there is a level two cache which is shared across all

these cores. Now, we know that actually the cache memory is going to improve the overall performance by reducing the number of times, we are going to the memory because whenever we require the data.

And if the required data is going to be placed in the cache memory, which is closer to the processor as a result we can minimize the number of times going to the memory and that will improve the overall performance. And also most of the applications exhibit spatial and temporal locality. So, as a result we exploit this spatial and temporal locality available in the applications by using this cache memory that we already discussed as part of our memory hierarchy design in previous modules.

Now, given this multi core system with the two level cache hierarchy where first level of cache hierarchy is private for all the cores, and second level cache hierarchy is shared by all the cores. Given this scenario, now let us say core one wants some data, and it goes to this shared cache and get this block of data and put it in its private cache, and after that there may be a request from core two and it will go to shared cache get the data and put it here.

And simultaneously there may be a request from core four and if shared cache can support multiple request processing simultaneously. So, this request can also be serviced along with servicing this request, and after some time core three may request some data which is the same data, whatever is requested by core one earlier. So, this block is placed in this private cache, as well as this private cache. So, effectively once we have this multi core system, some data may be replicated in multiple locations in different caches.

Once we have this cache memory in our system, we can reduce the overall access time because rather than going to the memory always, now we can service a request from our the cache hierarchy itself, and that is going to reduce the access time. And also it is going to reduce the bandwidth requirement at the memory because now most of the request will be serviced by this caches. So, as a result the memory bandwidth requirement, at this level will be reduced.

And also once we have this cache hierarchy with the private caches and shared caches, so now what happens is multiple cores can perform load store operation simultaneously. So for example, if this core can send a request and that request can be serviced by this cache so it is not actually coming to the shared cache, and it can service the request here itself. At the same

time if this core has some load or store request, and the data is available here then it can also perform simultaneously.

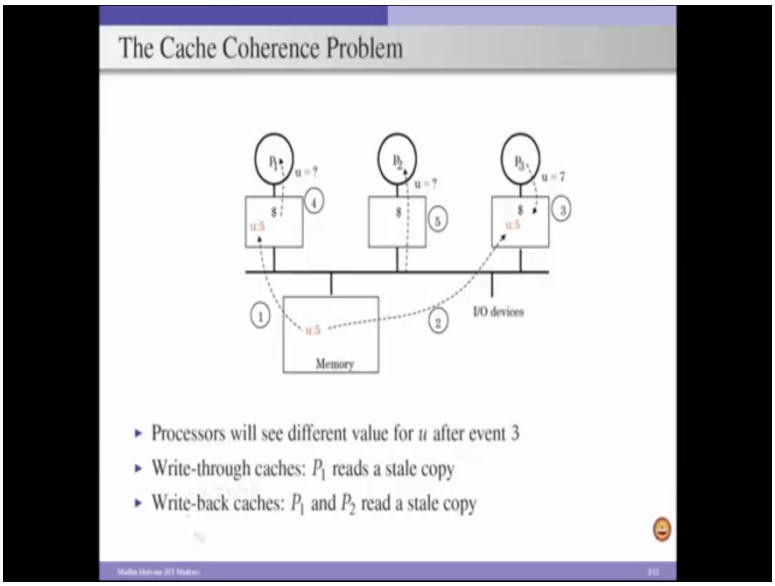
So, in other words once we have this multi core system where multiple cores can perform load and store operations on their local caches simultaneously. And also sometimes these requests can go to the shared cache, but again if the shared cache has support for multiple ports, then this shared cache also can service multiple requests simultaneously. As long as these requests are not conflicting with the same address and so on so, far so good.

So, having multiple levels of cache hierarchy we minimize the number of times going to the memory that is going to improve, the overall performance, also multiple cores can perform load and store operation simultaneously, that also results in increased performance, everything is good. But having multiple private caches in multi core system, where we are running a multi threaded application where multiple threads of this application are executed on multiple cores, and it may create cache coherence problem.

