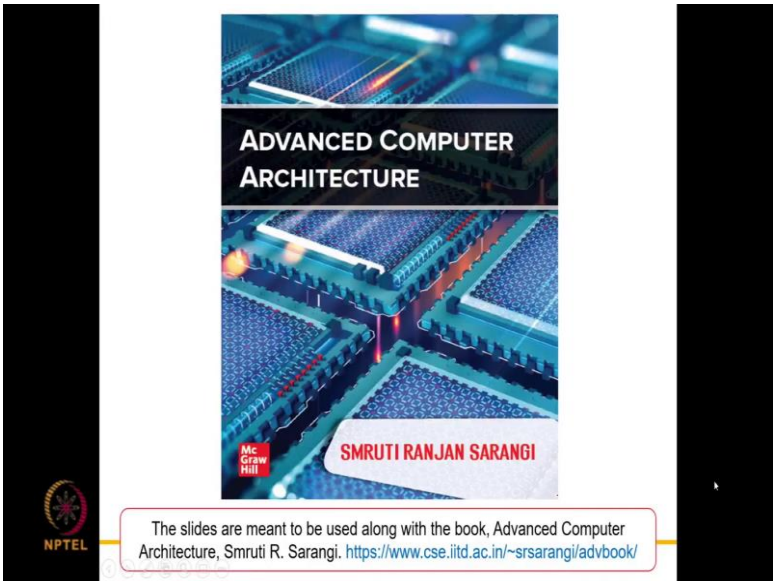
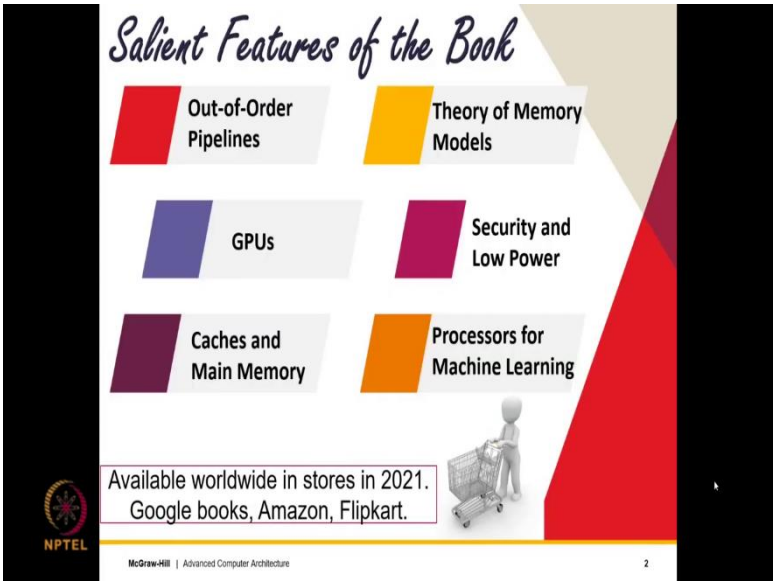
Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Chapter - 09
Lecture - 29
Multicore Systems Part - V

(Refer Slide Time: 00:17)



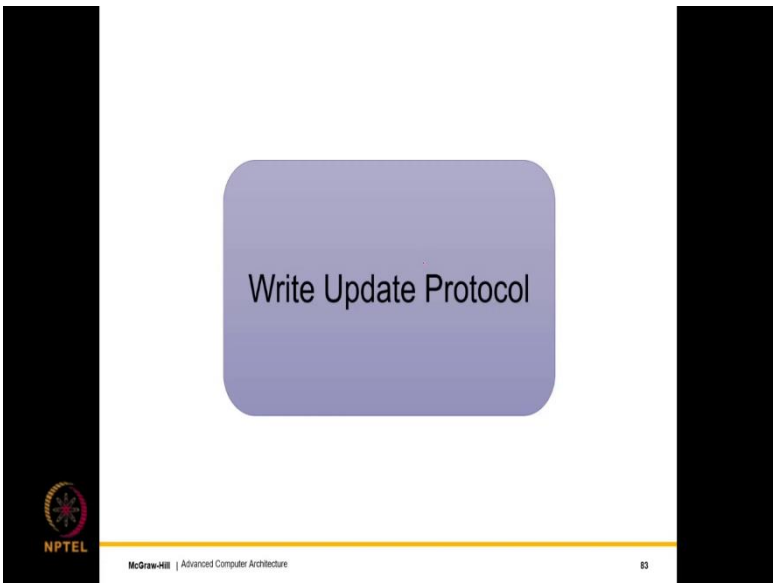
(Refer Slide Time: 00:23)



(Refer Slide Time: 00:38)

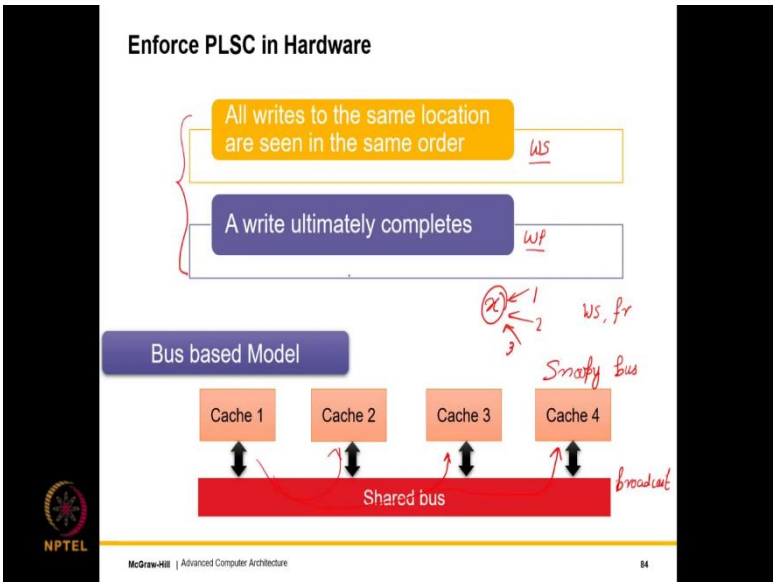
In this video, we will discuss Cache Coherence Protocols. So, cache coherence protocols are required for making a distributed cache behave as if it is the shared cache.

(Refer Slide Time: 00:56)



So, I will first start with the simplest protocol which is the write update protocol.

(Refer Slide Time: 01:03)



So, what is it that we need to ensure? So, we need to enforce PLSC in hardware and from there we came to the two cache coherence axioms. The first is that all the writes to the same location are seen in the same order. So, seen in the same order basically means that all the threads see the same location being updated in the same way; in the sense that if we have a location x if we write 1 to it, then 2, then 3 and so on.

All the threads will agree on this order based on this when we were creating our execution witness we made the WSH and the FRH global; in the sense that, there is complete global agreement about the order of rights to the single location. Then we had one more condition which did not come from PLSC, but which basically said that all writes ultimately complete it means a write message is never lost.

So, let us call this the write serialization condition, let us call it the write completion or write propagation condition. So, these are the two basic axioms that we need to ensure for creating any cache coherence protocol. So, let us start with a simplest model where, all the sister caches are connected with a shared bus.

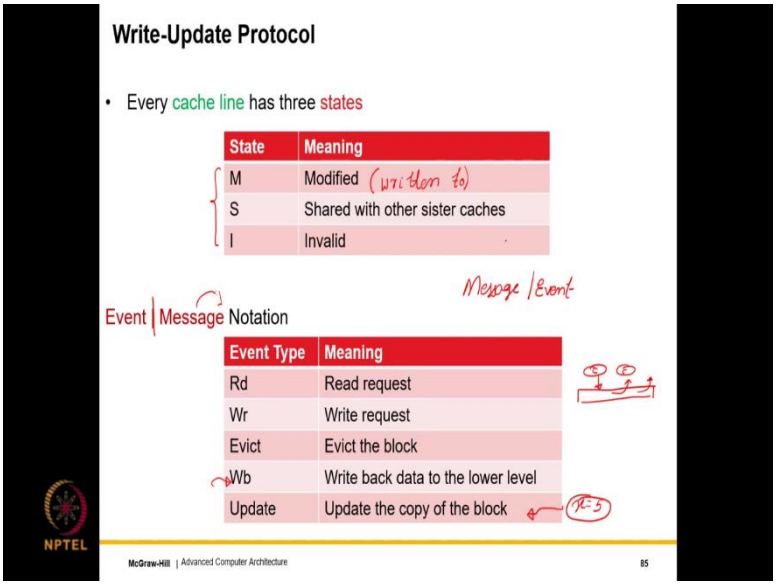
The shared bus is basically a set of copper wires and so, in this case, it is not the case that cache 1 sends a message to cache 2. It just sets the state of the shared bus to the message it wants to send and then all the caches can read it. So, it is like one writer and multiple readers all of them can read it, but of course, most of them will ignore the message because

it will not be meant for them, but the caches that are interested in the message can read a directly.

So, a shared bus by definition is a broadcast media where something is written and everybody else can read. So, these are also called snoopy buses because in a sense, if something is being written the rest of the sister cashes are in effect snooping on the data and the data is visible to them, the message is visible to them.

So, this is also called a snoopy bus. So, this of course, is a very simple design and it will not scale to something more than let us say 8 caches, but nevertheless if we have 2 to 4 sister caches the snoopy design can be used.

(Refer Slide Time: 04:01)



So, we will start with it. So, in a right update protocol what we do. So, this is something that we will continue to do from here on is that we will augment each cache line with three states M, S and I. So, M stands are modified. So, which we have seen this before this basically means that a write has happened, so, it has been written to; S means that the line is possibly shared with other sister caches so, that is the S state.

