Parallel Computer Architecture

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

Week - 09
Lecture - 51

Lec 51: Working of protocol(1)

Hello everyone. In this lecture, we are doing module 6 on directory coherence case studies. This is lecture number 5, in which we are going to discuss the working of the protocol and currently we are discussing the Sequent NUMA-Q cache based directory protocol. Okay, so what all things are we going to see in this particular topic of working of protocol? We are going to look at the primitive operations on the list because a cache based directory protocol is nothing but a distributed linked list where every node in the list is the cache block and it is pointing to forward and backward both directions. The home node is pointing to the head node of the list and the head points to the next and so on up to the tail. So we have to construct this distributed linked list.

So we are going to see what are the different types of operations I can do on the linked list. Once we understand that, then we will look at how to handle a read miss, subsequently how to handle a write miss, write backs and replacements. Okay, so to begin, we will see a quick recap of how the NUMA-Q system looks like and then we will start with the primitive operations. So we have a SCI ring, this is the SCI ring, which is a unidirectional link on which the quads are connected.

Every quad is made up of 4 Pentium Pro processors which are connected over the quad bus. They have their own slice of the main memory and the quad is connected to the ring using the IQ-Link. Okay, and in this example, I am going to take, there are 6 quads in my example with these colors denoting, so if the color is blue, it means this quad yellow would be another quad just to get an understanding of some examples. Okay. So these colors would represent data coming from different quads. Alright, so to understand how the data sharing would take place in a cache based directory. This picture is indicating that a single quad which has got 4 processors P1, 2, 3 and 4, a slice of the memory and we have the remote access cache and the IQ-Link board.

Alright, so if you can recollect from the previous lecture, the IQ-Link board, this one was made up of several components, it had the bus controller, the directory controller, network interface, it had the remote access cache and the directory storage and the controller. Okay, so we have the remote access cache and the IQ-Link board, so this is the interface to the SCI ring and here, up to here, you have the quad bus that is the symmetric multiprocessor of the shelf component which we are using. Okay, and now this one is depicting the L2 cache of the processor P1 and the different colors are indicating the blocks coming from various quads. So the blue color is the block coming from the local memory, so it comes from here. Okay, so that is a local memory block cached by P1 and when a processor caches a local memory block, the remote access cache has nothing to do with it. It will not keep any information about it and the coherence for such locally allocated blocks and locally cached blocks will be done by the quad bus's snooping protocol.

So the snooping protocol here will manage the coherence for this. Okay, so some more allocations are shown here, so the blue ones are the local blocks and the other colors come from outside. So all the blocks which are not blue in color, for example, this green, the pink and the yellow, they are coming from another quad. Now, who will keep track of the coherence of this? So the remote access cache is going to keep a copy of these caches or these blocks. So the yellow, the pink one, so this is one block of the pink quad, this is the second block, so this could be the one block, second block, the green one and the yellow one.

So this way, the blocks which are not blue in color are cached by the RAC and RAC will manage to establish a coherence for them. Okay. So the SCI protocol is not going to see P1 to P4, it is only going to see the RAC. So all the blocks which are there in the RAC will be seen by the SCI protocol and that directory coherence will be maintained only for these blocks. So directory coherence in the current example will not be maintained for the blue color blocks but it will be only maintained for the yellow, pink and green blocks. Okay. So blue color are locally allocated blocks and local snooping coherence will be used for them and the other colors are remote blocks coming from other quads and RAC will keep track of their coherence.

So with this understanding of how the blocks are present, we are going to look at primitive operations on the list. So first is list construction. List is constructed by first constructing the head node. So the home node will point first to the head node and then subsequent blocks can be attached to this list. So list construction is, first create the head node.

When you want to add a new node, normally in a linked list program which you would write in any programming language, you can decide to add a new node at the, either at the tail node or at the head node whereas in the SCI directory protocol, it is always added at the head node because when a request reaches a home node, it will create a node in the linked list. Home points to this new node and this new node will then point to the previous head node. So new nodes only get added at the head. So that is the list construction operation. The second is a delete operation which is called a roll out where a node comes out of the linked list.

