Parallel Computer Architecture

Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

Week - 05
Lecture - 27

Lec 27: Dragon Protocol

Hello everyone. We are doing module 3 on cache coherence. This is lecture number 9 on the Dragon protocol. So we are finally doing the fourth protocol in my list, four state Dragon protocol and this is different than the others. This is a write back update protocol. If you recollect the top three protocols are, one was write through the first one, then the MSI and MESI were invalidation based write back protocols.

VI was a invalidation based write through protocol whereas this is not invalidation based, this is an update based protocol for write back caches. Okay. Types of messages, processor read and write which you are all familiar with and in case of cache hits no issues but in case of a cache miss. On a PrRd we have to send a BusRd to acquire the data block. On a PrWr, what happens? First is we need the data block and then we are going to do some changes to this data block.

In the earlier protocols, invalidation based protocols on a write we had to invalidate the copies of other caches. But in this one, this being an update based protocol I am not supposed to invalidate others but all the other copies should get the correct data item. So we have to update all the other copies. Okay. So here on a PrWr instead of invalidating others we are not going to send a BusRdx but we send a bus update, BusUpd is the keyword for this, we send a BusUpd transaction which actually carries those words which I have updated. Right. So if we have updated certain words into the cache block only the modified words are sent onto the bus so that other caches having this block will pick up these modified words and update their local copies. Okay. So on a PrWr we have to do this. Then there is this popular flush transaction that if the data block gets evicted and the write is complete then you have to send the data item to the main memory, if the memory does not update itself every time on a bus update or if there is a new sharer into the system and we have the up-to-date data copy then we have to send the data block. Okay.

So data block has to be sent for new sharers into the system. We will start with the four state dragon protocol. What are the four states and where did this protocol originate? Right. So this protocol first originated in the Xerox PARC dragon processor and hence the name dragon protocol. Right. So it is for write back caches and an update based protocol. So we have been seeing invalidation based protocol MSI and MESI.

This is an update based protocol. Okay. So what are the states? We know that there is a state M for modified because you go into state M only then you can change the data item. So that holds for even the dragon protocol. Then we have the state E similar to the MESI protocol exclusive meaning this is the only sharer in the system of this data block and there are no other copies but E does not give you permission to write. Okay. Then we have the S state.

Now here the S state is divided into two names. One is called the shared clean and the second variant is called the shared modified. Okay. We will discuss the details about this as we move on. So exclusive only cache having the data block, memory is up to date. M for modified so we have the ownership, the block is dirty, memory is stale in this case and this is the only cache definitely having the data block.

Then we have the new state shared clean. Okay. Now shared word says that there are other caches having this data block and the word clean says that all the copies are clean meaning the memory and all these are consistently having the same data. Okay. So all the caches having the Sc state have the same data and mostly the memory also has. Some variations say that do not update the memory each time in an update based protocol. So the memory may not be up to date but most of the time it is up to date. Overall the Sc state says that the current cache or the cache in Sc has the most up to date data item.

So when a particular cache is in Sc state there could be other caches having this block in Sc that is they also have a shared clean but they will also be a cache having the Sm state. So I will come back to this when we first understand the Sm state. For the time being you can just do mapping that the Sc is similar to the S state in the MESI protocol. Right. Everybody has a correct read only up to date copy. The state Sm, now the state Sm says that there are potentially two or more caches having the block because of the word shared.

It is shared and modified. So we are using two opposite words rather to construct this new state Sm. Shared word says that there are possibly two or more caches having this block but modified word says that this particular cache has the most up to date data copy and the memory does not have the up to date data copy. So memory is out of date and the memory is updated by the cache which is having this Sm state upon block eviction. So if we are in the Sm state and we evict the data block. Okay, then update the memory.

So memory gets written only when we evict this data block from the Sm state. Okay.

Now Sm that is when we evict update the memory, so we have the up to date copy hence there is exactly one cache or max one cache in the system in the Sm state. You cannot have two caches having the block in Sm. So there could be several Scs in the system. So I can have shared clean, another shared clean in processor P1, P2 and maybe P3 has shared modified. Okay.