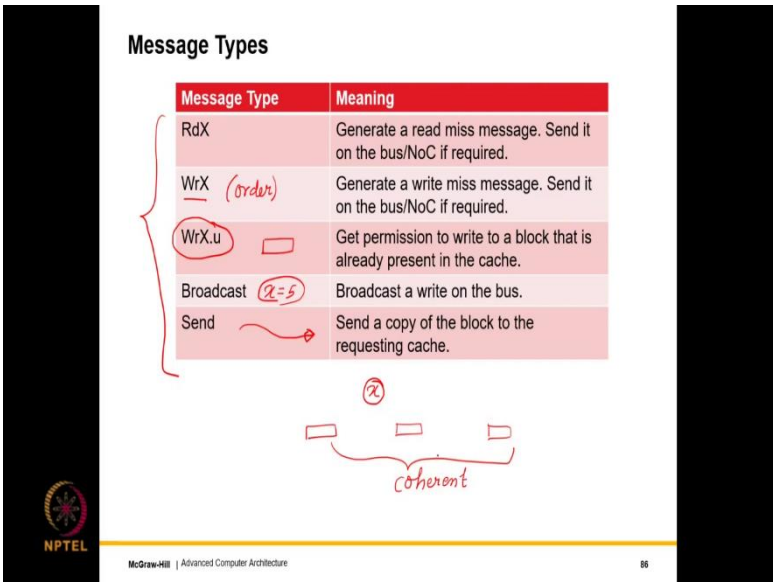
And I mean, it is invalid which means it does not contain valid data. So, what we will be studying in this cache coherence protocols are will be studying a set of state diagrams and a set of transitions a. So, with each edge with each state transition edge, we will have an event in a vertical bar message notation.

So, which basically means if the certain event happens you send this message. Or you do this and also we can have a message event notation where if we receive a certain message from the bus, then we either fire the certain event or we do something. So, let us see.

So, Rd is a read request, Wr is a write request; Evict means that there is a need to evict the block; Write back is we write it back to the lower level and Update is that we receive an update on the bus which sets the value of a certain variable. So, let us say it sets x = 5. So, we quickly update the local copy of x = 5. So, what again is the broad idea over here? The broad idea over here is that we have a shared bus which is just a set of shared copper wires. We have the sister caches connected to it.

The only way for caches to communicate is by setting the state of this bus and of course, we assume that we have a smart circuit a bus master that allows only a single write at a time. And once a write is having been done it is immediately visible to the rest of the other caches and each one of them runs a state machine. So, it is possible that one variable might be present in many caches also then the state machine needs to run on all those cache lines.

(Refer Slide Time: 06:35)



There are several massage type several kinds of messages that can be sent on the bus. So, when we later on extend this protocol to the NoC we will also have the same message format. So, RdX is a read miss message. So, we send it on the bus NoC if required; similarly, WrX is a write miss message.

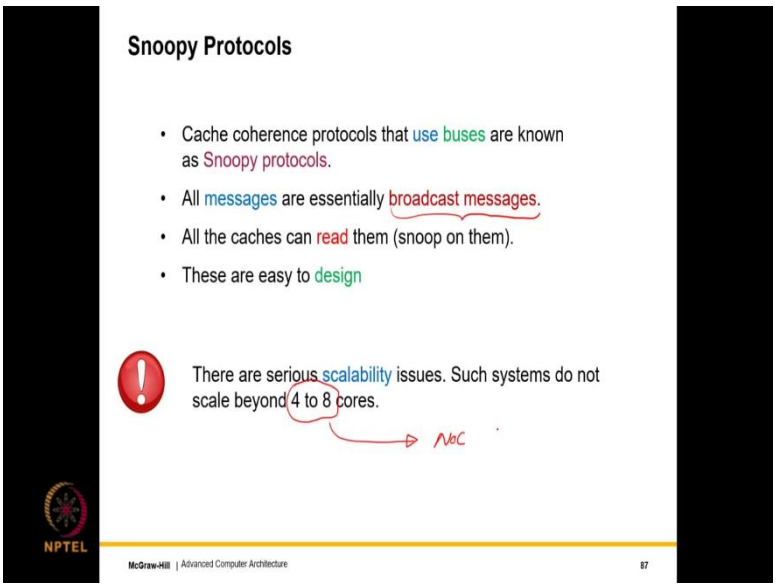
So, recall from our discussion on caches that whenever there is a write miss, we need to get a copy of the block first and then we need to write to whichever subset we intended to write but getting the copy of the block is important. So, this we have already discussed in chapter 7.

So, now, the question is why do we have separate messages a read miss and a write miss? Well, we will see. So, because reads and writes we treat somewhat differently and so, the writes the order to the same location does matter. So, they have to be separate messages and write X. u is just to get the permission from the rest of the sister caches to write.

So, in this case, the current cache actually has a copy of the block, but it does not have the permission to write to it. So, write x.u message is sent to actually get permission to write to the block. Broadcast is where we broadcast a write on the bus we basically say that look I have set x = 5. So, then this is a message to the rest of the sister caches to do something and finally, sent send means send a copy of the block to the requesting cache, if the current cache has it.

So, this is of course, assuming the current cache has a copy of the block. So, what again is the idea? Well, the idea is that the block that contains a variable x might be present in multiple caches there will be multiple replicas. These replicas have to be kept synchronized or what is called coherent, which means that the action axioms of cache coherence need to be followed.

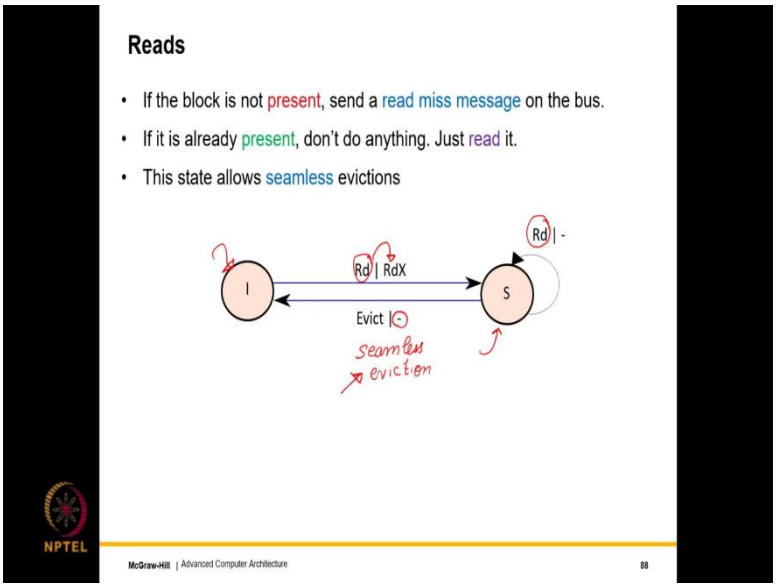
(Refer Slide Time: 08:53)



So, as we discussed cache coherence protocols use buses they are known as snoopy protocols. All the messages are essentially broadcast messages. So, in effect they are broadcast messages and all the caches can read to them and such protocols have serious scalability issues in the sense that they do not scale beyond 4 to 8 course this is the maximum.

So, beyond this actually they do not scale. So, they have this fundamental scalability issue. And, so that is why we will use other protocols that will rely on the NoC not on a snoopy bus, but we will gradually extend our ideas to an NoC.

(Refer Slide Time: 09:44)



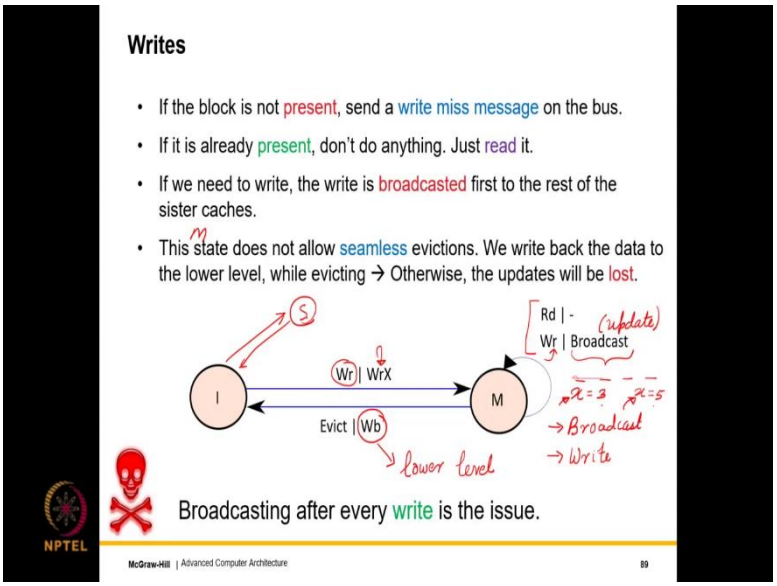
So, let us now come to reads. So, we will be having as I had promised we will be having such kind of state machines. So, for reads if the block is not present, we send a read miss message on the bus. So, let us say we come here we find the invalid state which basically means the block is not present and we want to read. So, we send a read miss message on the bus, fair enough?

If it is already present so, then so, let us say if it is already present in the shared state then we do not do anything we just read it. So, we read it do not do anything. So, this minus indicates nothing has done. And from the shared state also what we do is that we have seamless eviction. So, this is an important thing. Seamless basically means if you want to evict the block we just throw it out and we do not do anything.

So, this is known as the seamless eviction which is a good idea because every time awake there should be no need of actually doing something it will make the process of eviction more complex. So, this for us is a simple state diagram. So, what is this again? If I do not have it, I send a read miss message on the bus.

And then I hope that somebody will respond and if it is if no one response of course, so, within a certain maximum time frame I read it from the lower level. And, then in the shared state and then of course, I transition to the shared state and in the shared state I can keep on reading as well as I can seamlessly evict and the seamless eviction will become a bit deal later.

(Refer Slide Time: 11:44)



So, now let us come to writes and let us look at the same I to M transition. So, in this case if the block is not present I send the write miss message on the bus indicating that I would like to write and of course, some other cache needs to reply within the finite period of time otherwise I read it from the lower level.

Now, let us consider the M state if we need to write the write is broadcasted first. So, from the M state if I want to read I do not do anything I just read, but if I want to write I first broadcast it to the rest of the sister caches and then I do a write. So, this is why this is called an update protocol. So, so what is it that I am doing?

So, what I am doing is that if I want to read nothing needs to be done I have the block, but if I have a write given the fact that all the sister caches have to agree on the order of writes. Any time that a write is actually done, it is broadcast the rest of the sister caches because they possibly might have a copy of the block.

So, they will also update themselves with a copy of the block. So, that is why I do a broadcast and then I do the write. So, that is an important thing and also if you read this chapter in the book so, a subtle point is made that I broadcast first in the sense I tell everybody this is what I am writing and then the sister cache updates its cache line.