When you delete a node from a linked list, what do you have to do? The previous node in the list should connect to the next node in your list. Okay. So a node which is deleting itself has to communicate with the left and right neighbors and make them contact each other and only then this node can come out of the linked list. So that is rolling out of a single node. The third operation is purging. Purging the list is deleting the complete list and when are the occasions that I would want to do this? So when a node wants to do a write to a data block, it is the only writer in the system and other sharers have to be deleted.

All the sharers are in the linked list and so the head node which gets the permission to write will have to delete all the other nodes in the linked list. So to do this, it uses the operation of purge. Okay. So we are going to see these three operations in more detail. Okay. So now we are going to see how NUMA-Q will construct the distributed linked list. In this example, we are considering four quads which would prospectively share a data block.

The quad which is shown yellow in color, it requests a data block from this particular memory. So it comes to this memory, memory gives the data block and establishes a pointer from the memory to the quad. Okay. So from the home node, a pointer is established to the yellow color quad making it the head node. So this is the head of the list. Only one node is there in the linked list.

Subsequently, the pink color quad also requests for the same data block. So this pink quad goes again to the home memory. So this is the home node. So it comes here and it has to get linked to that particular list. Okay. So overall I will have a linked list constructed from the home across all these quads. Okay.

So home will point to the yellow, to the pink, to the green and so on. So this way a linked list looks like. The construction of this will be done slowly in the future examples, but overall you will get an idea how a distributed linked list looks like. So we have a home node, three quads and then the last quad points to the null and all these links are bidirectional. Okay. So understanding primitive operations on a given list.

So list construction is the simplest one. We have to add a new node at the head node. So that is the home memory. Request comes from a particular quad which wants to read that data block. So it will come here, access the home node.

Home node will create the first node in the linked list and make that node as the head node and the head node points to null. So this is how list gets constructed. So list has begun getting constructed. We can keep on adding nodes to it. So in case a new node wants to get added itself, it will come to the home node.

Home node says okay, now you become the head node and now this new node will then connect to the previous head. So this way nodes will keep getting added. Okay. Then the next operation is roll out. So roll out is when I want to remove a node from an existing list. So I have shown an existing list here with home memory, the head node, a mid node and a tail node.

And suppose this mid node wants to roll out itself. So what should it do? It has to send a message to the previous and the next node. So it will send a message to the previous and the next node and remove itself. Okay so list construction takes place in the following manner. The new node always becomes the head node.

So if the pink node has come, it becomes the head node. Subsequently if this particular node comes, it will become the head node. So this node will become the head node. Using this philosophy, if I go back to this particular example, here you can see that because the list consists of the home, then the yellow, then the pink, then the green, so I have home node which is pointing to the yellow node, yellow is pointing to the pink node and pink is pointing to the green node. Right. So if you look at the end of this construction, we can conclude that the green node was the first node to approach the home, then the pink node came and then the yellow node came. Because the newest requester always gets added as the head node.

Currently the head is the yellow node and whenever a new node comes, it first goes to the home node and home pushes this new node as the new head of the linked list. So this is how the list gets constructed. Considering the next operation which is the roll out, here I have already taken an existing list consisting of a head node, a mid node and a tail node. If the mid node wants to come out from this list, it has to communicate with its neighbors in both directions, so that these two neighbors can connect to each other and the mid node can safely come out. Okay. So this is how the mid node comes out and the head and the tail get connected to each other. Okay.

Third operation is the purge operation. Purge is nothing but invalidation operation where the head node is going to start invalidating all the other nodes in the linked list. So only the head can issue a purge. So what should head do? Head has to send an invalidation message to the node next to it.

Right now, it is the mid node. So it sends an invalidation. It receives an acknowledgement that the node has deleted. In addition to this, it also gets the sharer ID of the next node. Because head would not know what node was connected here. So this mid node gives the identity of this sharer to the head node. So once it gets this, what will happen? Head will then go to the next node to delete that node. Okay.

