Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 10
Lecture - 53

Lec 53: Correctness and Protocol Interaction

Hello everyone. We are doing module 6 on directory coherence case studies. In this lecture, we are going to look at correctness of the NUMA-Q protocol and also look at the interaction of the two protocols. That is, we have a directory protocol on the outside and a snooping protocol on the inside. So how do they talk with each other is what we are going to discuss. So starting with the correctness aspects, so for correctness we have to consider serialization, deadlock, livelock and starvation freedom.

Okay, so serialization. So here the question is who does the serialization? So what is the serializing entity? In a snooping, we said that the bus was a serializing agent. In the SGI Origin also we said that the home was a serializing agent in the order in which it accepts the request. If the home is busy, then it forwards those requests to the owner in case of dirty blocks.

So clean blocks were serialized by the home, dirty blocks were serialized by the owner. What happens in the NUMA-Q? In a NUMA-Q, all the requests reach the home node and the home node simply adds the new requester as the new head of the distributed linked list and it does this very meticulously immediately without going into a busy state or NACKing the new request. It simply adds the new requester as the head. Once this new requester becomes the head, it will contact the old head and then establish the connection. In the meanwhile, another request reaches the home node, what is it going to do? Home again makes this new requester as the new head.

This new head will again contact the previous head. So overall all the requests which are accepted by the home node will get added as head of the list. Either they will become the head and get serviced. If they are not able to connect as the true head, they will remain in the pending list and eventually get serviced. So the order in which they reach the home node is the order of serialization.

So that's the overall idea. So we'll see it slowly. Okay. So here in the NUMA-Q, home node determines the order in which the cache misses to a block gets serialized. So there is no busy state at the home node. Home is going to accept each request and either it is going to serve the request itself or it is going to direct the request to the existing head.

So the existing head could become either the true-head or the pending-head. So it just adds it to the list. Before it redirects the request, it changes the head pointer. See even if the new requester has not yet become the head, but home already points to the new requester as the head node. It says that you are the head.

Eventually you connect to the distributed linked list, but I will point to you. Okay. So head immediately points to the new requester. Why? So that all the subsequent requests will now go to this new requester and we can construct the distributed linked list either a true list or a pending list. We are going to see an example of this.

Okay. So if a request is not satisfied, then the node will remain in the pending list and await its turn. If it is satisfied, then it can continue with the request. The nodes which are connected in the pending list, that is those which could not become the true heads, so they will get to access the node in the first in, first out or first come first served order because that's the order in which they get constructed or queued up at the distributed linked list. So ensuring that the order in which they complete will indeed be the same order in which they reach the home. Because if you reach home in a given order, you will enter the pending list in that particular order and you will get serviced in that same order. In case the home NACKs a request, we saw this only in the exceptional cases of race conditions, but these requests were NACKed.

They were not accepted by the home and later when they will be retried, they will be retried in a new form. They will be not retried in the, as the same request. Hence in the current form, they will not succeed, but they will succeed in as some new request and eventually this accepted new request will get serialized in the order in which it is serviced by the home. Okay. For example, I have taken four nodes in different colors.

The memory is the green color node. So if it's the only node in the system, the state is home. When the brown color node approaches, it gets added. It will become only-fresh node in the system.

So single node. When third node comes, this shifts here and this becomes this. Okay. So this is how nodes will get, getting added. What is the serializing order? The serial order is the brown one, which is the N2.

N2 got serviced. Then N3 got serviced and then node 4 got serviced. So this is the order of serialization because that's the order in which the home services those requests. Okay. So suppose I had this list to begin with. When the pink node comes, it attaches itself first.

So the home makes the pink node as the new head and then this pink node will connect with the yellow node. Because in case I have a new requester coming and the pink is yet to attach. So this is, this process is still going on. But to maintain the order of serialization, in case a new request from a blue node comes, it goes to the home. Home immediately makes this as the head node and then tells go and connect to the pink node. Pink node will say, well, I'm still connecting to the previous node, but it would attach this and they can form a pending list if required.

So in case the previous operation is not finished, this can become the pending list. If it is finished, then the nodes will get their correct status in the list. So overall, we are going to service the request in the order in which they reach the home node.