So, why not the other way round? Well, the reason is that there might be two concurrent writes. So, let us say one ones to set x = 3, the other ones to set x = 5. So, it goes ahead and writes first, then one sister cache x will be get written to 3 first and in the other sister cache x will get written as 5 first and then they will do the broadcast which is not write.

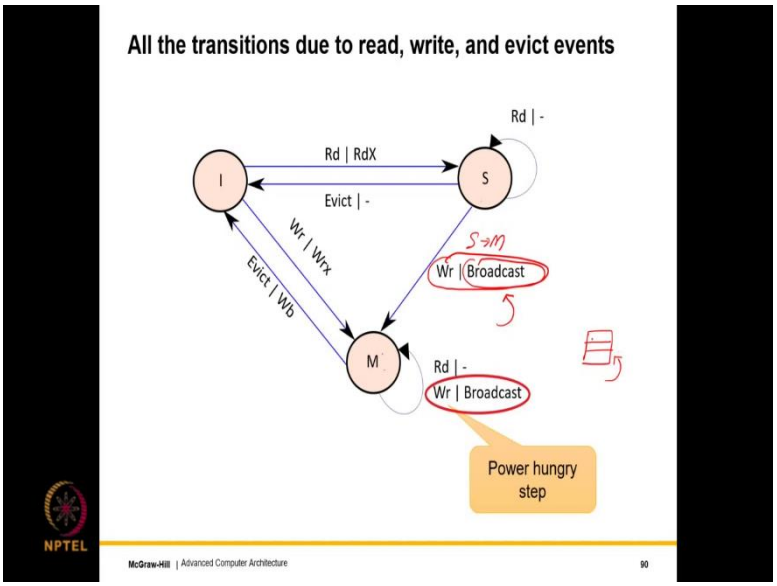
So, if the if there is a shared medium the bus master will ensure that one of these requests gets control of the bus first and then the second one. So, this ensures that there is a global order of writes. So, if let us say x = 3 succeeds first, then everybody records this and then everybody records that x was set to 5.

So, this is the key, this is the broad idea that if you want to write from the invalid state we move to the modified state there we can keep on reading there is no issues, but if you want to write, the write needs to be broadcasted which is of course, is the lower head, but this is why it is called an update protocol that whenever we write we just broadcast the write.

Then the M state, so, this state this state which is the M state does not allow seamless evictions in the sense that if the block needs to be evicted. So, well in this case the block has been modified. So, if let us say that we evict the block; we do not want to lose the update because we do not want to lose the update we write it back to the lower level. So, seamless evictions in this state are not actually allowed.

So, we write data back the data to the lower level while evicting otherwise the update will be lost. And, clearly the most important source of overheads is the fact that we broadcast on every write and this is in overhead in terms of time power and performance ultimately. So, this part of the state machine where we have seen a similar interaction with the S state and an M state. So, let us now complete the state diagram.

(Refer Slide Time: 15:51)



So, the complete state diagram features all the transitions due to read write and evict event is shown over here. So, we have seen this kind of piece meal. So, we have not seen one part I will come to it. So, we have seen the transition from the I to the S state and back. So, in this case, what we do is that we read and then the read miss message is sent over the bus.

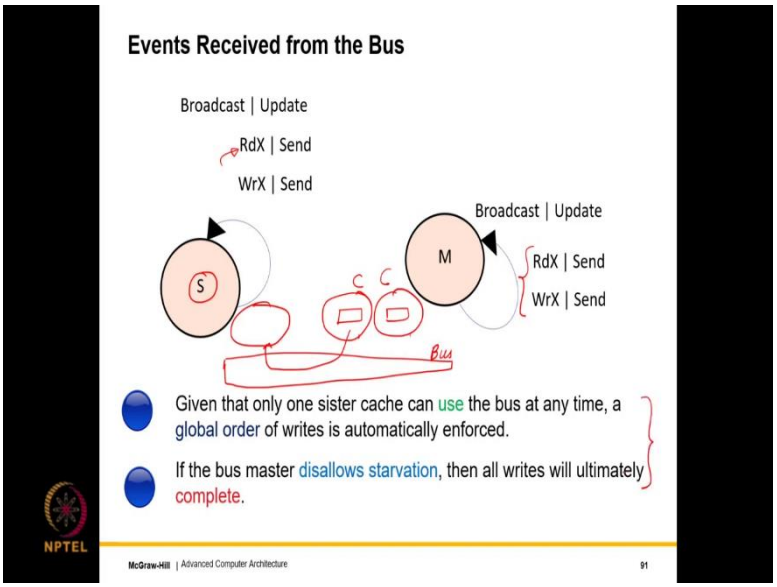
And similarly, there is an eviction seamless eviction from the shared state, but from the modified state it is not seamless eviction. In a shared state we can keep on reading. So, the transition that we would like to introduce over here is actually this transition where from the shared state let us say we want to write.

So, then because the copy of the block is there nobody else needs to send a block, but the block will be now being recorded to be modified. So, a transition from the S to the M states is necessary. So, that is the reason there is an S to M transition and in addition we broadcast the right for the same reason to maintain a global order of writes.

And, such that the rest of the copies also update themselves and needless to say this operation is a power hungry step in the M state because every write involves a broadcast and not just that whenever a cache sees a broadcast on the bus it needs to locally query itself and find if it has a copy of the block or not; if it does, it needs to update it.

So, that is also another source of overhead both the broadcast as well as the fact that all the caches internally have to check and this overhead is clearly avoidable as we shall see later.

(Refer Slide Time: 17:54)



Now, let us look at events received from the bus. So, we can receive several events from the bus. So, let us say we are in the shared state. We receive a broadcast. Well, nothing needs to be done what we do is we update the value. We receive a read miss well; again nothing needs to be done we send the data and we receive a write miss; again nothing needs to be done we send the data.

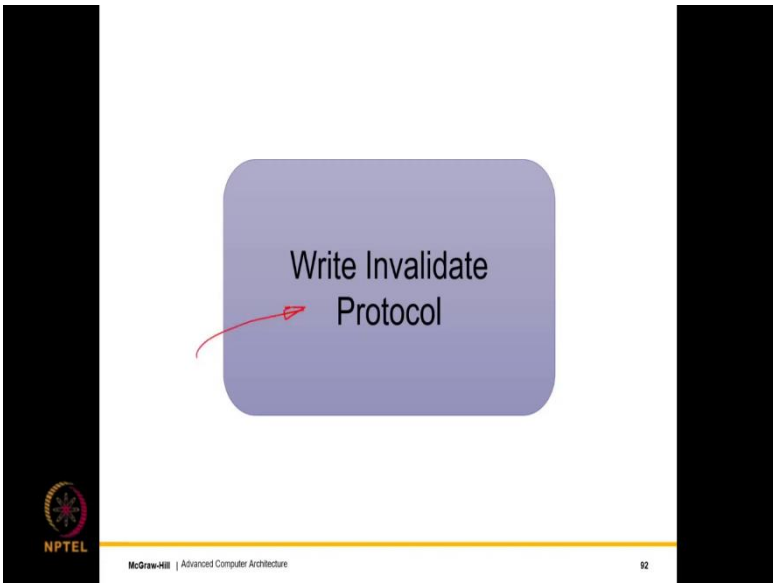
Similarly, in the M state if there is a broadcast message we update the value and any read or write message we send the data. So, of course, here there is a subtle point and we will make a big deal out of the subtle point later that let us say there are two copies of the blocks in two sister caches and now, a new sister cache wants a copy of the block who supplies it this one or this one?

Well, whoever gets access to the bus first supplies it. So, whoever gets control of the bus first supplies the data and the other cache snoops the bus. So, it moment it sees that data has been supplied it keeps quite.

So, here of course, given that only one sister cache can use the bus or get permission to the bus at any point of time a global order of writes is automatically enforce; the same

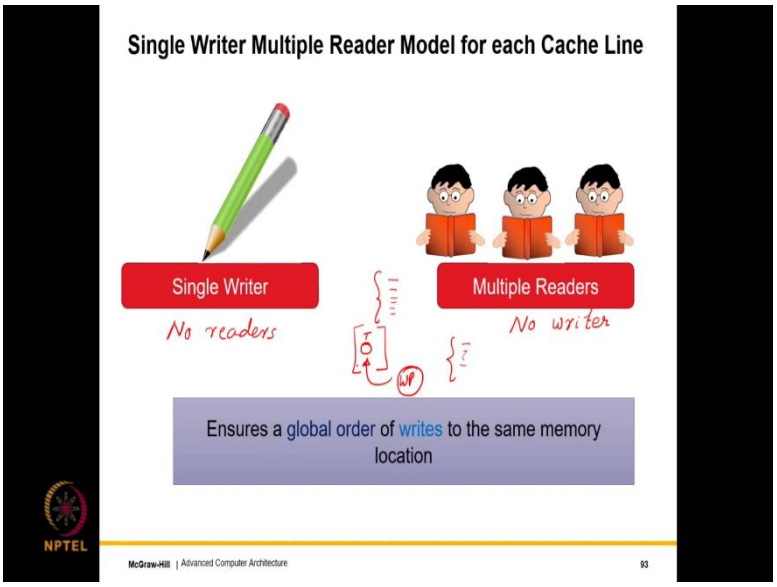
order in which you acquire the bus and also the bus master disallows starvation in the sense if every write ultimately gets access to the bus then all writes will also complete. So, this ensures the axioms of coherence the fact that they actually hold.

(Refer Slide Time: 19:53)



So, what we have been arguing up till now is that the write broadcast protocol has high overheads and that is because on every right we need to broadcast the write. So, we will optimize this and propose the right invalidate protocol.