So exactly your max one processor will have Sm, others will be in Sc state. It is also possible that there is no Sm and there are only Sc, shared clean states. Okay. So if I go back to the Sc shared clean, you will understand that when we are in the Sc state, this cache is in Sc, you cannot infer about the memory, because if there is another Sm in the system that would eventually update the memory. So being in Sc only says that this cache has the most up to date copy and it can use it, but memory may or may not be there. There may be some Sm which will eventually update the memory, but right now nothing is guaranteed. Okay.

Do we have the state I? Of course there is an implicit invalid state, but we do not explicitly say that there is an invalid state here. Why? Because this is an update based protocol, we do not invalidate others and because we do not invalidate, we cannot force a cache to go in the I state. So there is no formal i state as such. Other caches may have a copy, they may not have a copy. But whenever a tag matches and an updated word is going on to the bus, the copy gets updated. Right. So suppose this cache does not have the data block and the data is going on to the bus, so this cache picks up the value.

Suppose I am transmitting x equal to 2 from this processor. So this x equal to 2 picks up by this one, this one picks up and this does not pick it up because the tag does not match. Okay. So actually you can say that it matches the invalid state in the other protocols, but here we do not explicitly mention this. Okay. So because we do not have an explicit I state, what happens when we start the system? Because the cache will always have some data when we begin or power on the system and some value will be there in the tag and the data. So actually incidentally some tags may match.

So do you say that the block was valid? No. So when you start the system, the Dragon protocols make sure that initially until the cache is loaded with valid data items, we do not go into any of these protocol states and once the states are loaded, that is once the cache blocks are read, that is the cache is full, then we move into the protocol mode. Right. So any garbage value based tag hits will not affect the protocol correctness. Okay. So general description of the Dragon protocol and then we will move towards the FSM design. So when a processor reads a block, it has to send a request, there is a cache miss, we send a bus read transaction and similar to the MESI protocol, we either load the block into the E state if there are no sharers and if there are other sharers, I am going to load it in the Sc, shared clean state. Okay.

So on a read, either we go to E or to Sc. On a processor write, on a processor write, if there are no other sharers and this block is in state E, that is exclusive, we can start writing without informing others or without even sending other bus transactions, simply move to M and start writing and if we are in state shared clean and shared modified, we need to do something else. Okay. So when a processor write, state is E, directly go to M without generating any bus transaction because there are no other sharers. If my current state is Sc, Sc means I am reading, others are reading the data block and I want to modify this data item. So it is an update based protocol, hence we can simply start writing to the block plus send updates onto the bus.

So any words which we change to the block are transmitted onto the bus so that other caches will modify themselves. Okay. Now because this particular cache sent the updates, it becomes its responsibility to hold the ownership of this block. Okay. So therefore, this block changes to the state Sm. Alternatively, when a processor write happens and the state is Sm.Okay. So we do not need to change the state but we have to keep sending the updates on the bus. So this is similar to initially I was in the state Sc and change the value of X equal to 2, so transmit this onto the bus and then change your state to Sm.

When in Sm, keep on changing X or Y, whatever you are changing, keep changing, keep transmitting but stay in the state Sm. Okay. So that is how the Sc and the Sm states interact on a block write. But remember, memory is not updated using any of these updates and which is more practical because we might be sending only few words at a time and we need not bother the memory to keep updating these small words into the cache block. Similar to the MESI protocol, we also need the shared wiredOR signal here because we have the state E. Okay. So if any of these caches have the data block, they will raise the signal, so this will be equal to 1 to decide that there is a sharer in the system.

Hence, we cannot go to the state E, if this is 0, we can go to the state E. Okay. A quick example, right. So let us go to the step 1. Here, P2 does not have the data block, P3 does not have the data block and P1 did not have but P1 initiates the first request. So P1 says, I want to read the data block. Because P2 and P3 do not have it, they will not raise the shared signal.

Hence, where do you think P1 will load the data block in? Yes, you load it in the state E for exclusive, okay. And I am not listing the state for P2 and P3 because it is invalid and I am saying we do not have an explicit I state. So just say that it is invalid in these two, okay. Second event, P2 does a read. When P2 does a read, P2 does not have the block presently, so it sends a request and there is a sharer P1 into the system, so the wired-OR signal will turn true because it turns true, it cannot load in state E, but it loads it in state Sc because I am reading it, it goes to state Sc.