Okay. So same idea. Pink node is still busy. It is currently the true head, but it hasn't finished the work. Hence, both of these new nodes will enter the pending list. This is the pending list and this is the pending head and the order of services. So this is the first node, second, three, four, five. Right.

So this is the order of serialization. So serialization is done at the home node. These three aspects are very straightforward in the SCI protocol because that's how the protocol is designed in a robust manner. We have a distributed link list of requesters. We also hold a pending list of nodes.

Now when I have a list, do I need extra storage for doing this? Right. So deadlock problems comes, when I have a fetch deadlock or a buffer level deadlock. But here in the SCI protocol, I don't have buffers at all. We have a space in the RAC with forward and backward pointers and that's the only storage I have. Every time a node is part of some of the other distributed linked list and we need no extra storage. Hence, because there is no buffer, we won't have the problem of deadlock.

We are following a strict request response protocol. We are not sending NACKs to any node only in case of exceptional conditions where the NACK will be retried in a different form not using the same form. So therefore, there won't be issues of livelock because we are following a strict request response. So there are no NACKs and hence there is no contention due to race conditions. The nodes simply join a pending queue if they are not serviced.

If they are serviced, good. If otherwise, they join a pending queue and progress in a particular fixed manner. Hence we have no livelock problem. The order in which the link list is constructed in a first in first out manner, the order in which they reach the home node ensures that each node gets serviced in a fixed order. Therefore, we have no starvation. So we have no deadlock because we don't need extra storage.

We are following strict request response, do not NACK, hence we have no livelock. And because the linked list is guaranteeing a fixed order, we have no starvation. Well, a node can be part of variety of pending list. So a particular node, that is I'm not talking of a memory block or a data block, but a node which is a quad. So one node can be part of multiple pending list and this list number is decided by how many outstanding requests can go out from one quad. Okay.

The space for the pointers is already there in the cache. Even if you have more outstanding requests, it won't matter because you already have allocated space in the RAC for the data and the forward backward pointers which is managed nicely. Hence we have no extra storage overheads. Okay. And any pending node cannot replace because unless you make complete your operation, you cannot leave your cell in the RAC until you become a normal node.

Hence we don't have this problem. So this way we have proved the three things. The next topic to discuss is the protocol interaction between the directory and the snooping. Okay. Right. So I would say that you pause the video and reflect how the architecture of the NUMA-Q was that we had a SCI ring to which quads were connected. Every quad had four processors connected over a quad bus. Inside the quad we had snooping protocol, outside we had the SCI directory protocol.

Till now we are only discussing the SCI directory protocol where RAC is the only visible entity from the quad. So from the quad you can only see the RAC which represents a pseudo processor or pseudo node for the outside protocol. But at the end we have to understand how these two protocols talk to each other for correctness as well as how the actions are handled. So we are going to discuss that now. Okay. So on the slide you can see we have, on the left you can see the IQ-Link board with the bus controller, the directory controller and the network interface.

And within the quad we have the quad bus, four processors with their L2 caches. Quad bus is connected through the bus controller OBIC to the SCLIC which is the directory controller. Both of these have access to the tags as well as data of the remote access cache. This whole thing through the data pump network interface connects to the SCI ring.

Okay. Now interactions for a read request. So processors-level read miss is going to the

L2 cache. If I go to this slide here, this processor sends a read request there is a miss on this read in this cache. Right. So I have a read miss here. Who will satisfy this read miss? Either the memory connected within the quad. Right. So somebody here will satisfy at the quad level or it may have to go outside.

But if it has to go outside it has to go through the directory controller. So we will see how this happens. So processor-level read miss will go on to the quad bus. All the processors on the quad will snoop this bus.

Even the OBIC bus controller will snoop. Why should the bus controller snoop? Because it is attached to the RAC and RAC may be able to satisfy this read miss. You don't know whether the read miss is for a local block or a remote block. So all the processors as well as the RAC, which is the directory controller snoops on the bus. If the request can be satisfied by the RAC, that is the RAC has the data block and the directory state is appropriate, then it is serviced immediately. The local memory or the local cache can also serve this request either RAC serves or the local cache or the local memory can serve. Okay.