(Refer Slide Time: 20:13)



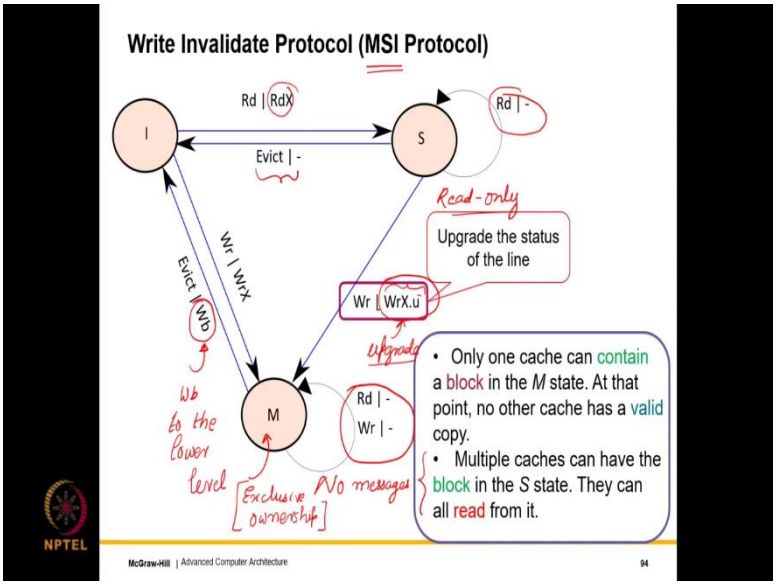
So, what we see is that in practice we do not have that much of sharing. So, it is not that a single line is like concurrently accessed all the time and so, variables are protected within critical sections. So, we typically access a bunch of data then somebody else accesses it and so on.

So, a common pattern is that at one point of time we can have a single writer and nothing else just a single writer no readers. So, maybe I should note it down or we have multiple readers, but no writer. So, if I were to take this particular case in mind, I can create a new protocol.

And let us say for every block if I sort of create some kind of a token and I just pass around the token between the writers. In the sense that whoever holds the token that cache can write nobody else can then the order in which the token is passed will ensure a global order of writes to the same memory location, conceptually.

Furthermore, if let us say a I am a cache and I want to write if it is guarantee that ultimately I will get the permission to write or ultimately I will get this token, then there will be no starvation and all the writes will ultimately complete. Writes or the write propagation axiom is guaranteed. See, if we can design a protocol on these lines, it will be more efficient at least more power efficient than the write update protocol.

(Refer Slide Time: 21:55)



So, in this case, what we will do is I will directly introduce the state diagram primarily because we have already become mature in interpreting these state diagrams from the previous exercise. So, we will directly be in a position to appreciate this state diagram. If you are not able to understand I will request you to go to the previous write update protocol and understand such diagrams in great detail.

So, we will have the same MSI protocol, same three states the that meanings are slightly change I will tell you how? So, let us first look at the salient points. So, it is that only one cache can contain a block in the M state. So, for a given block write only one of the caches can have it in the M state, at that point no other cache will have a valid copy. However, it is possible that multiple caches can have the block in the S state and they can all read from it, but not write to it.

So, we have the single writer or multiple reader's philosophy. So, as per this particular philosophy we will only have a single cache which can write to it in the M state and we will have multiple reader's or we will have multiple readers, right "or" in the case in the sense exclusive OR.

So, let us look at the state transitions. In this case we have the I state. So, from the I state if we have a read request, we send a read miss message and be transition to S and of course, as I have said if we do not get it in finite time we read from the lower level, but we still transaction to the S state and a similar transition on a write to the write miss we send a write miss message and we move to the M state.

In the S state a cache can continue reading there is no problem. So, think of the S state as a read only state for this protocol of course, in the previous protocol we could write here, but not now. This is only a read only state you want to keep reading be in the S state, but let us say in the shared S is for shared. In the shared state we decide to write. Well, then it is necessary to get permission to write first we just cannot write like that.

See if there is a need to write u write x. u and upgrade message needs to be sent to the rest of the caches and once there is an agreement. So, agreement by the fact that the message is actually sent on the bus and you see the message being sent on the bus and since the bus enforces a global order everybody else would get it.

So, then the rest of the caches will essentially invalidate themselves which means that they will discard the copies of the blocks that they have and so, then what can be done is that from the S to M state we can make a transition and here is the fun part. The fun part is in the modified state we can continue to read and write, no messages are sent. So, this makes this part of the protocol extremely power efficient.

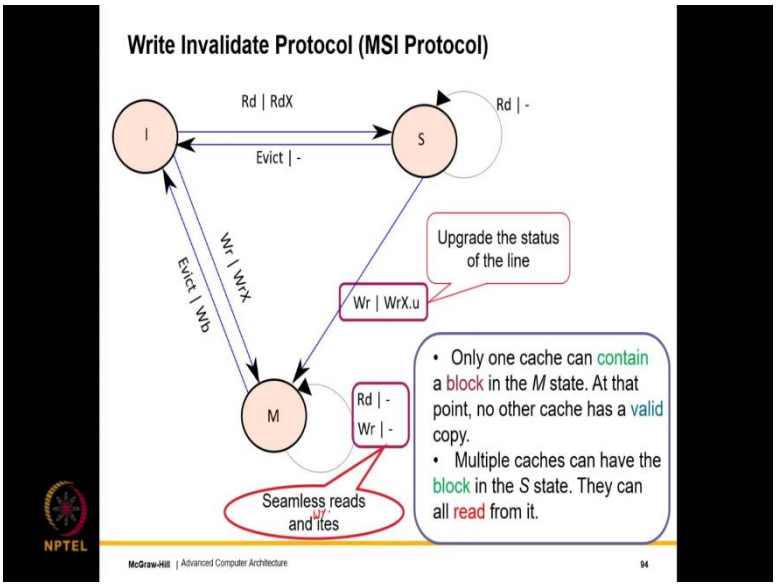
The reason being that we can continue to write in the M state without broadcasting a message and broadcasting a message the point that I made before that first power goes in the broadcast and second power goes in actually reading in a for all the caches to actually check is the broadcasted value is there in their caches or not the broadcasted address.

But, in the invalidate protocol the nice thing is that since only cache has a has the block in the M state you are sure that no other cache has a copy. So, you want to write, we will continue writing. Now, here are a few subtle points. So, the evictions are still seamless in the shared state that is mainly because it is read only nothing has been modified.

So, if you want to evict well we just go ahead and evict, but from the M state evictions are not seamless because the block has been modified. So, in this case write back to the lower level is necessary. So, this is the key idea here that we do not allow readers and writers to coexist and if a cache wants to write it better be the sole exclusive owner of a block.

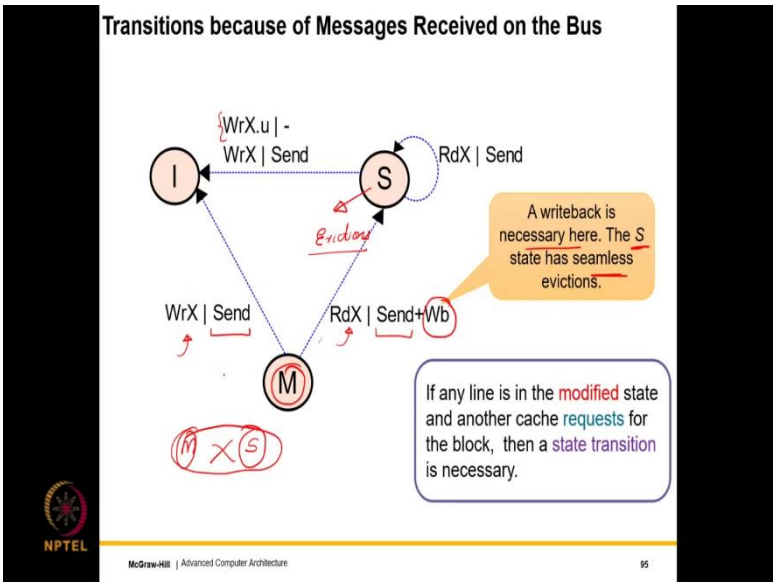
So, the M state enforces this exclusive ownership. So, then it means that you want to write while keep writing. And further more the order in which the caches reach the M state is a globally visible order and that ensures the write civilization axiom of cache coherence. So, the upgrade was the new part and that we have discussed, so that we might not be able to see it.

(Refer Slide Time: 27:34)



w r is d o y.

(Refer Slide Time: 27:42)



Let us now look at transitions because of messages received on the bus. So, clearly if I am in the invalid state, then transitions do not matter. Transitions only matter for the M and S states. So, in the S state several messages can come; a read miss message can come, in this case the data is sent the block is sent. Unless of course, another cache sends it earlier in a sends it gets access to the bus earlier and sense it, the contents of the block are sent.

But, if let us say write upgrade message is received, then there is a need to transition to the invalid state because it is not possible for one cache to have a block in an M state same time another cache to have it in the S state – this is not possible. So, we either have a single writer or we have multiple readers, we never mix them. So, because we do not mix them what we do is that we invalidate ourselves.

So, this is where the name of the protocol also comes from the write invalidate protocol and similarly, if there is a write miss the block is sent. From the M state we can either have a write miss or a read miss message clearly no upgrade messages because there is no shearing. So, if there is a write miss message which means another cache would like to write then what happens is that the data is sent the block is sent to the other cache.

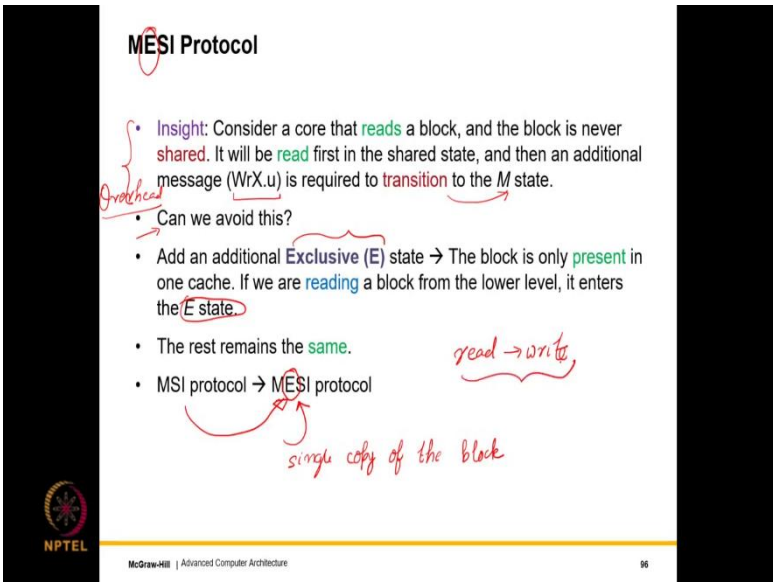
There is no reason to do a write back over here because the contents of the block whatever has been written to will be transferred to the other cache and then the other cache can do a write back if there is a need. In this case, if there is a read miss. so, then because if any line is in the modified state a state transition is necessary if there is another request.