As I mentioned earlier because this data is requested by core one and core three, so a result this block is placed in the private cache of core one, as well the private cache of core three. And now a thread executing on core one, if it updates this data and after that the thread running on this core three, if it requires updated data, this updation is not visible here, as a result this thread is going to read a stale copy and that is going to give wrong computation. And this is called as a cache coherence problem.

So, we have to deal with this cache coherence problem, if we want to execute a multi threaded application on our multi core system. So, now we are going to see in the rest of this module, how we are going to deal with this cache coherence problem, and before that we actually illustrate this cache coherence problem with an example.

(Refer Slide Time: 06:48)



Consider a three core system, where each core has a private cache, and we have a memory. And this is connected to this common bus. So, this common bus is connecting all the caches of this cores with the memory, and this memory has some data 5 in an address location u. Now, let us say core one issues a load request to read the data that is loaded at address u in the memory. So, it sends a load request here because once it sends a load request data is not there in the private cache associated with this core. So, this request is going to the memory, and memory is supplying the data.

So, effectively this data 5 is placed in the cache associated with this processor p1 or core p1. After sometime core three also requested the same data, so again 5 is brought into the cache of core three by using the load request. Now, core three wants to update this value 5 to 7. It issues a write request or a store request, to store 7 in the address location u. So, it performs this so as result, now we are going to have 7 here, after that if core one wants to read this data again, and now what is the value this core is going to get? Whether it is going to get value 5 or value 7? If this write operation is not communicated to all the cores, then when it issues a load request to read the value from address location u, this load is a hit in the cache. So, as a result this cache is going to supply 5 as the value loaded in the address location u. So, as a result p1 is going to read the wrong data.

Now, if core two wants to read the data from the address location u because core two's cache is empty initially. So, this load request is a miss in the cache, so this request will be sent to

the memory and now the memory also has a value 5 in address location u. So, as a result it is going to supply value 5 to this core because if this write request is not reflected at the memory, then still this address location u has value 5 only. So, as a result when a request from core two comes to the memory, this memory is going to supply on the stale data. So, as a result this core one and core two are going to get wrong data and which is going to give a wrong computation at the end. And this is what is called as the cache coherence problem.

So, in order to overcome this problem we have to come up with the set of techniques called as protocols and we are going to discuss one protocol in our design, but before that we know that our cache can be designed as a write through cache, or write back cache. So, in the write through cache whatever we update in one level of cache, we have to update that in the next level also that is what is called as a write through. When we are writing something to a cache location, then we have to update that in the lower levels caches also.

Whereas in the case of write back, we can accumulate all writes in the first level of cache, and only when we are evicting a block from the first level cache and if the block is dirty and if the dirty block is going to be evicted then we are going to write it to the next level, that will be the case only when we consider the write back cache. In other words in the write though cache our writes to the first level cache, will be reflected in the second level cache also. But where as in the write back cache writes to the first level cache will be reflected only when the block is evicted from the first level cache. So that this will be returned to the next level cache.

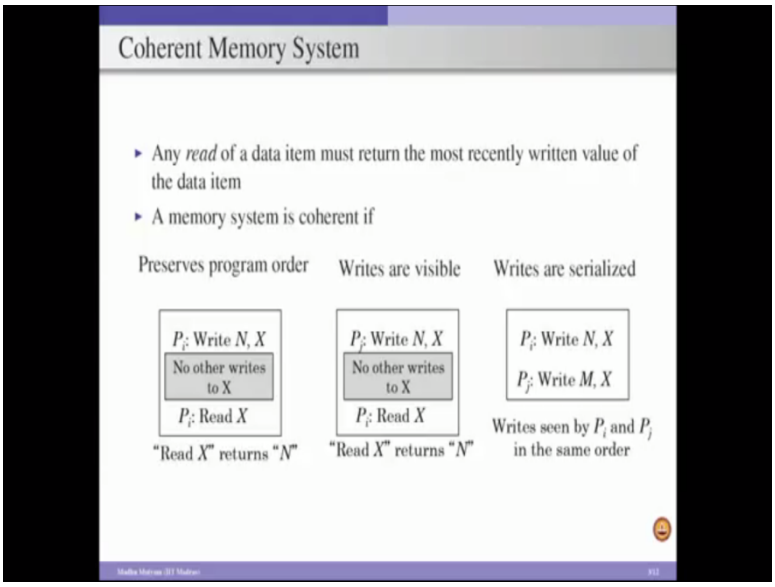
So, even when we consider write through or write back in any of these cases also we still have this cache coherence problem. So for example, we consider write through cache, so when we consider write through cache, so all these caches are write through caches, so whatever writes we perform to any of these caches that write will be reflected in the next level that is the memory. So that means when p3 is writing value 7 to this address location u in this cache, this write also will be updated in the memory.