So that mid node has gone out. So this was deleted and now head is connected to the next mid. So we will continue this operation until both the mids are deleted, then the tail remains. Eventually the tail will also get deleted and what you get is the home memory pointing to the head which is the single node in the system. So you will only have a single node in the linked list, so that you can start writing. Because the purge operation is called only when a node wants to write. Okay. So that is how the three operations are implemented in a distributed linked list.

Next we are going to look at how to handle a read miss. Okay. So read miss processor sends a cache request for reading. It misses in its cache and so it has to go somewhere to get serviced. In my current setup, I have a quad in which there were four processors. This cache miss emerges from the L2 cache of a processor and then it will reach to the local memory.

If the local memory within the quad can satisfy it, then as you understand, each will be managed by the bus snooping protocol and the remote access cache and the directory will have no role to play and you already know how to handle this using bus snooping. So I am not going to consider this case. Okay. So we are taking a case where the directory protocol will get invoked. Hence the cache miss is on a block which is not locally allocated. This data block will come from another quad. Okay.

So this request is now going to the directory controller sitting near the remote access cache. Okay. So SCLIC is the directory controller, the RAC is attached to it. This request comes to the directory controller. Directory controller now has to see if it can satisfy this request locally. That is, in case it has already brought this block into the particular quad from outside, it can satisfy.

Otherwise it has to go to the home node. So let us say it decides to go to the home node because that block is locally not available with RAC. Okay. Neither RAC has, definitely not the processors have it. So this request goes on to the SCI ring to the quad which houses this particular data block. Now when it, when you reach to the directory, the directory could be in variety of states. So we have seen three states of the directory, they were home, fresh and gone.

So when the request reaches and the directory state is home, what does this mean? The block is there in that quad and there are no sharers. So a very simple case for us. The new requester which we have just sent will get added as the head node of the linked list. It will get the data block and it can start reading.

So operation finishes here. The second option is when the request reaches to the home node, the home is in fresh state. Fresh means there are read only sharers in the system and we already have a distributed linked list constructed. Even in this case, the newest requester will get added as the head node. It will get the data from the home node because the data is fresh and it can start reading. The third case is that the home node is in the state gone.

Gone means there is a single writable copy elsewhere in the system. Home does not have up to date copy. So what should we do? We have to obtain that copy and give it to the requester. In Origin, the home took the responsibility of doing this whereas in Sequent NUMA-Q, it is a strict request response protocol.

So it does not take the responsibility. What it does is the home will add this new node as the head and tell this head new requester that you go to the old head and fetch the data for yourself and then subsequently that block may or may not get invalidated depending on the type of request. Okay. So this is a scenario we are going to do in detail. And while all these several operations which are happening, every block in the RAC, the allocated space, it will always go into a pending state. So there are a variety of pending states available in SCI protocol for every type of an action, a different pending state is invoked. Okay. So handling a read miss at the requester, what happens at the requester? The block is nowhere in the RAC, it is not a local block, so it is not in the memory.

So we have to allocate space for that block in our RAC because now we will bring the block into the squad and we need space to keep it. So in my RAC, I am going to allocate space for this incoming block and we will go into a pending state. So we will allocate space. So the state of this block is declared as pending and we do not go into a busy state because we are not going to NACK requests. Origin was NACKing request, NUMA-Q does not NACK request, so it goes into a pending busy state so that any future requests will be handled appropriately.

Once this is done, then we can start the list construction operation and to do this, we have to go to the home node to add ourselves as the head node. So now this request will go to the home node. Now what happens at the home node? At home, the directory state could be home, fresh and gone. If the directory state is home, that is there are no sharers, we have to give the data to the new requester and establish a link from the home node to the new requester and the directory's new state will be fresh because now we have a one sharer added to the system. The second case is when the directory state is fresh, it means there are shared copies.

So we have to give the data to the new requester, make the new requester as the new head and then remain in the fresh state. Third state is gone where the remote cache has a writable copy, this yellow quad does not have to copy, memory is not valid and so we cannot give the data to the requester, but we still add the requester to the new head and then we will see what happens next in the next slide, but once the requester is added as a new head, the state still remains gone because the data with this node is still not valid. Okay. We will do these things slowly. When the directory state is home, there are no sharers. When the new requester comes, it gets added as the head node of the linked list and once we do this, the home memory would change from the state home to fresh because now we have one sharer added.