See if there is a read miss then the current line has to transition from the M to the S state the reason is that we cannot have an M and S state together; M state in one cache and S state in another cache either one writer or all readers. So, get into the all S readers we have an M to S transition and the block is also sent.

In addition, a write back is performed to the lower state because we have seamless evictions from this state. So, we can just toss a block out, but what if it was modified and it came from this state? So, that is why to enable seamless evictions in the shared state, the data is the block is written back to the lower level.

So, this is why a write back is necessary because the S state has seamless evictions, which means we can just evict a block without doing anything. So, this was the MSI protocol the two-state diagrams for cache coherence with the write invalidate mechanism. So, let us now make this more efficient.

(Refer Slide Time: 30:58)



So, the crucial insight over here is that consider a core that reads a block and this block is never shared. So, what will it do? It will first read in the shared state and then if it wants to write it will send an additional upgrade message write X . u and then transition to the M state. This for us is an overhead because if let us say we just read first and write later, it is clearly an overhead.

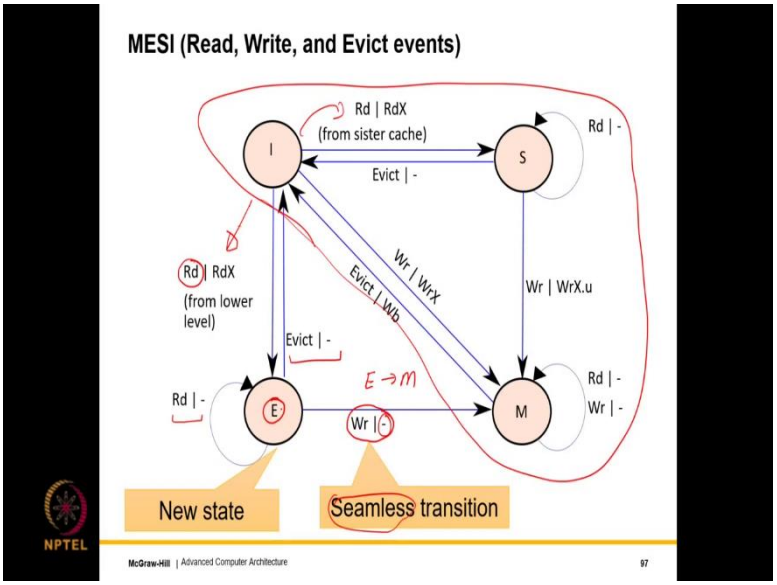
So, we should avoid this that is why we propose the MESI protocol with an additional state called the exclusive state or the E state. The E state essentially indicates that the block is only present in the current cache and how do we set it in the E state? Well, if we have a read miss and we request the rest of the caches, but none of them has a copy and we read it from the lower level then the block is directly put in the E state or in the exclusive state and the rest remains the same.

So, the MSI protocol actually becomes the MESI protocol where we have an additional E state or exclusive state, which indicates that there is a single copy of the block. So, this is what it indicates there is a block that is a single copy. We do not have multiple copies we just have a single copy just one.

And so, this is the MESI protocol which we will look at next. Say, it will reduce some of our work some of our additional work particularly that captures the pattern that we read something and then we write. And we are of course, the solo reader and the solo writer;

previously more some messages were required to be sent on the bus, now nothing will be required. So, let us see.

(Refer Slide Time: 33:06)



So, in this case in the MESI protocol this is how we actually work instead of three states we have four states and we can clearly see the e state over here. So, let us start describing the protocol from the E state first and then we will see the rest of the states and they are roughly the same.

So, this state diagram should be correlated with the earlier state diagram on the basic MSI protocol and we will take it from there. So, initially from the I state if there is a read and we are reading from the lower level not from a sister cache. so, we clearly make a differentiation over here whether we are reading from a sister cache or we are reading from the lower level.

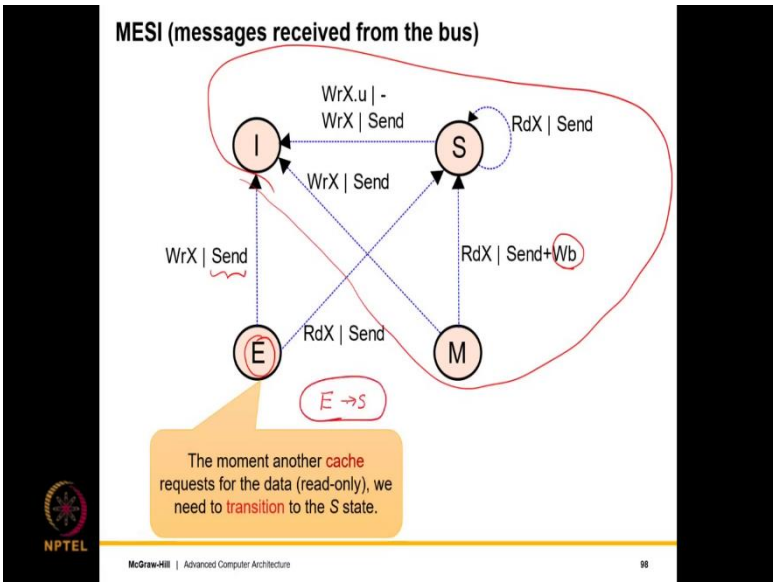
If we are reading from a sister cache we transition to the S state which is the usual thing that we had seen previously with the MSI protocol, but in this case if a read is being made from the lower level after sending a read miss message it clearly indicates that no other cache has a copy. So, this is what it indicates. So, in the other cache has a copy we directly transition to the exclusive state. So, this is clear that we directly make a transition to the E state.

In the E state we can keep on reading there is no problem and we can also seamlessly evict it because no write has been done. So, here is the trick we from an E to an M transition can be done if we desire to write, nothing needs to be sent on the bus. So, this is totally seamless right nothing is sent on the bus and this transition is the most important new aspect of this protocol that if let us say we read something and then we desire to write to it, then we can easily make a transition.

And, this transition does not require any additional permissions from the bus and as you can see we transition from E to S, and, so nothing else is required. So, these are only the read write and evict events I will look at the events from the bus in the next slide. The rest of the transitions for the M, S and I state remain the same.

So, this part is identical I will not discuss this. So, we have already seen this part, it remains the same it is identical and the only difference is we make a differentiation between the reads from the lower level and reads from a sister cache. From a sister cache goes to the upper triangle otherwise we come to the E state as we have described.

(Refer Slide Time: 35:59)

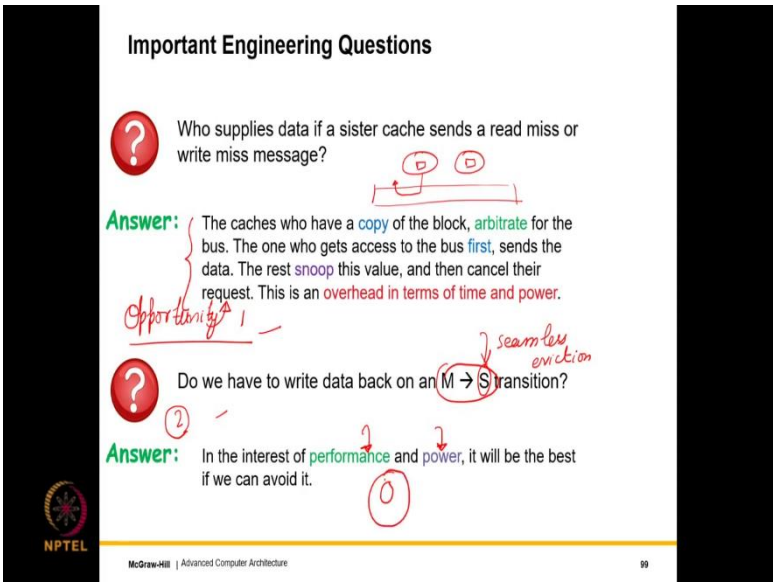


So, here are the transitions on bus events. So, let us say we are in the E state, we have a write miss. So, then it is guaranteed that the other sister cache will not have a copy of the block. So, we send the block. If you have a read miss then of course, there is a need to transition to the shared state because multiple cashes cannot have if let us say multiple caches have the block in with read only permissions all of them need to be in a shared

state. The rest of the transitions again for this upper triangle remain the same which is in the shared state we can keep reading write and if let us say read miss comes we send the data and from a M state if a read mix read miss comes we do a send and write back first seamless evictions.

And from an M state write miss comes with send and from a shared state any write we invalidate and send if the other cache does not have a copy. So, this was a simple addition to the basic MSI protocol to make the MESI protocol and the important point is the E to S transition fair if let us say another cache requests for a copy we transition to the shared state.

(Refer Slide Time: 37:25)



So, we are unfortunately not satisfied, we have gone far, but we still want to go even further. So, in this case we would like to ask a basic question. So, who supplies the data if a sister cache sends a read miss or write miss message? So, see that is an important question.

Up till now, we have been saying that look a multiple system caches have a copy of the block any one of them requests the other can whoever gets access to the bus first will send it. So, this does have issues, this does have overheads you need additional circuitry. So, the caches serve a copy of the block arbitrate for the bus. So, the send request to the bus master they arbitrate for it.

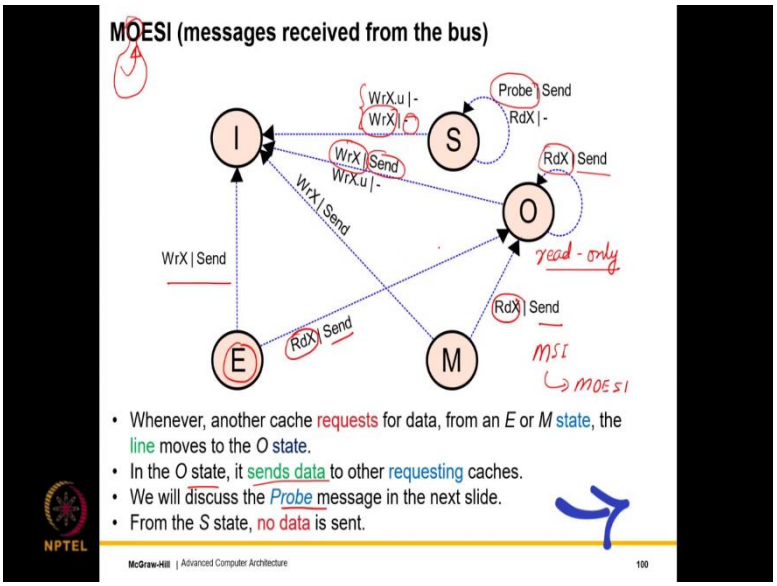
Source of overhead number 1. The one who gets access to the bus first sends the data. So, whichever cache gets accessed to the bus first will send the data. The rest will snoop this value and see that look yes the data has been sent. So, some sort of an additional state machine has to be maintained and then they will cancel their request. So, this is a substantial overhead in terms of time and power and in terms of time.

