Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 10
Lecture - 52

Lec 52: Working of protocol (2)

Hello everyone. We are doing module 6 on directory coherence case studies. Continuing the previous lecture, we are discussing working of the protocol. In this lecture, we are going to look at writes and writebacks. Handling the write miss. So, write miss is a request for writing the data block. The rule in the directory protocol is that if node wants to get writing permission, it has to be the head node.

Any node in the distributed linked list cannot do the writing, only the head node can do the write. So, the requester has to first become the head node of the distributed linked list. One option is, it is already the head node, so it can start writing, but before that it has to make sure the other nodes in the distributed linked list are invalidated. So, we need to purge the complete list, become the only node in the system, then modify.

When a node changes the data, the status of the directory is gone. It is not home, it is not fresh, the directory status becomes gone because there is a dirty copy elsewhere in the system. So write miss is handled this way, we will do it in detail now. Okay. So the requester has five different possibilities of its location in the distributed linked list. One is that, it is not there in the sharing list.

If it is not there, it has to get added as the head. It is there in the sharing list, but not the head, so it is in the middle somewhere. So if it is in the middle, it has to remove itself from the middle position, attach itself as the head and only then it can write, that is the second one. Third is that it is already a head node, but when it is the head node, there are possibilities that the directory state is either fresh or gone. Accordingly, the head could be the fresh or gone.

And the fifth case is it is the only node and it has the dirty status. Okay. So these are the five conditions, we will do them slowly one by one. So possibilities with the requester. It is the easiest one is this only-dirty. The requester is the only node in the system. So the home memory is pointing to the requester, status of the requester is only-dirty.

Only-dirty means what is the state of the home, it is in the gone state. So this is the perfect situation when we can start writing immediately. Now second case is this requester is the head node and luckily it has the dirty status. This dirty status implies that the home is in the gone state. When home is in the gone state, we do not have to inform the home because it already knows that the dirty block is elsewhere.

Hence we can start writing immediately. But prior to that invalidate all the sharers. Because this node is the head node, it essentially says, there is a list which is existing. First we have to purge that list, become the only dirty node and then start writing. So when we are in condition 2, we have to move to condition 1 to start writing.

From head-dirty, move to only-dirty, only then write. Okay. Third case, head-fresh. So now we are head-fresh. So we cannot start writing because first thing we are head so we need to invalidate the sharers, become only node. Second condition is we are fresh node.

Hence the directory state is also fresh. So first we have to inform the directory to change from fresh to gone, only then we can start writing. So we first go to the home node, change the state of the home node to gone. Once this is done, we get the status of a head-dirty node. Once you get head-dirty, you know what to do.

So once we are in state number 3, we go to status 2, then to status 1 and then start writing. So head-fresh becomes head-dirty, head-dirty becomes only-dirty, then it can start writing. All right, fourth case. The requester is not there in the sharing list. Once it is not there in the sharing list, we have to allocate an entry in the remote access cache for that particular quad.

Then go to the home node. Home node will tell us about the head of the list. We have to attach this new requester to the head node and then become the new head. Once you become the new head, end of 4, what is the option of end of 4? You could be either head-dirty or you could be head-fresh. Right. So from 4, you could be either in state 3 or state 2, accordingly we have to take the action.

Fifth condition is requester is in the sharing list but not the head node. So once we are not the head node, we have to remove ourself from the list using the roll out operation by communicating with the neighbors using request response. Once we move out of the list, we have to add ourself as the head node and then start the above operations. So if we are in state 5, we have to roll out and go to state 4 and from 4, you can either go to 2 or 3 and then follow the process. So this is the list of all the operations.

We will again do them slowly. So once the node is in head-dirty state, we have to purge the list. So how does purge work? Purge is working in a serialized fashion because we cannot delete the full list in one go. We can delete one node at a time. So the head communicates to the next node.

The next node deletes itself, then the head connects to the node after it, then that node gets deleted and so on. So one by one, the nodes will get deleted. So we send the invalidation request to the next node which rolls itself out, gives the idea of the next pointer. Now the head points to this one, so it will delete the next node and so on. While doing all of this, the head is going to remain in a pending state till the purging completes.