So when this read miss goes on to the quad bus we have the concept in the snooping protocol of sending the snoop replies. So you have to send the shared wired-OR signal and tell whether the snoop results are ready or not. So we have a provision to wait for the snoop results. If the snoop results are not ready, we can wait for few more cycles and only then every module will know what action to take. Because the snooping would be completed. So once the snoop completes, the data block will get transferred and will be loaded locally in the MESI style, because the SMP nodes which we are connecting follow the MESI snooping cache coherence protocol.

We are using the quad bus which is a split transaction bus, but it has, it expects an in-order response. It is not out-of-order response, it is the in-order response split transaction bus. Okay, we will see an example. So you have the quad here, with four processors, caches, directory controller going outside to the SCI ring. We have the requester, this one suppose this processor request for block A.

Now this block A where is this block? If I say this block was present with this particular processor, so when the snoop request goes on to this quad bus, the fourth processor will readily respond to this request and give the data block to the requester. So this is how locally allocated block can be satisfied. In case the requester says I want block B and the block B is sitting in the RAC. Because it is sitting in the RAC, when the snoop request goes, this directory controller will see this, and then supply this block B to the requester. Okay. So till now we are relying on the snooping protocol to service either

A from another processor or B from the remote access cache.

Now the third case is when we have to go out. So when we have to propagate a request for a block which is neither here, it is not in the quad, it is not in the RAC, so you will have to go outside and fetch the block from another quad. So when the request has to be propagated off the node, how do we handle snooping? Because here the snoop has gone, none of the local processors have the block, the RAC doesn't have the block, but the directory controller understands that this block is a remotely allocated block. So local caches can't handle it, the memory block is out of their range, even the local memory cannot handle it. So the directory controller comes into picture here and it says I will bring this block for you from outside and in the meanwhile, the current ongoing snooping transaction has to be stalled. Well we have a in-order response type of a snooping bus, but this is a special case, where a deferred transaction from the directory controller can be serviced in future.

So the bus can wait for this particular transaction or rather it stalls this transaction and services future request. So the snooping bus simply says, this, this transaction which is going out is deferred, future transactions will start happening, and the directory controller takes this request, goes out onto the SCI ring, follows the complete directory protocol, acquires this block, puts it into the RAC and then services the request back onto the quad bus using again a response to the deferred transaction. Okay.So this is what is going to happen. The directory controller will pause this transaction, fetch the block and restart the transaction. So when I have to go off the node this OBIC detects and sends a deferred response onto the quad bus. It tells the quad bus, I will bring the answer for you, you continue doing your regular work. So quad bus leaves the request as pending, it leaves an in-order way of working and then starts servicing new requests.

Okay, so these new requests are mainly internal requests because the quad will have its own processes running. Okay, in the meanwhile, the OBIC, the bus controller passes it to the SCLIC to invoke the directory protocol. Eventually you will get the remote response, so the block will come using the SCI directory protocol. Once the block comes, it is placed onto the quad bus by the OBIC and it completes the deferred transaction. Okay. So here now the request is for block C and where is C? C is sitting in a remote quad up there. So because C is outside, what will happen? The OBIC will go out go there. Eventually using the SCI directory protocol obtain a copy of block C in its RAC, once it gets this block, it is going to service this onto the quad bus to be handled by the or taken up by the requester.

Okay, so that is the path of a cache miss for a remote block. So this quad bus which we are talking about it is a bus on a symmetric multiprocessor which is an off the shelf node

I am connecting to a SCI ring and I am expecting that buses within the SMP nodes which I connect, must be split transaction why? Because in case I am not able to service the request locally, we have to go out and if we have to go out it is going to take a lot of time for the response to return. If the bus was not split transaction, it will be stalled for a long time and it will not be an effective system. Hence for performance reasons all the SMP nodes which I am going to connect must have a split transaction bus. Okay, so we need this. It is also required for correctness because if I hold up the bus for a long time what is going to happen? I am not going to allow local misses, neither I am going to serve incoming requests and if I do not serve incoming requests, I am inviting potential deadlock problems. Okay, so to solve potential deadlock problems to keep the things moving, even if one transaction goes out of the SMP system, other transactions should proceed, hence we should have a split transaction bus when I am connecting such nodes. Okay, so now interactions for a write request. So that was a read request, this is also straight forward. The actions of bringing the block snooping everything is same as in the read case OBIC will snoop for the block with respect to the status in the RAC. If the block is owned by the local quad or it will send the request outside. So block is owned by the local quad or the request must be sent outside.