And, power clearly this additional state machines that time, power, area, performance overheads are there. So, that is why we would like to reduce this overhead as much as possible that is or let us say opportunity number 1 that for providing data if we can add an extra state well, why not. Opportunity number 2: on an M to S transition we have basically been writing back data. Why? Because we want a seamless eviction in the S state.

So, because we want a seamless eviction over here whenever there is a transition from modified to shared we write data back. Of course, this does have an implication on performance as well as power. Is it actually required can you do it later? can we do it lazily? So, in this case, what happens is that whenever there is a transition we write data back.

So, there is no opportunity for an optimization in the sense we cannot do it slowly or lazily. At a later point of time, where the lower level memory is slightly less loaded. so, such opportunities are not there with us. So, can we explore this? So, there are two opportunities 1 and 2. Let us solve them and how have we been solving it? By adding an additional state colon owner state.

(Refer Slide Time: 40:19)



So, now, the state diagrams keep getting more complicated, but the assumption is that you have become an expert in this kind of stuff already. So, here what we do is we have an owner state, which is an additional addition on the MESI protocol. So, we have a MOESI protocol where the additional owner state is extra.

So, the idea the owner state is that looked yes, the data can be shared fair enough. Multiple copies can have multiple caches can have a copy of the block, but I am the owner. If I am the owner, then it is my job to send data. So, let us slowly glow go through it. So, I have deliberately reversed the order of the slides.

So, in this case, we talk about the events from the bus first; messages received from the bus first the transitions due to them and then we talk about events. We talk about the read write and evict events. So, the order is reversed and it is important because it is easier to understand if we present it this way.

So, whenever another cache requests for data from an E or M state from the E state. so, of course, this part remains the same. And, but from an E or M state whenever another cache request for data. so, previously what use to happen? Well, what use to happen is that, there was a transition to the shared state. So, let us look at the read miss messages from the E or M state.

Previously they used to go to the S state, but now we do not do that. Now, whenever there is a request we send the data whenever there is a request we send the data and we make a transition to the O state which means that this records the fact that the current cache is the owner of the block. So, it is true that multiple caches have copies of it and that to in a read only state. So, O state is the read only state we are not allowed to write.

But, that is not important. What is important is that the current cache is an owner. So, later on if a new cache joins and once a copy of the block the owner will supply. So, if let us say new cache does not have a copy of the block it sends a read miss message it is the job of the owner to supply. And, if another cache wants to upgrade well the owner of course, invalidates but if there is a write miss again the owner supplies. So, that is important.

So, what do we do? So, what we do is that from the E or M states well, if the write miss comes then of course, we send the block send the contents of the block and we invalidate ourselves which in the any case has to be done because it is a write invalidate protocol. Otherwise, what happens is that in the O state we send the data to other requesting cashes.

And, we enter the O state when we receive a read miss to the E or M states and this records the fact that is my I am an owner it is my job to send. So, we are not going to complete on the bus anymore. So, this makes this part of the protocol very power efficient and the owner's job is just to send an answer all queries for a given block.

Then let us look at the S state. The S state by and large remains the same. In the sense if anybody else requests for the data then there is an invalidate there is a key important reference here. So, previously when there was a write miss the S state used to possibly send a copy of the block, but now that is not required.

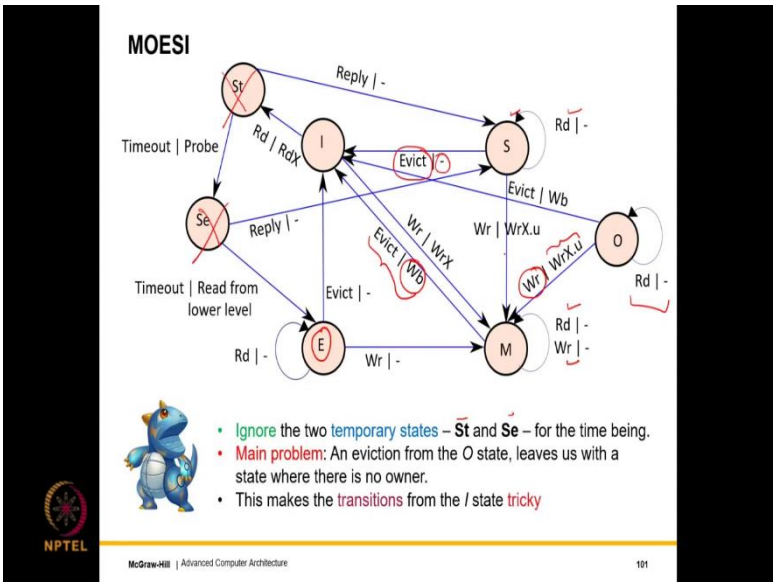
So, just look at this. So, now, that is not required because you already have an owner and the owner will send write. So, other S states need not concern themselves with actually becoming an owner. So, basically they do not really try to send a copy of the block and similarly, if there is a read miss also they do not send a copy of the block, they only send a copy of the block if let us say they get a probe message which we will see.

So, the probe message is sent to handle a certain corner case. So, this will be discussed in the next slide. So, only on a probe message is the S state responsible for sending data otherwise it is not. So, by and large this simplifies our state machine to a large extent; in

the sense that whenever a copy of a block needs to be send we do not have to actually compete an arbitrate for the bus and send.

So, this part of operating the protocol becomes substantially simpler, but of course, to do that from MSI we have moved to MOESI, no problem. So, I hope that by this point these five states are well understood and the fact that the owner state is actually the owner in this case.

(Refer Slide Time: 45:40)



So, let us take a look at our familiar friends, MIS and M; the new states we add three more states, right. So, we add two temporary states St and Sc and we are an owner state O. So, let us go slowly. So, we will not go very quickly.

So, let us for the time being ignore the two temporary states St and Sc. So, let us just ignore them for the time being. So, what is the main problem? So, let us take a look at what happens with the owner state. So, for an internal read, write and evict event we actually do not enter the owner state. So, see just see that there are no incoming arrows to the owner state.

So, I become an owner if I was let us say previously I had it in I had a copy of the block in the modified or exclusive state and then somebody else wanted read access. So, then I become the owner. So, in the owner state if I want to read well, like if continue to read

there is no problem, but if there is a need to write then we need to invalidate the rest of the copies.

So, write and upgrade messages sent on the bus if there is a write request and for the M state everything remains the same. So, we do not have to bother and similarly, for the S state also everything remains the same. So, what is important is that we need to take a look at the owner state. So, the owner state as I just said it is again a read only state.

So, reads are not an issue and writes will require an upgrade to the M state. Let us now see what is the big deal with eviction? So, eviction in this case is the same for S, M and E. So, eviction in this case is that from an E state we seamlessly evict; from the M state we evict it and we do a write back to the lower level this again is required and from the S state we evict and it is a seamless eviction. So, we do not do anything.

Now, the important thing is what do we do from the O state? from the O state when there is an eviction what we actually do is that we do a write back. So, this write back is actually required. The reason it is required is basically because we transition to the O state from an M state as you can see in the previous slide and here no write back is being done. So, in this case no write backs are being done.

The O state possibly contains a modified copy and since it contains modified copy which is not reflecting in a lower level, whenever there is an eviction from the O state there is a need to actually write back. And of course, we can have smart versions of the protocol which can actually once block reaches the O states lazily or opportunistically write it back to the lower level whenever the lower level is free. So, that can be done.

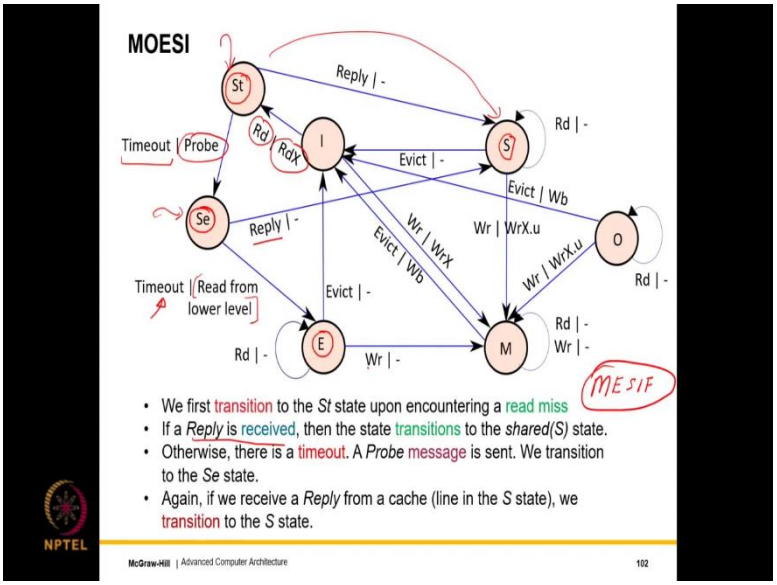
But, in any case when we are evicting in the O state we need to ensure that the block is reflecting in the lower level because the O state possibly contains modified data. And, also the other thing is that any transition from M to the O state, when let us say when I consider piece of data which actually does not leave the caches so, there is no reason to actually write it back from an M to S transition it will go from M to O.

And, later on if you want to write from the O state well again be transition back to M. So, there are no write backs involved. The problem is that if we decide to evict from the O state the first problem is we need to do a write back well that is ok that is not that bad, but the other is that this block will not have an owner.

So, we might have multiple other caches that have it in the shared state, but none of them is actually the owner because there used to be one owner, but we evicted the block and so, now, currently there are no owners. So, there are sophisticated versions of the cache coherence protocol where we can search the rest of the caches and designate one copy as the owner. This of course, is tricky.