So, effectively now this u is going to have the value 7 after we perform this write operation because this is a write through. So, whatever you write here it'll be written to the next level that is the memory. Now, what happens when core one wants to read the value from the address location u, because this write through is not actually sending this data back to this cache and once we are not sending this data to this, so previous data will be there as it is, so as a result when core one wants to read the value from address location u, it is already having

value 5. So it is going to supply value 5. So, as a result still we have the cache coherence problem even when, we consider write through type of caches.

Now, consider write back cache when we are considering a write back cache. So, automatically this write will not be reflected in the memory, and because of that when core two wants to read the data from address location u, it will send the request to the memory and the memory is going to supply the stale value. So that means core two is going to get a stale data, when we consider write back caches here and core one is going to get the stale data when we consider the write through cache. That means irrespective of whether you consider write through cache, or write back cache the cache coherence problem is there. So, in order to deal with the cache coherence problems, we have to come up with set of protocols those are called as a cache coherence protocols.

(Refer Slide Time: 13:36)



In order to have a coherent memory system, we have to preserve the program order in our system. When I say preserving the program order, for example, we have n core system, where we have n cores in our multi core system and we are dealing with the i th core. And this i th core is issuing a write request, write value n to an address location x and after sometime, the same is core is issuing a load request. So, load from the address location x.

Now, this load request has to return the value n because between the store request and the load request there are no other writes happened to the address location x, in such scenario this load request is going to supply the value written by this write request, or a store request. So

that means, when we have a system which is executing a set of instructions, then we have to maintain the program order. Especially, when we are dealing with loads and stores, only when we maintain this program order then we can clearly say that this load request is going to supply the value returned by this store request, provided there are no other intermediate store request to that address location x.

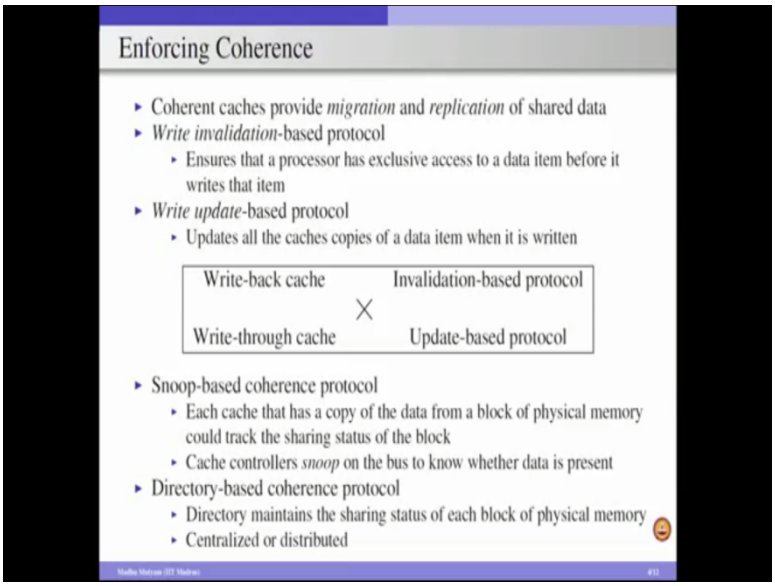
Second writes are visible, so this is again when we have an n core system. There is a write request from core j writing a value n to an address location x, and after that core i wants to read the value from this address location x. So, effectively this core i wants to read from some location, which is previously written by core j with a store request. And also we assume that there are no other writes happened to this x, whether from the same processor or from different processor, no other requests happened. In such scenario, so this load request has to return the value n, if it is so then we can say that writes are visible because this write is happening in core j, but this write is visible to core i. If it is visible then only this core i is going to return the same value, whatever is returned by this core j that is called as writes are visible.