So home becomes fresh, the new added requester when it enters the linked list, it is the only node in the system, hence its name is only and fresh. It is the only node and the next term is it is fresh means the data with this node and the memory is up to date. All these actions which we are discussing now are atomic because we will go in pending state unless we finish all these actions, we will not come out of the pending state and we are following a strict request response protocol for doing this. Next is the directory state is fresh. So directory is fresh, new node comes to access this particular block.

Now when directory state is fresh, we already have an existing linked list here. So we have a list existing old head and a few more nodes maybe there. In this case, I can send the data because the data is still valid with the home. So when the requester comes, it is added as a new head of the system and it is told the identity of the old head, so the requester can go to the old head to attach itself here, because home has already attached the new requester.

Once this is done, the requester becomes the new head. The previous head is definitely the old head which now becomes a middle node and along with this data is also given by the home node. Again while doing all of this, we have to go to intermediate pending states, before we do all this link establishing. Okay. Now we are going to discuss the state change for the nodes in the list. Here I showed you that there was a list of green nodes existing and the pink node came and became the new head. So when this is a new head, what should this be called? What should this be called? And what should this be called? So we are going to see that now. Okay.

So suppose I have this existing list in green color, home was in fresh state, head was in fresh state, yes, it was in fresh state. Now a new node, this pink color node comes and adds itself as the new head. So because this becomes a new head, the previous head will become a middle node and now it will be termed as mid-valid. In case I had a single node in my previous list, so this is case 1 when I had 3 nodes.

This is case 2 when I had a single existing node in the linked list. So we had this purple node as a single node in the list. The pink node has gone to the home node, pink node becomes the head node and what will this node become? It was only fresh because it was the only node. Now what is the status of this purple node? It is the tail node of the system and hence it becomes tail-valid. So this is how the states get changed.

The previous head, if it was head-fresh, it becomes mid-valid. If it was the only fresh, it becomes tail-valid. So once we have updated the designation of the previous head to either mid-valid or to tail-valid, we now have to give a state to my new requester. Okay. So this requester would be in pending state and when it comes out of the pending state, it is the head node now and it is head and the next term will be head-fresh because the memory is also fresh. So here, the head node becomes head-fresh, it is the head node and memory's directory is fresh so it gets the term fresh from that notation.

So requester moves from pending to a head-fresh. So this is telling you how to change state of the modified nodes. So in this slide we have summarized the whole thing once more. Okay. Next thing is the directory state is gone. Here it means the data with the memory is not up to date, hence we cannot send the data to the new requester.

So memory state is gone, then we already have an existing list. The new requester comes, home, gives it the pointer to the old head, so it tells go to the old head and get the data because memory does not give data. So the purple node attaches itself to the previous linked list and also fetches the data from the previous head. So now the requester goes in a new pending state, sends a request to the old head for data and also requesting to attach itself as the new head. The old head, what does it do? It gives the data to the purple new requester and it updates its back pointer. What was the back pointer pointing to earlier? The old head's back pointer was pointing to the home node, now it points to the new requester.

So the purple node will now become the new head. Okay. Updating the states of all these blocks which changed. Initially home is in the gone state, we have a new requester, new requester and an existing linked list in the green color. When home is in gone, the linked list head will always be dirty because home does not have the valid copy and the head node is the writer of the system hence it has the dirty copy. Therefore it is called head dirty. Now when the pink node comes and attaches itself, what should the pink node be called and what should be the green head be called? So now the green head becomes a mid-node and it turns from head-dirty to mid-valid and the pink node which is the new head will then become from a blank entry to a head-dirty node.