So therefore, any new attempts to add a new node to this list will go into the pending list. See now the head is in the pending list because it is busy purging the nodes. In case a new request for the same block goes to the directory, that block will get added as a pending head to the same list. Okay. Now you would say that why should I delete the nodes one by one? Can't we use an optimized strategy that the node which deletes will also forward the request to the next node in the system. So can I voluntarily send invalidation to the next node and so the next node will directly add to the previous node.

So I have this node. Instead of deleting one by one, can this node forward the request on my behalf and then this one directly acknowledges to the originator node. We could do this, but what happens is, this is going to distribute the state of invalidation. So we don't know whether the invalidation has finished in one go. There will be multiple transactions before that one invalidation finishes. So we don't want to distribute the state of invalidation because we already have a distributed network.

All the blocks are scattered in the system and hence to safely manage other complications we follow a strict request response type of a protocol. So we do not do this network based optimization in the SCI standard. Okay. So we have discussed this. Quick example of purging the list. So node is at the head and if the home state is gone then you simply send an invalidation message. So this node comes out and you connect to the next node.

So this way you will keep on removing all the nodes. Our objective is to start writing and we can only write when we become the only-dirty node. Until then you can't start writing. So right now we are still in the head-dirty state because we have one, two more nodes in the system. So we need to delete even them before we can start writing. Okay.

So eventually the tail node will be there. So we will also remove it and then we will get the status of only dirty node. Why only dirty? Because home is in the gone state. If home was in the fresh state, our status will be only-fresh. Right now home is in the gone state.

Our requester is only dirty and hence we can start writing. So purge the list, become only-dirty, start writing. Okay. Second case is head is in the fresh state. Previous example was head was in the dirty state. So head is in the fresh state because home is in the fresh state.

To start writing you have to go from head-fresh to only-dirty. On the way you may go through a head-dirty state or maybe directly. So right now we are in head-fresh. So head goes to the home with a write request. So home says okay, I will go into the gone state.

So home becomes gone. Home from home-fresh becomes home-gone. Then it gives the status of head-dirty to our head node. So home becomes gone, head becomes dirty from head-fresh and now what do we have? This bottom list looks similar to my previous slide where we had a head-dirty and few more nodes. What do you do after this? You are going to purge the list. Once you purge this list what will happen? The head-dirty node will become only-dirty.

Subsequently you can start writing. So you purge the list and start writing. Okay. Considering a race condition in the current operation, see here if the head node goes to the home node saying that I want to write, in normal scenario home will say okay, you start writing I will go in the gone state you become head-dirty purge the list and write. But the situation I am considering is, head-fresh goes to the home but home is no longer fresh or home is no longer pointing to this requester as the head node because some other request has already reached the home node. Okay. So head-fresh goes to the home to request changing the state to gone. So we have gone there saying that okay, please go to the gone state. I want to start writing but it finds that either the home is no longer fresh, probably something else has happened in the middle or it is pointing to a newly queued up node.

So home says that I am anyway not pointing to you when this one when this reaches home says, I am not pointing to you as the head node but I am pointing to somebody else. So what should home do? Home simply NACKs this request because it says I cannot cater to your request because you are not the head node. So home sends something like a NACK to the writer which is the head-fresh node we are considering. When the writer receives this request what should it do? It will have to wait because in anticipation that there will be another node requesting to link itself. Okay. So writer will receive this request from the new node eventually to link it.

So the new node links with the current writer. End of this what will happen? home is pointing to some new node. This new node is pointing to our writer, so writer is no longer the head node. So the next request it has to roll itself out from the list become another head once more and then start to attempt the write operation. Okay. So let us see this as an example. This is my current scenario: head is in the fresh state and head sends a request to the home. Home says I am not fresh or I am already pointing to somebody else as the head node.

So home says no, you are not the head, so the brown one is not the head but the pink one is the head. So a request from the brown node asking permission for write cannot be handled by the home. Hence home sends a NACK to the brown node. So the brown node can keep on trying this request but in the meanwhile, the new requester goes to the brown node saying that I am the new head please attach me. So the pink node goes to the brown node with an attach request. What will happen? We will get the list looking like this where the pink node becomes the new head, the previous head becomes the mid-valid node. Okay.