The third one is writes are serialized, so here we can see core i is writing value n to address location x, and after sometime core j is also writing some other value that is m to the same address location x. If these two write operations are happening, then if both processors or both cores p_i and p_j if they see this writes in the same order, then we can say that these writes are serialized. If these two are seeing this order in a different way, then as a result the writes are not serialized, and then the memory system is said to be non coherent.

So, in other words we can say a memory system is coherent, if and only if that memory system satisfies these three properties. One is individual cores will maintain the program order, and writes are visible across the cores, and writes are said to be serialized across the cores. If these three properties are satisfied then we can say our memory system is coherent.

And once the memory system is coherent, when we want to perform a load operation then we will get the latest return value from that location. So, as a result our computation will be correct. So, once these three properties are satisfied, now we can clearly see here any read of a data item from any core will return their recently written value to that particular location. So, the goal of any cache coherence protocol is to maintain this coherence across the memory system.

(Refer Slide Time: 18:45)



So to enforce the coherence, we have to come up with the set of protocols. And so once we have a coherent cache, then we get the correct data when we want to perform a load operation or a store operation. And coherent caches provide the migration and the replication of shared data. When I say replication, the same data can be replicated in multiple private caches, but even when we maintain the replicated copies when we update one copy of this replicated copies, in one particular cache because we have a support of cache coherence, so automatically the other copies will be invalidated or updated, whenever we perform an update on one replicated copy.

Also this coherent caches are providing the migration. Migration indicates that, let us say block will be there in one cache associated with one core and if the other core wants the data then this whole block can be migrated from the previous cache to the cache associated with this requesting core. So, we can move the entire block of data from one cache to another cache, so that we still have only one copy of data. And this copy is now with the recently requested core.

So, we can move the data from one cache to the other cache, or we can replicate the data in multiple private caches associated with the multiple cores of the system. And once we have this migration and the replication, now we can consider write invalidation based protocol to maintain the cache coherence. When I say write invalidation, whenever I want to write something to a replicated copy, I will invalidate all other replicated copies associated with the

other caches in the system. So that we ensure that only one copy of data is available, with the core that is actually going to perform this write operation.

So, in other words the core which is going to perform the write operation is going to have an exclusive permission to use that particular block. So that it can perform whatever the write it wants to write it to that location. The other type of protocol will be write update. So, in the write update based protocol whenever we are actually going to write to a block of data, or a replicated block of data associated with a particular core, we are going to send this write request to all the other cores also. So that their caches will see if they have the data or not, if they have the data then they will update in that particular locations.

So, we have two types of protocols one is write invalidation based, the other one is write update based. And also we have two types of caches, one is write through cache, the other one is write back cache. So, once we have this two types of caches and two types of cache coherence protocols, our cache coherence protocol design space consists of four different types of cache coherence protocols. One is write back cache, with the invalidation protocol, or write back cache with update based protocol, or write through cache with invalidation based protocol, or write through cache with update based protocol.

So, we can come up with a cache coherence protocol which belongs to one of these four classes. And depending on the type of cache we have, so one of these two will be fixed, and once we have that then we will see whether we are going to go for invalidation based protocol or update based protocol. So, in the invalidation based protocol, we are actually sending just one invalidation signal, so that all the other caches which are having the replicated copies they can invalidate that data.

So, all you need is we will send the address of the block where we are going to perform the write operation, and this address will be sent to all the caches in our multicore system and the cache controller associated with each of these private caches, will look at this address and search in their caches to see if there is a match or not. If any cache controller finds a match in the associated cache, then that block will be invalidated when we are considering the invalidation based protocol.

So, as a result it is simple to send the invalidation signal and it is not going to take too many interconnection wires or the bus. So, the bus width of this invalidation based protocol will be simple, all we require is the address of the block needs to be sent. So that the replicated

copies can be invalidated, but the disadvantage with this invalidation based protocol is, when we invalidate the replicated copies in the other caches, and if any of the corresponding cores request the data next time, then they find a miss in their caches. So, they have to again go to the previously updated core and that core is going to supply the data and so on.