In the case of write as you all understand, we have to first make this node as the head node of the sharing list, then we have to invalidate all the other sharers, that is the purge operation and then finally we will be able to write. Okay, so we have to become the head, purge the list, and only then start writing. So write is similar. Once you get a block in the RAC, you have to follow the SCI protocol. When the SCLIC is done with all of this, it gets the permission to write. See after purging, the node will become only-dirty, right? Only then we can say that you can start writing. Here is when the directory controller puts a response on to the quad bus so that the processor which had requested the block can start writing to that block. Okay. So for writing follow the same processes as the previous slides, first obtain a copy of the data block, become the head purge the list, become the only dirty node, then put the answer on the OBIC bus, so that the requesting processor can acquire the block and start writing, right. So that's all we need to understand here. Quickly we'll see two limitations of this quad and how it affects certain actions. A locally allocated block that is the one who has this address in this particular local memory but it is cached remotely. So this means the block from my memory module has been cached by some other quad and eventually this block will be written back or you will want to bring that block for a own read or write miss. So your block has gone outside, it is changed there, and now you want it back into your quad. So when you bring it back, what happens? When you bring that block back and put it on to the quad bus, by that time the memory has forgotten that this was a deferred transaction. Memory said, I don't understand what data is coming because this is for some previous transaction which I did not keep track of. Okay. So here is the limitation that the OBIC bus controller has to have a special action to update the main memory In normal cases whenever there is a write back, the memory gets updated.

Okay. So normal cases data comes, memory gets updated. But in this case, we are bringing the data, the data is coming from outside. So I had a deferred transaction, beyond the deferred transaction memory forgets that I was supposed to receive a data item and it starts servicing other requests. When the data really comes, the OBIC will put the data response on to the quad bus, the requesting processor will fetch this data. But the memory will not fetch this data because it is not in the particular order it is expecting. Hence, in this particular case OBIC has to make a special write to the main memory. So that is one limitation. The second limitation is, when I am having two write requests for the same block which are going outside. So I have local write requests, write to the same block by two different processors. First write request has gone out to fetch the data block. When the second write request for the same block comes, what should we do? We should rather keep it on hold, pending and when the data comes, we can service it. So we have two write requests, when the first write request has gone out and the second one has come, the option is to keep it pending. But here it is not kept pending, but we rather NACK that request and it keeps retrying. So that is one more limitation of this current SMP node which we are considering. Okay. So quick example: here in the quad there is a write to block B and where is block B? Block B is outside somewhere in the quad, so there is in this SCI ring elsewhere in another quad the block B is sitting, then write B comes from another processor, so two processors from the same quad want to write to the same one and when this request is gone out, we NACK this request. We should have rather kept it pending, but we NACK this request and once you NACK it is going to keep retrying itself. So this is going to waste some amount of performance but that is the limitation of this quad.

So ideally we should have buffered this request and when the response for the first one comes, we could give that same answer to the second request, but no we do not do that we knack and create some performance issue. So that is the second limitation of the quad. Now last point is handling serialization in the protocol interactions. We are going to see two cases for locally allocated blocks and for remotely allocated blocks. Serialization when the block is locally allocated. So which is the serializing entity when my block is locally allocated? The picture is showing the quad bus. We have a memory block A which is a locally allocated block because it is sitting in my local memory module. Second processor wants to access this data block, so it takes it from its local memory. Right. So we are going to consider serialization for locally allocated blocks. So the overall idea is, what is the serializing entity for the blocks which belong to the local memory. Okay, so these blocks in the local memory will go through the quad bus and get accessed by variety of processors within the SMP. Hence the quad bus could be the serializing entity here.