Now when P2 is in Sc, P1 cannot remain in E, so P1 also moves to Sc, nothing changes for P3. Third event, P1 performs a write. P1 is in state Sc, it writes something to that data block, it sends this modified data onto the bus and when P2 sees this that okay, a data item is being modified in this block, it picks up the changes and updates itself, okay. So it locally updates the block here and it remains in the state Sc, it updates itself and remains in Sc, but P1 has recently written to the block and hence it moves to the state Sm, okay. Now you can understand the transition from Sc to Sm.

What about P3? P3 also sees the update that a new value is going for so and so block, but it says oh I do not have the block in any of these four states, so I will ignore the update, alright. The fourth event, P3 wants to write. P3 wants to write, so it will acquire the block, first is it will get the data block and then it wants to start writing, so it sends the updates onto the bus and becomes Sm, so that happens in event number 4, P3 becomes Sm because it is the most recent writer into the system and because P3 moves to Sm, P1 which was in Sm goes back to Sc, okay and P2 is already in Sc. If you can see the difference that although P3 is a new writer, it did not invalidate the copies of P1 and P2 because Dragon is an update based protocol. In every event, there could be multiple Scs, but in every event, there is exactly or max one Sm state, okay.

So now we look at the Dragon transitions, processor read write and bus read are the usual ones. If there is a read miss on the processor, because we do not have an explicit I state, whenever there is a processor read miss, we actually are going to send a request, a bus read request and eventually obtain the data block. If we want to do a write and there is a write miss, again obtain the data block and then start modifying it, okay. So there is something called an I state, but it is not listed in the protocol. When we make the FSM, you will understand that how a transition actually happens from that hidden I state to a valid state in the Dragon protocol.

Then we need a new transition or new transaction called the bus update because this is an update based protocol, the BusUpd is going to carry the most recently written data words into the system and whether we load a data block in E state or Sc state, we need the shared wired-OR signal, okay. So similar to the MESI, I need the wired-OR signal S. We have the bus update as a new transition which broadcasts the modified words, right. If I am changing a few bytes, those written bytes will be broadcast on the interconnect, be it a bus or any other interconnect. And the idea behind only sending this is that we want to limit the data transfer.

Every time we don't want to send the full data block because we don't modify the whole block at a time. We only modify a few words. So only those words will be sent. And when such new words go on to the system, we also need the receiving cache controllers to update their caches, right. So when this x equal to 2 is going on to the bus, this my cache is sitting here, it should have the capability to, okay, this is my block important to me.

So let me pick up this x equal to 2 and update the cache of my system, right. So we update the cache with this incoming block. So we need the cache controller to have this new capability. Now we will start drawing the Dragon FSM. So I request that you draw it with me by pausing the video frequently and trying to do it yourself before you restart the video.

Same convention, black for states, blue for things coming from the processor and red color for things coming from the bus. I am not drawing the state I because it is implicit. Okay, so these are the four states. Now a processor sends a read request and it misses into the cache because the cache does not have this particular block. So if it is a read, then it sends a BusRd, other sharers are there, then go to Sc, other sharers not there go to E, okay.

So here I will say if the processor sends a read, send a BusRd and go to E if we have S bar, okay. There are no sharers go to E, but if there are sharers we have to come to Sc. I will write from here processor read and okay, so actually it is a read miss because even here it is a read miss. The block was not present and we sent a BusRd. In this case the S was true, the shared wired-OR signal was true and so we go to Sc.

On a processor write miss, write miss. So we have to either go to M or to Sm. We will go to M if there are no other sharers similar to the concept of E, right. So here I will say processor write miss and send a BusRd and S is false. Remember we do not have a BusRdX transaction in dragon because it is an update based protocol. On a processor write miss and if there are sharers we will be going to the state Sm.