So, that means this invalidation protocol may sometimes increase the number of the cache misses because this invalidation based protocol is going to invalidate the blocks. And if these invalidated blocks are required by the corresponding cores, then they are going to have increase number of the cache miss rate. But on the other hand if you are considering a update based protocol for every write, we are actually writing this value to the other cores also. Even when the other cores are actually not requiring this block anymore.

So that means when we are performing this update based protocol, so we have to send not only the address, but also the data what whatever we are going to write it, through that store operation. So, we have to send both the things to all the cores in our system, and the cache controller will take the address and search for a match in the corresponding caches. And if there is a match then this data will be placed in the appropriate location in the particular block.

So, as a result all the caches which have this replicated copies, also will be updated based on this write operation. But if none of these cores requesting this data in the future, then this write operation is unnecessarily wasting the overall power consumption. So, as a result there is a trade off, in the invalidation based protocol, we are minimizing the wastage of this multiple write operations and associated energy. But we are incurring more number of the cache misses, but where as in the update based protocol we are unnecessarily increasing the amount of energy consumption because of this unnecessary writes. But our caches will be up to date so that if they request the data in the future then they will get hits.

Now, it is up to us to see whether we want to go for invalidation based protocol or update based protocol, based on our requirements. Whether, if you are not concerned more about the energy and so on then we can even go for the update based protocol. But another problem with the update based protocol is this unnecessarily increases the bandwidth requirement because you are actually consuming the bus bandwidth for updating, or write request in across all the replicated copies and so on.

So, as long as we are not worried about the bandwidth, we are not worried about the energy then we can go for update based protocol. But in reality the energy consumption is one of the critical things, so we have to minimize this unnecessary energy consumption. So, in order not to put pressure on the bandwidth as well as in order to minimize the energy consumption, it is always better to go for invalidation based protocol.

So, again when we have the number of cores in our multicore system is let us say 4 or 8. So, as long as the number of cores is reasonable number, or as long as in our multicore system we have reasonable number of cores, then we can consider a bus based system where all the cores are connected by a common bus. And they use this common bus to communicate across these cores.

And when we consider this bus based system then our cache coherence protocol, whatever we design are called as snoop based cache coherence protocols where, whenever we are performing a write operation or a read operation we send this request on the bus and all the cache controllers associated with all the caches will snoop this bus, and take that request and search in their caches to see if there is a match or not. And if there is a match then the corresponding cache controllers are going to take the appropriate action either by invalidating the copies or by updating their copies and so on.

The other type of design is the directory based cache coherence protocol, here when the number of cores in our multicore system is large, in terms of let us say a 16, 32 or 64 cores or even hundreds of cores then bus based connection is not a suitable mechanism for multicore system. Then we have to go for network on chip or interconnection network where multiple cores will be connected through a mesh type of topology, or something.

Once we have that type of design then we have to go for a different type of cache coherence protocol design that is called as directory based cache coherence protocol. But underlying protocol mechanism is same, but the way in which protocol is implemented is slightly different. When we consider a snoop based protocol to the directory based protocol. In the directory based protocol, we maintain the directories for each of these the caches, and shared cache is going to maintain the sharer's information. And this sharer information is going to say, which cache, which private cache is actually having a copy of the requested block.

We are going to maintain the sharer's information for each of the blocks in our shared cache. And this block in the shared cache can be replicated in multiple private caches associated