What about request for the locally allocated block which come from outside? So from the SCI ring, if there is a request for the block which is sitting in this memory module, it will come through the directory controller, through the OBIC and again go on to the quad bus. The status of the block is not known to the RAC, but we have to find out whether this block is locally dirty or read only. Because within the quad ,we could have modified the data block. This is required because, in case the external request is for writing, you want write permission, to give write permission ,we need to see that the, we have to provide the latest copy of the data. The latest copy could be in one of these local processors caches. Hence we have to go on the quad bus to the local cache, fetch the most recent data and give it to the outside quad. So overall for all the blocks which are sitting in my local memory, we have to go through the quad bus and hence quad bus becomes the serializing entity for locally allocated blocks. Okay. So if the directory state is home and no remote node has cached this block, that is the block is here and some local processor is using it then we have to go through the quad bus and hence quad becomes the serializing entity in this case.

In case there is an external request coming to my quad, so read a comes from outside, Now this outside request goes to the OBIC controller which puts the request on to the quad bus and to put this, as you can recollect you will have to follow the snooping protocol for a split transaction bus and it will get order in a particular manner. So to serve incoming request for a block that may be clean or dirty in the local cache, we have to go on to the quad bus. Third case is an incoming request that would make the status has gone, that is this incoming request is for writing to that block. So write request comes, previous was a read request. Now when the write request comes, can we permit to write? Because we have to first give that data to the writer, before giving the data we have to obtain a fresh copy. Not necessary that the fresh copy is there in the memory because the fresh copy could be in any of these nodes if that copy was dirty. Therefore even in this case, we have to go on to the quad bus to get the latest copy of the data item.

So to get the latest data item we have to go on to the quad bus, fetch the data, either from the memory if it is a clean, or from another cache if it is dirty, and while doing this definitely you have to invalidate all the local copies within the quad. Right. So overall since the incoming requests from outside or the local misses from the processors amongst themselves, all of these appear on the quad bus and hence it is very natural to let this quad bus become the serializing entity and therefore the quad bus at the home is a serializing entity for locally allocated blocks. Okay, that is the first case. Now second case is what happens for remotely allocated blocks. Remotely allocated means, those blocks which I have in my RAC ,but they belong to external quads, that is the home is elsewhere. So request from a local processor for a block that is in a remote cache. So this block B is in my RAC. It has come from outside and suppose there is a request for accessing this block.

So we have a write request for B, B is in RAC, B belongs to another quad. So we have to go to the directory controller. In case I have another request for B from outside, SCI request coming from outside when will this happens, if there is a purge operation or if I have the block in dirty state, in some condition a request will come from outside for this block B either to be updated or invalidated or something or to link this block B to some node in the distributed linked list. Right. So there are variety of occasions where you will get a request from outside for a remotely allocated block. So when such a request comes, even that request goes to the directory controller. If I have a third request from the same quad for the same block, even this request will go to the directory controller because this block B is not in my local memory. Right.

So this was the local memory. It is not here, so OBIC will do the snooping, it will win the snoop and then try to cater to this request. So the request in green color purple color, that is request 1 for writing, request 2 which comes from outside, request 3 which comes from inside, all of these requests are going to the directory controller, that is the heart of the system and who is going to serialize this. Naturally the directory controller. Because the order in which they reach the SCLIC, is the order in which request to this remotely allocated block will be serviced. Right. So activities within a quad bus which are related to remotely allocated blocks are serialized at the directory controller and not at the bus. Because bus cannot service it, so it has to go to the OBIC and from there to SCLIC. So request from the local processor for a block in the remote cache or incoming request from the SCI protocol to the blocks which are in my RAC, are serialized at the SCLIC and hence SCLIC becomes the serializing entity for remotely allocated blocks. So with this I think it was clear that you could understand the interaction of the directory protocol with the snooping protocol, how reads within the quad are sent out, how writes are serviced, how serialization happens and so on.

So with this we complete the topic on scalable cache coherence. Thank you so much.