So now the brown color node wanted to write but it is a mid-valid node. What should it do? It cannot start writing here and hence it has to delete itself from the list, contact the home once again, become the head and then restart the operation. So it will roll out, this node rolls itself out, it will contact the home node, home node will make it the head in future and then you know the remaining process. Okay. So write request is, just to recap, you have five different cases to consider. A write can only happen if the node is the only dirty node a single node in the system. So you can start writing in the only-dirty state. In case we are head-dirty, that is the node at head is termed dirty but there are other nodes in the system. So we need to invalidate those nodes, once we invalidate, we will move from head-dirty to only-dirty then start writing.

Third case is we are head-fresh so first tell the home to go to gone state, then you become head-dirty from head-dirty, become only-dirty, then start writing. Fourth is we are not in the sharing list, so go to the home, become the head. Once you become the head, you will either be head-fresh or head-dirty and then follow the same process. Fifth state is, we are in the sharing list but not the head. So you remove yourself from the list, become the head and then from step number 4 or case 4 you can follow the path 4, 3, 2, 1 or 4, 2, 1. Alright. So this is how writes were handled. Third operation is handling write backs and replacements.

So replacement is when the block gets deleted from the cache. Now what is the cache we are discussing here? The block is allocated space in the remote access cache within the quad and we want to remove this block either because it is getting invalidated, that is somebody is purging the list or we want space for this block due to conflict or capacity misses within the quad. Okay. So there are multiple reasons that I want to delete this block and in case the block is dirty, that is we are the head dirty or the only dirty node. It is our responsibility to update the main memory. So we need to do a write back. Okay. So we will see how this is done. Okay. So node is in the sharing list and it wants to delete itself for the following three reasons.

One is that it is being invalidated because somebody else is purging the list. It needs to be replaced in the RAC because of capacity or conflict reasons. RAC is full and you want to delete a block in the RAC. Hence this block has to be evicted. The third reason is the roll out because this particular node wants to pick up the head, so it removes itself from the distributed linked list and then becomes the head node. So it wants to become the head and therefore it rolls itself out from the distributed linked list.

So in variety of reasons you would want to replace the data block. So a block in the sharing list wants to delete itself from the list. Now you would say that if the block is replaced, it has to be removed from the sharing list. Can it still hold the capacity or space in the RAC? So should I allow it to stay in the RAC and simply remove the pointers? Okay, so the block is there but the forward backward pointer are removed. So well, we cannot permit this because within the RAC, when you have the data block, we have the data and the forward and the backward pointers stored in the RAC.

If I am going to replace this block this particular location will be taken up by another data block and that data block will again need these pointers. So we need the complete space in the RAC for another block and hence we cannot retain the data. When we delete it we have to remove the entry as well. So the space it occupies in the RAC has to be freed for the new data block because the new data and the associated pointers would need this emptied space. Alright. And in case your data is dirty, that is you are at the head position either in head-dirty or in only-dirty state, we have to update the memory with the data and delete ourselves from the sharing list.

Okay, replacement of a middle node. So when a node in the middle of the distributed linked list wants to delete itself, it has to first go in a pending state. That is the normal strategy applied by the SCI protocol that every time it goes in a different types of pending states before all the actions can be completed. Because as you understand, this network is large and it is going to take time to finish all the possible actions, hence going in the pending state is safest. Alright. So this node goes in the pending state, it sends a request to the neighbors to update their pointers because we are going to roll out our self from the list and when we contact our neighbors maybe the neighbor is also wanting to remove itself from the list. So this is the current node. It is going to contact this, okay, if I call them node 1, 2 and 3. If 2 wants to delete, it will tell 1 and 3 to connect to each other but it may happen that node 3 is also trying to delete itself, and it has sent a request to 2, that you remove me and connect to 4. Okay. If both of these nodes simultaneously want to remove themselves then what do we do? We have a race condition here and to solve this race condition, the easy solution is to give priority to one of them and hence we give priority to the tail side node.

So in case 2 and 3 both want to delete themselves at the same time, we give priority to node number 3 which is on the tail side. So once we delete the node, it is marked as invalid in the cache and other nodes in the system have to update their state. Update their state because if your position in the list changes because of this deletion, we will have to change, the head's position might change, the tail's position might change, okay. So if there are only 2 nodes in the system then you will be the only node left because if 1 and 2 are there and 2 deletes, only 1 is left, if 1 and 2 are there and 1 deletes then only 2 is left, okay. So we have to accordingly update the status to an only node if I have a 2 node system , okay.