So, because this will take more time, it is not tricky, but the overheads are more. So, what is done in this case which of course, is not all that common is that we introduce two temporary states to take care of this and the transitions on the I state become more involved. So, we will see why and how in the next slide. So, now, we will discuss these two temporary states St and Se.

(Refer Slide Time: 50:57)



So, the rest remaining the same will confine our attention to this part of the diagram. So, let us say that from the I state there is a read miss. Let us first see what happens if there is a write miss in the I state. So, there is a write miss in the I state well what is done is that we simply invalidate the rest of the copies.

So, all of them are invalidated and if there is an owner then the owner supplies the data and if there are no owners. So, then either one of the caches supplies the data or we read from the lower level. So, that really does not matter because since writes are often of the critical path we can afford to read from the lower level as well that will not be that is not that serious a thing.

So, that can be done, but what we are more concerned about is reads. So, this is a read miss from the I state. so, this is the read and the read leads to a read miss from the I state, then we transition to the St state; St is the first temporary state. In this case, if a reply is received which means the valid owner exists, then we have a regular state transition to the S state because we received a reply nothing needs to be done.

We just transitioned to the S state because the owner supplied a copy of the block, otherwise there will be a timeout in the St state, which means that look I waited and I did not get anything. So, if there is a timeout a probe message is sent. So, what the probe message does is. So, you recall that original probe message where if let us say a probe message comes then from the S state we send a copy of the block.

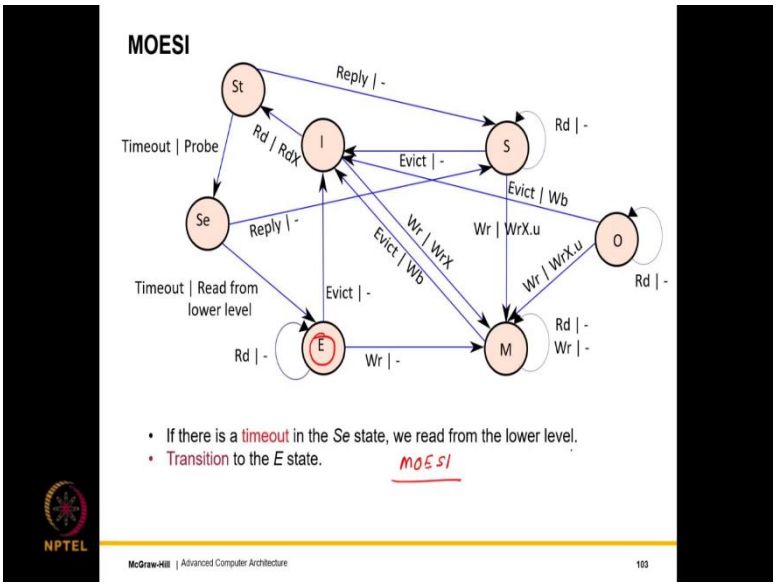
So, in this case, what we do is that we send a probe message and the probe message is sent to all the rest of the sister caches and let us say multiple sister caches have a copy of the block like the previous mechanism they arbitrate for the bus and one of them sends a copy, but in this case because probe messages are rare the additional power over heads are kind of low.

So, then we enter the Sc state. So, we wait for a reply to the probe message. So, if you get a reply which means the one of the caches had it in the shared state, well we are happy. So, we transition to the S state and we resume our work; otherwise if there is a timeout which means that the rest of the caches do not have a copy, then there is a need to read it from the lower level read the block from the lower level.

And, we transition to the E state or the exclusive state. So, because there has been a read from the lower level so, a transition to the exclusive state is required and the rest of the state diagram works the way that it used to work. So, in this case, the aim of these two temporary states St and Sc is to basically take care of the case when there is no owner.

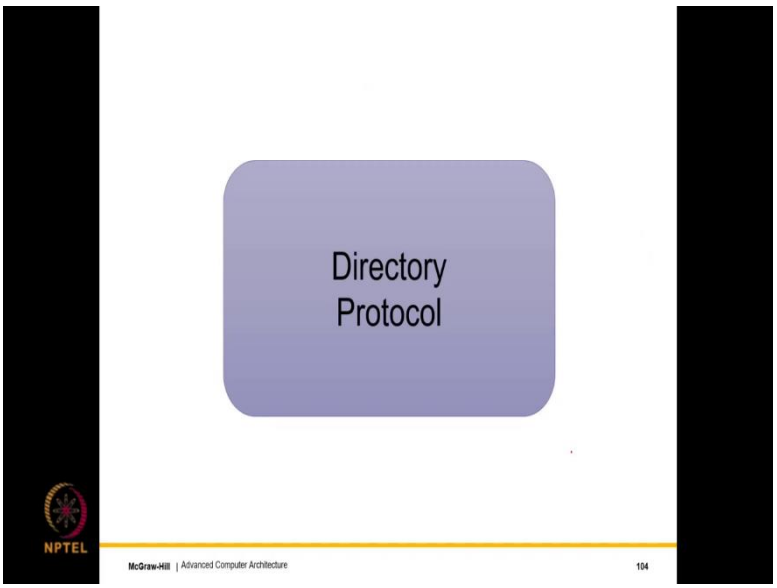
So, I really speaking we should have some way of ownership transfer, so, there is a protocol called the MESIF protocol, where F stands for the forwarding state that does some of this. So, I will not describe it, but you can take a look at it in your free time.

(Refer Slide Time: 54:36)



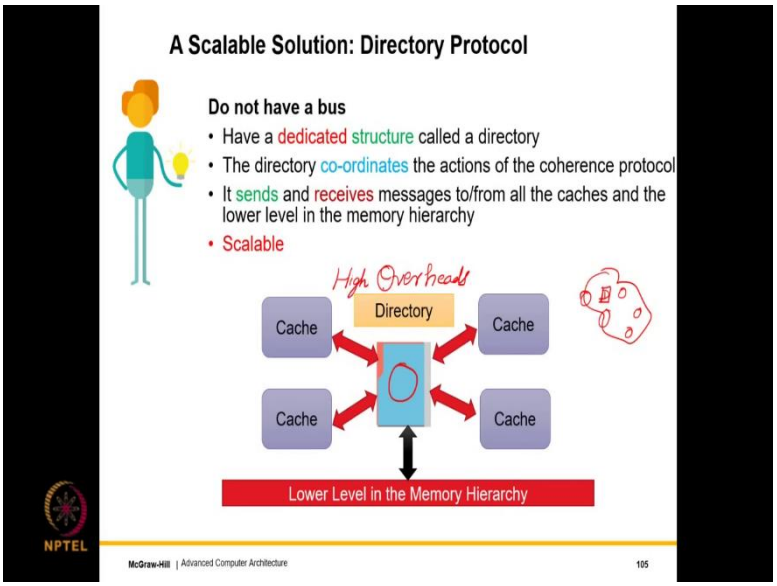
So, as I have said if there is a timeout in Sc state we read from the lower level and we transition to the E state. So, this completes our description of the MOESI protocol with a right invalidate flavor and since all of these were using a bus at their core they have scalability issues. So, we will try to fix the problem with scalability next.

(Refer Slide Time: 55:07)



So, we will introduce the directory protocol which to a large extent a very large extent rather tries to fix this issue. So, this is the current state of the art for a bigger system.

(Refer Slide Time: 55:20)

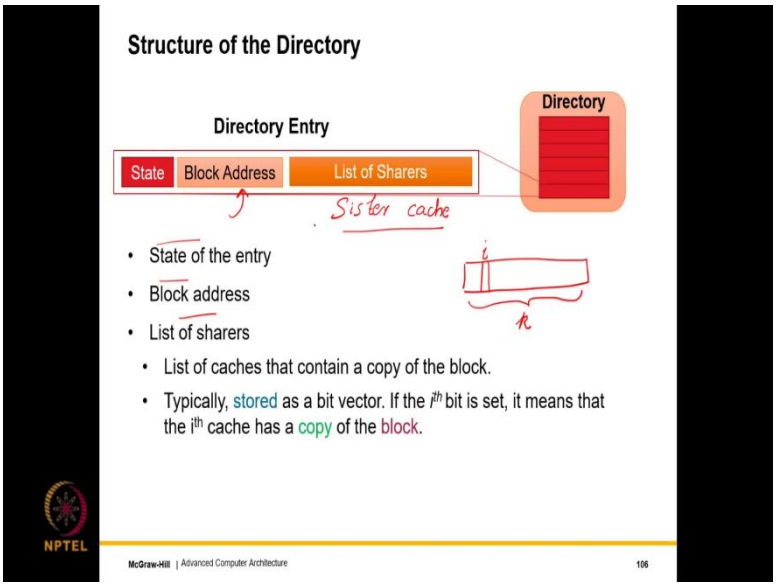


So, in this case, we do not have a bus we have a dedicated structure called a directory and so, the directory is one of the nodes that is connected on an NoCs and the complex NoC we have our caches kind of scattered everywhere and then we have one centralized structure called a directory.

So, a directory coordinates the actions of the coherence protocol it sends and receives messages from the caches and the lower level. So, we can think of the directory as like one centralized coherence protocol ensures and a caches primarily communicate with the directory and the directory orchestrates the entire coherence protocol for the caches.

So, of course, the snoopy protocol is faster in a smaller system given the fact that we have a bus it is faster is just that it fails to scale, the directory protocol to begin with as high overheads, but it scales. So, for a large system a directory is required not a snoopy, but if small system a snoopy protocol is smaller and efficient.