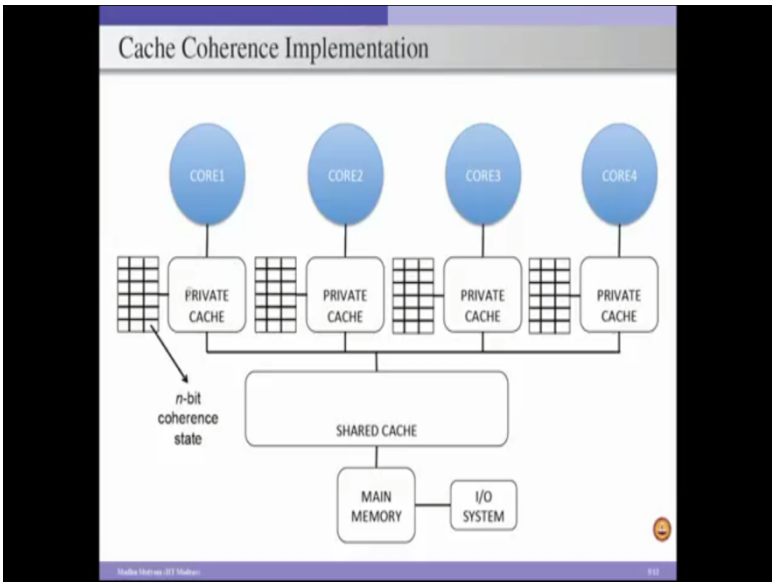
with the multiple cores, and using this directory we are going to know who are all sharing this block. And whenever we want to perform any write operation or whatever, then we know whom to send our invalidation signals and accordingly we just invalidate the copies.

And this directory based protocol can be implemented either by consider a centralized design, or a distributed design. So, when I say centralized design so we will have a single directory and we will have a single monolithic shared cache. So that whenever we are going to perform any operation, all the requests will come to this centralized directory, and they will get the responses from the centralized directory. But the problem with the centralized design is because everyone is coming to this centralized directory. So, this centralized directory needs to have multiple ports, and that is going to incur significant area and the power overheads.

So, in order to overcome that we can go for a distributed design, where our cache is distributed across multiple cores, and this cache is a shared cache so effectively shared cache is distributed across multiple cores. And each of the chunk of the shared distributed cache will have a corresponding directory and which maintains the direct, the sharer's information, for all the blocks that belong to that the chunk of shared cache. So that once we have this distributed directory, so each distributed directory will have a single port, and based on the address location we can send our request to the appropriate directory of the associated chunk of the shared cache. So, as a result we can eliminate the problem associated with centralized design.

In summary, we have invalidation based protocols or update based protocols. And based on the type of cache we can design invalidation based protocol for write through cache, or for write back cache. Or similarly, we can design an update based protocol for write back cache or write through cache. And also when we have more number of cores then we have to go for a directory based protocol design, or if our number of cores is 4 or 8 that is a reasonable number, then we can go for a snoop based cache coherence protocol.

(Refer Slide Time: 32:45)



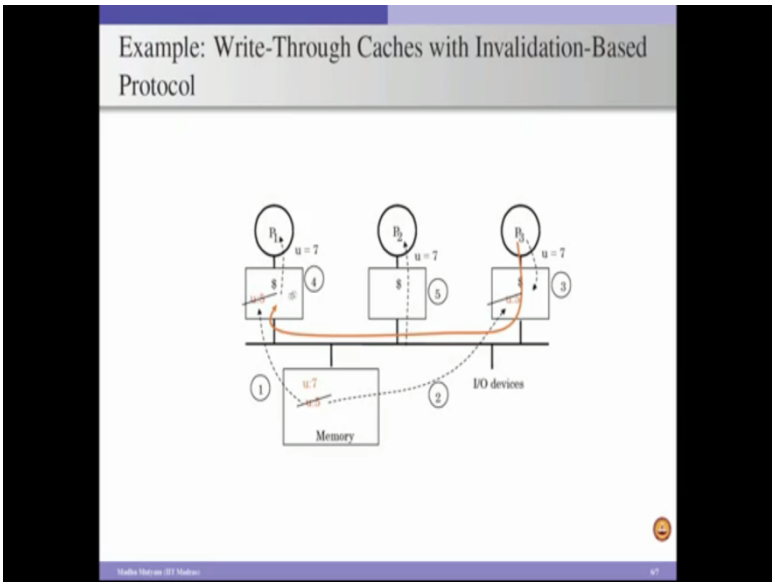
So, in order to implement this cache coherence protocol in our design, so we have to associate a state information for each of the blocks, that are present in our the private cache. So, let us say this private cache has for example, 1024 blocks then for each of the blocks, we are going to have some n-bit information, in this separate table. And this n-bit information is actually going to specify the state of the block. The state can be whether the block is invalid, whether the block is shared, whether the block is modified or whether the block is exclusive, or something.