So entry is dirty, so we have to write back if entry is clean, simply replace the block without writing back to the main memory. So head will put itself in the pending state and messages will go to the downstream node. See the previous case, we were considering replacement of the block from the middle node. So middle node is not the head node and if you are not the head node you do not have to update the data with the home because the head node will take care of it. So you simply delete yourself and the other nodes will manage their status to become mid or tail nodes.

But if you are the head node which is getting removed, then we need to take care. So head, if this is the head node, it puts itself in the pending state and sends message to the downstream nodes. The downstream node has to now point to the home node because head is going out, so the node after the head will now become the new head of the system. So this new head has to connect to the home node.

So home has to now point to this next node. Okay. Let me say this is 1, 2 and 3. This is the home node and 1 wants to delete itself. So when 1 wants to delete, it will say, it will contact 2 and tell 2 to point to H and it will tell H to point to 2. So this is what it has to do. And in the meanwhile it goes into a pending state. Now this node after the head node, the node 2 or 3 depending on its position, it will have a new state. Suppose we had only 2 nodes, node 1 and 2 then when 1 removes, only 2 is left and now 2 will become the head-dirty node, if it was a tail-valid node.

If it was a mid-valid node, it will become a head-dirty node. If it was a mid-fresh node, it will become a head-fresh node. So in case you have the second term as fresh you maintain that term. If the second term is valid you have to change it to dirty. Okay. So depending on the state of the original head, the terms of the remaining nodes will change. Okay.

We will do an example. So roll out of a head node, this is the list, home is in gone state, head is dirty, mid-valid and tail-valid. So these things say that, the data is valid memory data is in the gone state. Now head wants to delete itself. So it sends a message to the mid node saying connect to the home. So mid node connects to the home node, then head node also tells the home node to connect to the new head.

So this way the connection between the green and the peach happens and the brown color head invalidates itself. So this is how the new list would look like. So this is the deleted node and the new list has got home gone and see this mid-valid has become head-dirty. Essentially it inherits the status from the previous head.

Okay. So this slide tells the same things which we discussed in the previous one. So what happens at the home node? The home node will then update the pointer to the new head and change the state accordingly. After receiving the reply from the home node, the replacer will set its block to invalid state, like here. Once the connections are established, it will go into an invalid state. If the replacer is the only node, then it has to only communicate with the home node which is straight forward.

It is the only node, so it communicates with the home node. And in all this, if the block is dirty the memory is updated before removing itself. All right. So here we are going to look at one more race condition. So if the replacer head node reaches the home node and the home node has changed the state or it is pointing to somebody else, then it is going to send us a NACK similar to a previous race condition we discussed. So eventually this new node which was trying to become the new head will reach the current head will link itself and then the current head has to roll itself out.

So we have this case of head node mid and tail. We have the home node it says connect to this. But when the brown node reaches the home node, it says that, now you are no longer the head node but I have a new head so this will send a NACK. Once the NACK reaches, head node may retry, in the meanwhile, the pink node contacts the head node to attach itself and we get a list which looks like this. Okay. So we do one optimization here that because this head nodes that it wants to delete itself from the list, it knows that the pink node has come as a new head, so it takes this opportunity instead of remaining in the list and rolling itself out later, it removes itself right now. So once the pink node comes it directly attaches the pink node to the middle node and removes itself.

So it deletes itself from the list and attaches the new head to the next node. So this is the shortcut it takes in this particular race condition. Okay, now coming back to the final point to serve a write miss we need to replace the old block and load the new block. This is what we do in a write miss. First we have to write something back and load the new data block. In a normal bus based system, what do we do? The block I am deleting, will be put in a write buffer, so that we have space in the cache. We load a new block and start using it and the write back can happen in the background in a bus based system.

But we can't do this here, because when I want to write back a data block or delete a data block, we have to roll ourselves out from the list. This has to be done because the space I am occupying in the RAC with the pointers has to be emptied before the new data can come .Because we don't have the concept of a write buffer here. Right. So we don't have a separate write buffer. The RAC itself is the location in which the data is kept. So we have to first free this location by completing the write back or the replacement operation only, then the new data can be loaded. So with this we have looked at write back and replacement and associated race conditions. This is how the complete SCI directory protocol works. Thank you so much.