There are other sharers, processor write miss, other sharers present, so go to state Sm. As soon as we come to Sm we also should send the up to date data because once you get the block you start writing to the block, transmit the changes onto the bus. So I will put a semicolon here and say that I will also send a BusUpd transaction with whatever data items that changed, okay. So we are going to do these two things one after the other.

Now the usual processor read and write hits. When in state E, if a processor read happens nothing to do stay in E. If a processor write happens while in E we cannot change an E but we have to move to M. So this is simple, I will go here on PrWr with no other actions to be done. Once we reach M, for all reads and writes we have a self loop. Whatever are the immediate or intuitive transactions we will finish them before going into complicated ones.

So we have finished all actions related to read write with E and M. So now let us look at state Sc and what happens on a processor write. When the processor writes while in Sc we have to either go to the modified state M or to the state Sm. So let us do this transition on a PrWr. We have to send the bus update that is transmit the words which have changed.

So when you transmit these words other caches will compare this address and see if they have the cache block. If they have the block they will pick up the words, update their local copies but along with this they also affect the wired-OR signal S. Okay. So the wired-OR signal S if it is true it means that there are other sharers and we will go from Sc to Sm but if there are no other sharers we can safely go to M which gives us more feasibility to continue read write without informing others. Right. So on a processor write send a bus update and in response to this if the wired-OR signal is false we can go to M. All right. Now Sm, in Sm reads are simple so self loop on a read and on a write.

Now what happens on a write? On a write if there are other sharers we can remain in Sm but if there are no other sharers we can go to M. So processor write, send a bus update other sharers present S is true so remain in Sm. And if other sharers absent go to M. Here I will write processor write send a bus update and here S is false.

So we have s bar if S bar go to M. So with this we have finished all the read and write transitions from the processor side we will use the red color pen and finish the transitions on the bus side. We will start with state E. In a state E, if we see a bus read transaction it means there is a new reader into the system we have to move to state Sc shared clean because a new reader has come. That is on a bus read actually no action to be taken because the memory will give the data. In state Sc if a bus read happens nothing to do memory will give the data we only have to take care when an update comes onto the bus. Right.

So when in Sc if an update comes bus update that is some other cache is sending a few modified words for this particular block we have to pick those words up. Right. So take those words and update your copy and stay in the same state. So we have a self loop on this on a bus update. In Sm, we are in Sm and we see a bus update. Okay. So we have to change our copy but now the most recent writer is not us but somebody else. Right.

That particular cache will go to Sm and this cache has to go to Sc. So from this we go to Sc on a bus update. On a bus update update your local copy and move to the state Sm. While in state M a bus read, bus read says that there is a new sharer into the system similar to the thing we did for state E. But this cache has the most up to date copy, so it is responsible for giving the data. So once it gives the data it can retain a copy and go to the state Sm because this was the most recent writer of that block.

On a bus read go to Sm. While in Sm, if a bus read happens. A new reader has come into the system so give the data but do not change state. So we have a self loop. Okay. So if we cross check we have done all the required transitions. The bus read, bus update and in the case of E you are not going to get a bus update because before the bus update comes there would have been a bus read which will move you out from the state E and take you to either Sc state or the Sm state eventually. Right.

So when a bus update comes we will not be in the state E. Okay. So that is how we worked out the FSM and a neater diagram is given here on this slide. Now although we have verbally discussed about all these state transitions I want to go in detail for each of them because this is an update based protocol and there are some intricacies I am going to do these four types of actions or transitions which happen in detail. Okay. So first is the read miss then we are going to look at a write hit, a write miss and a replacement. Okay, read miss. So on the read miss we are going to send a BusRd transaction and we will either go to the state e exclusive if there are no other sharers or we will go to Sc depending on the shared wired-OR signal.

If there is another cache with the block in M or Sm then that cache is going to give me the block. Okay, so a bus read has gone so who gives me the data? Either the memory gives the data or another cache having the most up to date copy that is in the state M or Sm gives us the data. If all the other caches are in Sc means the memory is also up to date so memory gives the data. So, go to E or S depending on the wired-OR signal if other cache has the block in M then that cache provides the data.