(Refer Slide Time: 56:29)

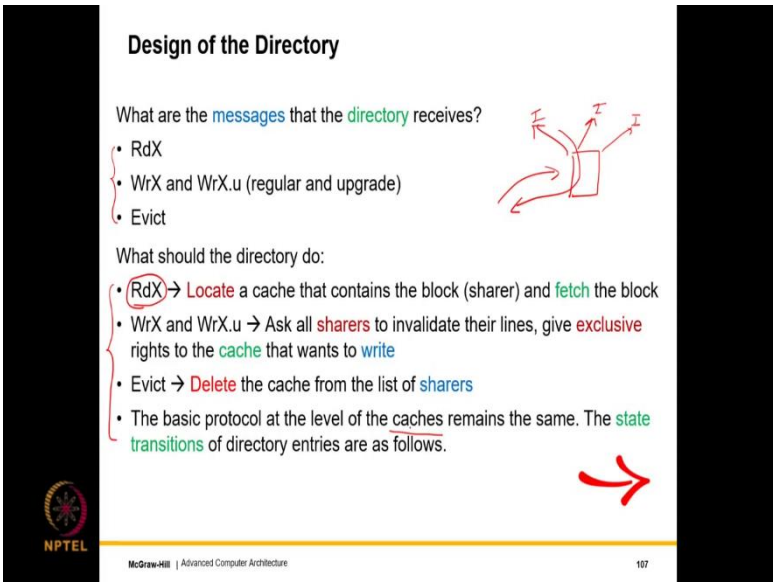


So, here our basic philosophy is slightly different. So, the basic philosophy is that a centralized directory for each block maintains a list of sharers. So, a sharer is basically a sister cache that contains a copy of the block. So, we explicitly maintain a list of sister caches that contain a copy of the block and each such sister cache is called a sharer. So, we explicitly maintain a list of sharers or a list of sister caches that maintain a copy of the block.

And, furthermore some amount of state is kept with the directory. So, we have a state of the entry, we have a block address and a list of sharers. List of sharers is typically maintained as a bit vector. So, let us see if there are 64 cores, then we maintain one bit per core where we are assuming that we have one sister cache with each core or let us make it general let us say we maintain one bit per sister cache.

So, in this case what happens is that we for the i-th bit indicates whether the block is present in the i-th sister cache or not. So, it is just a simple bit vector we maintain. So, if we have k sister caches so, we maintain k bits over here. So, we have a list of sharers the block address and a state. So, the philosophy here is different we do not have a shared bus we have an on chip network.

(Refer Slide Time: 58:11)

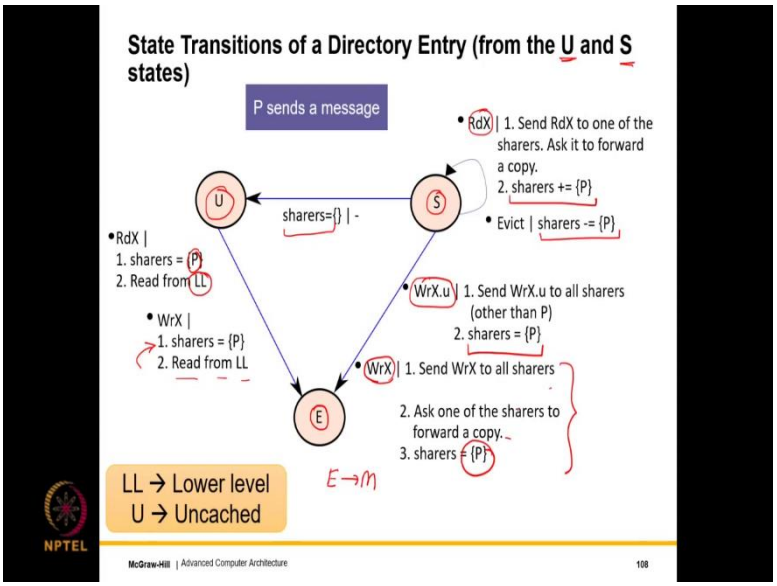


So, what are the messages the directory receives? The same messages read miss, write miss, upgrade and evict what should the directory do. So, well from the point of view of an individual cache the state transition diagrams remain the same. So, they actually do not change instead of dumping a message on the bus they simply dump it they simply send it to the directory. So, that does not change, but the directory of course, is a new beast that we need to look at.

So, when it receives a read miss message it locates a sister cache that contains the block that is a sharer and it fetches the block from it and it transfers, or it asks that cache to transfer. A write miss message it ask all sharers to invalidate their lines and then you need to give exclusive writes the cache that wants to write. So, let us say I want to write well I send the directory or write miss message it sends the message to all the sharers.

All the sharers invalidate their copies and it also requests for a copy from one of the sharers and this copy is transferred to the requesting cache. Evict, well, we can have different versions of the directory protocol, in one case we do not inform the directory in one case we do. So, let us assume that we inform the directory. So, once a block is evicted we delete it from list of sharers. And, the basic protocol as I said are the caches remains the same; the states transition or directory entries are as follows.

(Refer Slide Time: 59:55)



So, we will have three states U stands for un cache in the sense it is not cache; S stands for shared and E stands for exclusive and we assume that each of the individual caches follows the MESI protocol. Well, MOESI is not required with a directory an on a is not required.

So, let us go from U to E. So, let us assume that it receives a read miss message from let us say core P. So, it is actually processor P assuming that it is for the L 1 level. So, that same terminology L can been kept. So, we set sharers = P and we read from LL being the lower level because it is not cache. So, the read from the lower level and if there is a write miss message, then also it is entering an exclusive state in the sense nobody else has a copy.

And, since E to M transitions are silent, the directory will have no idea whether an E to M transition has actually happened. So, for that reason what is done is that the directory only maintains a single state, which is exclusive and in such sharers = P and it initiates a read from the lower level. So, now, let us look at the S state. So, these are the transitions of a directory from the U and S states. So, they originate from the U and S states not from the E state.

So, the S state let us say that we receive an evict message. So, an evict message we just delete the list of sharers and if the set of sharers become empty we enter the un cached state. If we need a receive a read miss message, then the read miss message is sent to one

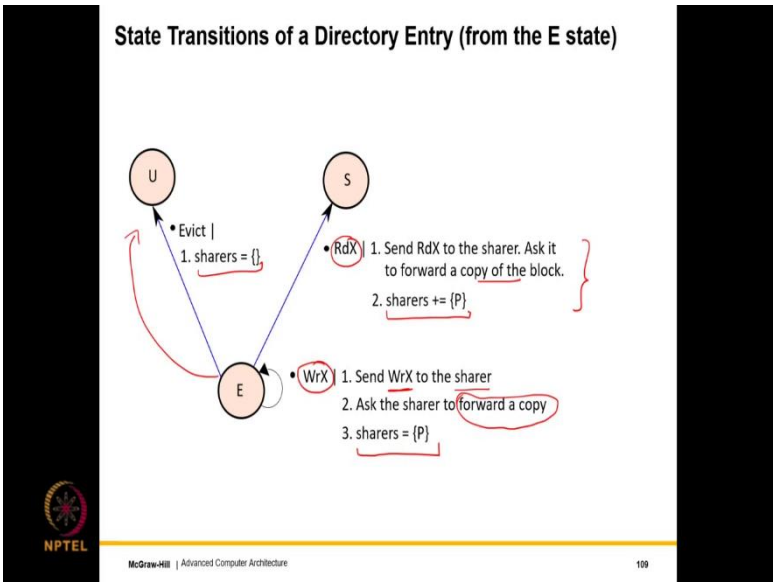
of the sharers and the directory will ask that sharer to forward a copy directly to the requesting cache over the NoC.

So, it need not be sent via the directory it can be send directory and this is more efficient and the requesting cache is made a sharer. So, write the sharers += P means we make it a sharer if there is an upgrade message then what needs to be done is that this miss message needs to be sent to all sharers.

And other than the requesting cache of course, and it actually told the look all of you kindly invalidate yourselves. You do not have to send a copy because it is an upgrade message and the set of sharers is only a single entity, which is the one that is requesting a write permission and if there is a write miss well, if there is a write miss we need to do three things.

We send a write miss to all sharers we ask one of the sharers to forward a copy and a list of sharers is just P. So, this I hope was very straightforward of what the directory is going to do. So, there is no hidden you know rocket science or magic sauce in here. This part is relatively simple.

(Refer Slide Time: 63:08)



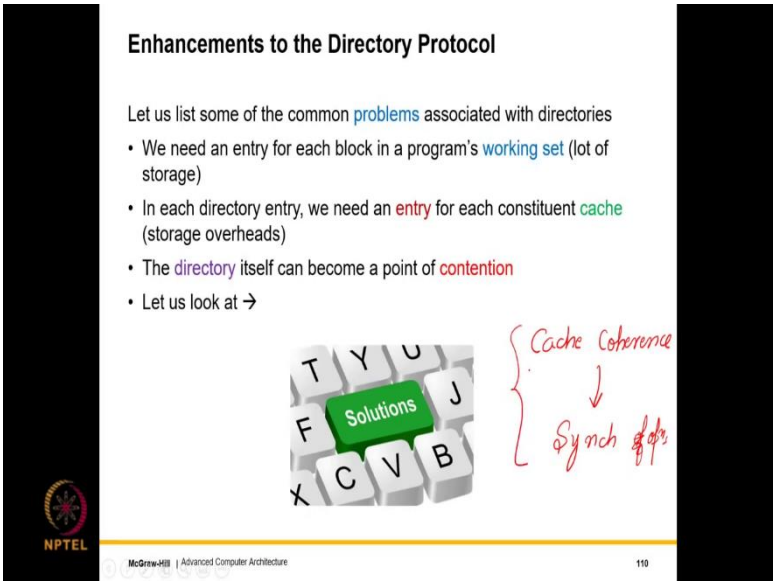
So, let us now look at the state transitions from the E state or from the exclusive state. So, here the state transitions are as follows. So, let us say that we receive an evict in the E

state, then of course, the sharers become empty and we enter the un cache state. So, that is obvious.

If we receive a write miss well, then we still remain in the exclusive state. So, we send a write miss to the sharer the only sharer that has a copy. It forwards it is modified copy to the new to the requesting cache and the set of sharers is set equal to P because only one cache can have write access to the block. If we receive a read miss message well, we do two things: 1st is we send a read miss to the sharer. So, we ask it to forward a copy of the block and sharers plus is + = P.

So, this is very similar to what we had in the basic MSI and MESI, MESI protocols. So, there is no as I said no rocket science, no secret sauce over here it is absolutely plain and simple. It is just that we have a dedicated directory that maintains state per block write in the entire system this of course, will become an overhead, but we will see. So, after discussing the basic protocol let us move on

(Refer Slide Time: 64:43)



So, let us look at some enhancements to the directory protocol. So, let us list some of the common problems associated with the directories and then we will look at the optimizations. So, in the next lecture we will discuss advanced versions of the directory protocol and we will also see that cache coherence is a generic mechanism to implement many other higher order primitives like synchronization operations and so on. And all of

these essentially build or let us say piggy back on the cache coherence protocol. so, this we will discuss in the next lecture.