So eventually this will become head-dirty because it will borrow the status from its previous head. In case I have a single node, that is case 2, linked list only has a single node, even that node will be called only-dirty and now the, when the pink node gets attached to this purple node, the only-dirty node will become tail-valid and my new node will become head-dirty. Okay. Now you would ask that this was a read miss. The node which went for reading, it is getting the status of a dirty node. So once the node declares itself as dirty, can it start writing? So does it mean that the pink node can write? Well it cannot, because the request was for read.

So it is only attaching itself as the new head, it gets the appropriate data from the previous head but it still does not have writing permission. So what should it do? In case it wants to write, it has to invalidate all the blocks in the sharing list before it can start writing. The only advantage here is, the pink node need not communicate to the home before writing. It can simply delete all the green nodes in the case 1, or it can delete the purple node in the case 2, and start writing without telling the memory because memory is already in the gone state. Okay. So in the gone state what we do? We first add the node as the head node, get the data from the previous head and then we have to update the appropriate status of all the nodes which get modified.

So normally the head node gets modified, head can become a mid node or a tail node. So this is summarizing what all things we discussed. Right. So we have seen three cases of the directory being home, fresh and gone. In any of these cases there is a special condition that when the home directory tells the new requester to go to the old head and attach itself, it is possible that the old head is in pending state because it is busy doing something, maybe invalidating the blocks or it has its own new request to generate. For some reason the old head is pending. So what should the new requester do? Home has sent the new requester to the old head, old head is busy.

So should we NACK this request? Should we buffer this request? And these questions we have to answer. So in the NUMA-Q protocol, the old head will cater to this request and say that now I will attach you as the new head. But you are still not getting the head status because I am completing some operation. Because the old head is pending, it still remains the head, it makes the new requester as another head but gives the status of a pending head. So whenever the old head finishes its job, it will give the status of head to the new requester.

Until then it just queues up into a pending list. In case a third node now comes to the same link list, old head is busy, the new requester is also in a pending state, the third one will also join the pending queue. So overall home is always pointing to the newest requester and all the newest requesters which could not become head, they join a queue to become a head with the old head node. So this is the overall picture, you will see it in detail now. Okay. So the question is what if the old head was pending? I am saying that let A be the old head and let B be my new requester.

So B goes to the home node. Now home will send the pointer for A to B, so that B can go to A and attach itself and A is pending. Because A is busy with some memory operation, it does not buffer the request B, it does not NACK the request B, but it adds B at the head of the list in the backward direction but gives it a status of pending. So it calls it as part of a pending list. So end of this, what is the head of the list? Is it A or is it B? So the head is A because A is in a pending state.

Unless it is in the stable state, it cannot delegate the status of headship to B. So A is called the true head right now and B is called the pending head. Once A finishes its task, it comes out of the pending status, then it will give the head node status to the node B. Okay. So this is how we have A in the pending state, it is called the true-head and then it has its own list mid and tail nodes attached, that is the sharing list. When B comes, it attaches to A but it does not get the true-head status, it gets the status of a pending head.

Once A finishes the work, B will become the head. In case a third node C now comes to the same queue, it will go and talk to B, because with respect to home, B is the head node. So home is pointing to B. If I take this example when B went, when B went even then home was pointing to B, it was not pointing to A. But the true head was A because A had a pending operation. Now when C comes into the system, it will go to home and home will point to C. So now home points to C, C goes to B because home tells we go and attach yourself to B, B says okay, I can attach you but I am a pending head, so I can only give you the status of a pending head.

So now C becomes a pending head, B becomes an intermediate node and A is the true head. So this is how we can construct the list in the backward direction for all the nodes which would have request for the same block. When will we come out of this pending status? Once A finishes its work, once A finishes, it will give B the status of the true-head, once B finishes its work, it will give the node C as the status of the true-head and so on. So in the backward direction, each of these nodes will get the status of the true-head to complete their operation. Okay. So the same thing is given in one slide here where the green one is the current sharing list and the B and the C or any other nodes which come will join the pending list. This is the true-head because the node A is pending, it is still the true-head and the others are the part of the pending list. Okay. Alright.

So with this we have understood how the read operation takes place. In the next lecture, we will look at how the writes happen. Thank you so much.