If everybody is in state Sc then memory gives us the data. Okay, right. So this cache has sent a read and there is already somebody else having the data block. So this one gives the data and this changes itself to Sm. Right. So this will change state to Sm. If there are other caches either in shared clean or shared modified and the current cache wants to read, then what will happen? This Sm will give the data and not the memory.

Okay, and it will remain Sm. The third case is when there is no Sm but there are only

Sc blocks in the system shared clean. So in case of shared clean none of them takes the responsibility to give the data hence memory gives the data to the new reader. Right. So this is the new reader which has come into the system and everybody remains in the same Sc state. Okay. So whenever you have a M or a Sm this cache will give the data if everybody is in Sc memory gives the data.

So that happens on a read miss. So what happens now on a write hit? Write hit if we are in the M state no action to take, if we are in the E state first go to M and then start writing without informing others. If the local state is Sc or Sm, Sc or Sm simply update the data but send the contents onto the bus using a bus update transaction. After this either move to Sm or M you have to go to a modified state because you have changed the data item but we will go to M if there are no other sharers, we will go to Sm if there are other sharers. Remember memory is not updated. This is important. Okay. So if this new cache wants to write, it has the data block it wants to write it will go to M provided others are in state I. Alright.

If others are not in state I but there is an Sm then this Sm first gives the data then this Sm has to change to Sc and current cache will change to Sm. Because now this cache the new yellow cache will be the most recent writer into the system. Okay. In the first case here I can write that memory gives data. Okay. What happens on a write miss? Write miss is first treated as a read miss followed by a write. Okay. So first we incur a read miss that is we send a pass read transaction, acquire the data block and then after you acquire the data block you would move to either the E state or the Sc state and then you will follow it by a write request. So, once you want to write from Sc or E, right, this is what is going to happen you will either go to here or here and then you will move out from there using a bus update.

So I will illustrate it. We do not have the block so we are in this hypothetical invalid state and then we want to write but first we generate a read miss. After the read miss if there are no other sharers, we will first become E and then we do all the actions which happen on a write hit and then move to M that is one path. If there are other sharers into the system, we will not go to E but we will take this path, become Sc then again execute the write hit related actions and move to Sm. So this happens on a write miss.

So write miss is equal to a read miss followed by a write hit. What happens on a block replacement? We have not shown the arcs in the FSM but whenever a block is replaced that is evicted, if we are in the state M or Sm it is our responsibility to update the memory. So update the memory only here but if the block is in Sc, then you need not update the memory because there is some other block in Sm who will update the memory or maybe there are no other blocks in the system so memory is already up to date. Okay. So if my block is in Sc probably somebody else will update the memory or the memory is already up to date. So replacement is simple, if the block is in M or Sm only then write otherwise do not do anything. The same cache coherence example. So we will see how this is implemented in the Dragon Protocol.

So using Dragon Protocol I am going to fill this example table for our cache coherence example. On the first column I have written all the actions which were happening in the example and then we are going to fill this state in P1, P2, P3 bus action and who supplies the data. First one, P1 reads u when P1 reads u nobody has the data block so P1 goes to the state E and what is it does is send a bus read transaction, memory supplies the data and go to state E, they don't have the block. P3 reads u, because P3 reads u, it will send a bus read, memory gives the data, P3 won't go to e it goes to Sc because there is P1 sharing and P1 also becomes Sc. So there is a change here. Okay. Now P3 writes u, when P3 writes u it will first incur a miss and then a write hit so eventually we will get a bus update transaction coming from here and when the bus update happens this moves to Sm, this goes to Sc and for this initially the data comes from the memory but later it becomes the supplier of data.

So it gets the data from memory but when you do bus update P3 supplies the data onto the bus so we can write both these things inside this column. Then we have P1 reads u, so when P1 reads u, it already has the up to date copy so no change, no action here, no action here, this remains Sm, no change here. Last P2 reads, when P2 reads P2 doesn't have the block, so it sends a bus read. Who provides the data? In this case P3 is cache because P3 has the block in the Sm state, so P3 provides P3 remains in Sm and other two go to Sc. Okay. So this is how we can implement the dragon protocol for this example. A neater version is kept here so here I will just say that first the memory gives and then P3 will become the giver of the data for the bus update.