Depending on the type of cache coherence protocol we design, so we are going to have different number of states. Let us say if you are considering a two state cache coherence protocol, then we are going to have two states associated with each cache block. And if you are going to consider a five state cache coherence protocol. For example, MOESI, so then we have to have one of the five state associated with each cache block, in our private cache. Though for the simplicity sake I am representing here this state information separately, but actually we can have this state information with the meta data of a cache. So, the meta data consists of the tags, other valid bits and so on along with that we can also have this cache coherence state information.

So, here in this particular design we are actually considering for each location in the cache, we are going to have n-bit information. So, our entire private cache is divided into, let us say k blocks, so each of this k blocks will have n-bit information. And this n-bit information is

going to specify the state in which the block that is present in that particular location, in the cache. Let us say if this is going to say that it is exclusive then the corresponding block that is stored in this location of this private cache, is in the exclusive state and so on. Now, we are going to see one the cache coherence protocol with an example.

(Refer Slide Time: 35:21)



So, here we consider write through caches and we implement invalidation based protocol. And we know that in write through caches if we write some data to a block then the data will be updated, in the next level cache or the memory in the hierarchy. Now we will consider the first transaction, so here the core one wants the data from address location u. So, it finds miss in the cache so the core one sends a request to the next level that is memory and memory supplies the data.

After some time core three sends a request for the same data, so it finds a miss here and it goes to the memory, and memory supplies the data here. And now after that let us say if core three wants to update the data that is stored in address location u. So because this is an invalidation based protocol, whenever core three wants to update some data then it has to send that request on the bus, indicating that. So this core is going to update, so as a result all other cores if they have the data, they have to invalidate that particular copy.

So, once it sends an invalidation signal on the bus, this bus is connected to all the caches of this multi core system, and also it is connected to the memory. So, all the controllers whether that is a cache controller or the memory controller, will snoop on the bus. And these cache

controllers or the memory controller will get this request and they search in their caches or the memory to see whether, there is a match for that particular the address. And if there is a match then these cache controllers are going to invalidate the corresponding blocks.

So, here in core two there is no data related to this address location u, as a result it is not going to do anything here, but in core one there is a match. So, automatically core one cache is going to invalidate corresponding block. Similarly, the memory controller also finds a match for this address location, so it invalidates the data. But now because core three is updating the value and we are implementing a write through cache. So, whatever the value we update here that will be reflected in the next level cache or memory in the hierarchy.

So, as a result as soon as core three writes value 7 to the address location u, so automatically it will be updated in the address location u in the memory also. So at the end of this third transaction, now core three is having the value 7 in the address location u. And similarly, memory is also having the value 7 in the address location u, but core one cache is now having a invalidated copy associated with this address location u.

Now, after this if core one wants to read the data from address location u, so in that case because it finds a miss for this address location u because previously it was invalidated. Now, core one sends the request to the next level memory and memory is going to supply the value 7 to it. So, as a result now core one is getting the correct data, so in other words whatever the write we have performed here in core three cache that is now actually reflected in core one's cache. So, its effectively we can say writes are visible.

Now, after sometime if core two wants to request the data, now in this case. So, core two finds a miss for this address location u because it has not requested this data earlier. So, it sends the request on the bus and then the memory is going to supply the data for that particular thing. And as a result core two also gets the value 7 in the address location u. And here in this particular design, we consider that it is the responsibility of the memory to supply the data as long as the data is not in a modified state in any of the caches.

But we can also consider a different type of design where, rather than memory supplying the data, we can even instruct one of the caches to supply the data, if those caches have the requested data. So, again so it can be like cache to cache transfer or memory to cache transfer so, but in this particular example we consider that the memory is going to supply the

requested data for any request from the processor, as long as the requested data is not there in any of the caches in modified state. So, with that I am concluding this module.

Thank you.