Now about coherence, dragon satisfies coherence. Again the proofs are very similar to the write through case, write through case was the VI protocol which we discussed. So proof is same as the write through case. All writes appear on the bus in an update based protocol. So write through and update are similar. Every change is going on to the bus, hence the bus serializes everything and therefore write serialization, write completeness, write propagation, write atomicity, everything is very straight forward with if we have an atomic bus hence dragon satisfies coherence using the strategies we have developed till now for the other protocols. So the proofs are straight forward and similar. Okay. So now we just need to look at smaller lower level design choices related to the dragon protocol. The question is can shared modified state that is Sm state be eliminated? Can I do without the Sm state? and this is done by the Dec Firefly multiprocessor. So I would request you pause the video, think the answer and the reason before moving on.

Okay,right. So for this if I want to eliminate the Sm state, look here, I have three caches in these one two three one of them is Sm and a new reader comes into the system and a new reader comes, the giver of the data is the Sm. Right. So if you recollect Sm is going to give data to this new reader of the system and that is the reason I had the Sm because Sm takes the responsibility of giving the data. So in this case, I need Sm. If there is no Sm in the system then memory gives the data. So if I want to eliminate the Sm state then I have to make sure that every time the memory gives the data. Because in case Sm is present Sm has to give the data. If I want to eliminate Sm then memory has to give the data. So how do I manage this so to do this? The easy solution is do not have the Sm state that is, whenever a write is done, also update the main memory. Okay. So if you update the main memory then we would not have the Sm everybody will be in the Sc state. Okay. So I can do this provided the memory also updates on every bus update transaction. Okay. So underlying assumptions for keeping the Sm. So I kept the Sm here, because I assume that the cache that is SRAM will give the data quickly than the main memory so keeping Sm is good because the cache is faster in giving the data but if you think that the DRAM is also fast enough and you don't want to disturb the cache having the Sm state then do not keep Sm. Okay.

So these are design choices available to you. So depending on your requirement you can choose either option. Okay. So I hope it is clear if you want to remove Sm on every bus update you have to update the main memory. The second question is should replacement of an Sc be broadcast when I am in Sc? Should I tell others I have removed the block? So this is a scenario. I have four caches one of them is Sm and others are Sc. Slowly what happens is, this block gets evicted, it does not inform anybody else, it simply deletes the block without informing. Right. And eventually this one also removes so we have just one block left in the Sm state. Okay. So now when in Sm state what would happen? Any write which happen in this state it is going to send the bus update transaction. Because it thinks because I am in Sm there are other sharers and they would need the data, so it will keep on sending these updates onto the system. Whereas if I had known that other Scs have disappeared I could have safely moved to the M state and this will not generate the bus update transactions.

So this is a good property of this transition provided I know that there are no Sc states. Okay. A similar example happens here when I had four readers then. So I had initially four sharers, then two sharers, and a single sharer. So I have a single sharer here. Now any changes done by this again would result into bus updates. But if I had known that I am the single sharer in the system we could simply move to the state E, which is more ownership giving state than in the Sc state. So I hope you understood the importance of telling that every time the Sc gets deleted, it should inform everybody. If it informs then I can possibly do this. If it does not inform, we will remain here and keep on sending the bus updates.

Okay. So we will keep on sending the bus updates if Sc is not broadcasted. So this allows us to either go to E or M. We have seen the example. And why is this good ? Because telling that I am deleting myself is not in the critical path. You just say I have deleted myself. But the transactions that it saves later, right, it is going to save the transactions of bus updates and these might be on the critical path to affect the performance. Right. So bus updates are on the critical path because when a processor changes it sends the bus update transactions and until they are finished that processor cannot proceed. So it is going to slow down. Hence, I want that all these bus updates do not take place unnecessarily. Okay. So to avoid future bus updates it is always desirable that we broadcast the replacement of Sc, so that whenever a particular cache is the single copy across the system, it can move to the state E or to the state M. Okay. So with this we have seen lower level design choices related to Dragon protocol and overall how it is designed. It is a bus update based protocol and slightly different from the MESI protocol. So with this we finish the Dragon protocol. Thank you so much.