Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 08
Lecture - 46

Lec 46: Proving Correctness (2)

Hello everyone. We are doing module 5 on Scalable Shared Memory Systems. This is lecture number 7 and here we are going to continue the discussion on proving correctness aspects. In the previous lecture, we have looked at the correctness aspects related to write serialization with respect to one location. In this lecture, we are going to start with proving serialization across different locations for consistency. The previous one was write serialization for coherence.

This one is write serialization for consistency. Okay. So, a quick recap about what do we mean by write serialization in the context of consistency is that we need to prove write atomicity and write completion. Okay. So, what was write completion? When a process writes to a particular location, it has to make sure that this write has reached all the nodes in the system before it is permitted to proceed with further instructions. That is write completion.

And write atomicity was that if a processor reads a value, it should be sure that the value it is reading, that same value is visible to all the nodes in the system, that is the write which, whose value I am reading right now, that write has reached all the processors. So, this was write atomicity. Okay. Write completion is after a write, we have to wait for all the processors to see this write before we can proceed to the next operation. Write atomicity was that if I read a value, then the process has to wait for everybody else to have seen this value. Okay. So, this, the write whose value I am reading must have been performed with respect to all the processors to guarantee write atomicity. Okay.

So, moving on serialization for consistency, here we are going to take three cases. The first case is, when I have a bus based system, the second case is, when I have a distributed scalable network, but I do not permit caching of shared data and the third case is our scalable cache coherence protocol. Alright, so in a bus based system, serialization is straightforward, because as soon as we get the bus, we are sure that the write is complete. Whereas in a distributed network, we are not caching the shared data. So, the shared data is only with the home node, no other cache can hold this data.

So, in case a processor N1 wants to do a write, it has to come here and perform the write, that is it will actually send write x equal to 2 and tell home node to write it because it cannot cache that block. Okay. So, when this write x equal to 2 reaches the home node, how does N1 know that the write is complete? It can know this only when home sends an acknowledgement to N1. So, it has to wait for an explicit acknowledgement from the home node that the write has reached the main memory location. And remember there are no other sharers to this block. So, home node simply takes the value of x equal to 2 and updates its location.

Okay, so can we do some optimization here in case there are multiple such writers, they will all get queued up in a FIFO order at the home node and as soon as a request enters this queue at the home node, we can say that the write is committed to happen. Because we need not wait for the value x equal to 2 to enter to the memory bank. In real, you can assume that eventually that value will enter the memory location because it is queued up at the home node. Home is going to take one by one request from its queue and going to update. So, as soon, as soon as you enter your request into the FIFO queue, we can assume that the write is complete. So, that is an optimization that we can assume commitment for completion. Okay, then write atomicity, here the write is visible, that is, is the write to a particular location visible to everybody? Yes, it will be because all the nodes will always go to the home node to read.

So, whenever a write reaches the home node that is reaches the main memory, we can say that it is visible to all processes. Right. This is very straightforward because all processes will have to go to edge to read the value of x, and the order in which they go, if FIFO order is maintained by the home node, we can guarantee serialization, atomicity, everything. Now the third case, third case is that we have a distributed scalable network, but in addition to that we are also caching the shared copy. So, we are dealing with a CC-NUMA architecture, cache coherent non-uniform memory architecture. Right, so here it is very difficult to assume that once we reach the home node, my write is complete.

We could do that in this case, in the second case because home was the only owner of the shared variables, but now shared variables can be distributed, will have multiple copies in the network and if a particular write reaches the home, it does not mean that the write is visible to all the nodes. Okay. So, let us take a scenario where I have a node N1 which sends write x equal to or write x not the value because it is a caching based protocol. So, it wants to do this write, so it will send a request for writing to the home node. Okay. So, it sends a write x request to the home node, if it is an update based, it would send a new value, if it is a invalidation based, it will send a write request to home

node and accordingly the protocol will take further actions. Now suppose this is finished as a first request and the second request is for writing to variable y.

Now, both these requests will go from N1 to the home node, assuming x and y are both mapping to the same node h in this case. So, the order in which H sees the value is it, it sees write x happens first and then y happens next. Okay. So, if it is an update bases , it will say that x equal to 2 took place and then y equal to 4 took place in that particular order. However, if it is an invalidation based protocol, then home is going to send invalidation for the block x to other processors, it may send an update to x to other processors depending on the type of protocol. So, suppose it sends an invalidation of x and subsequently for the request y, it will send another invalidation.

For y, when y comes, it will send an invalidation of y to the respective sharers. In case I have a node 3 which is caching both x and y. So, here it is very much possible that the invalidation for y reaches before the invalidation for x. So, the order in which it sees is that first y was changed and then x was changed. So, this is going to violate the sequential consistency requirement that we see the writes happening to the variables in a fixed order.

So, how do I prove write completion in the given scenario? Looking back here, we said my write x happened with respect to H, then write y went to H and when it reached home node, if I assume that the writes are complete, this is not sufficient because the order in which home sees the updates is not the same as other nodes will see the updates. So, how do we solve this problem is the next question. So, I would say that when my write reaches the home node, we need to wait for acknowledgments from all the processors before we can assume that the write is complete. Okay..So, N1 in this case, it sends x, write x request to the home node. Now once this says, write x is sent to the home node, home simply sending an acknowledgement to N1 is not sufficient because H has to do further actions of either invalidating other copies or updating other copies, depending on the protocol.

So, it has to do all these actions. So, it has to do these actions and all the nodes which were the sharers of this particular block, they all have to invalidate and send an acknowledgement to the home node for doing this. So, once this acknowledgement is received, only then h can tell N1 that now your write is complete. Okay. So, for write completion to be guaranteed, we have to wait for all the acknowledgments from all the sharers or all the processors before we can say that the write is complete. Well, this is going to take time because invalidations or updates will take their own time through the

big network to reach all the processors.

Once they reach the processor which is the communication assist, it will go through the hierarchy to the cache and then take appropriate actions depending on the state of the cache block. Alright, so should we actually wait for all this to happen or can we do some optimization? So, we can say yes, I have an option that as soon as it reaches to the communication assist at a particular node, so this is one node, it reaches the CA and then it will go to the cache and the processor. So, eventually it will handle the invalidation, but as long as I reach the CA, I can kind of commit that yes, I am going to invalidate and the protocol can proceed. Okay. So, a node can immediately send an acknowledgement as soon as INV reaches it as long as it guarantees that it is going to invalidate the block in the proper order. Next is serialization in CC NUMA with respect to write atomicity.

Write atomicity is that when I read a new value, I am sure that everybody else is going to read the same new value because the write which produced the value for me has reached all other processors. We have to handle this differently for the two types of protocols. We have an invalidation based protocol and the second is update based version. So, in invalidation based protocols, the current owner of the block does not allow access to the new value to any process until all the invalidation acknowledgments are returned. Okay, so what does this mean? Current owner could be a dirty node or a home node.

Suppose it is the home node for an example and a value is updated here recently. So, X has become 2 by some updation or a write back or update coming to this node. Now H is going to send invalidations to all the nodes for this write to take place and it can say that okay X equal to 2 has reached the home and if a new requester, now goes to the home to read X, should H provide X to this? So, it says give me X. So, should H give the value of X? We are saying it should not until it receives the acknowledgments of all the invalidations. Okay, so if I do this, I can guarantee atomicity because the new reader or any reader, any new accesses to this new value will be prohibited until all the acknowledgments of the invalidations are returned to the owner.

So, that is invalidation based. What happens in an update based protocol? Our solution to this idea was prohibit access that is do not allow access to the new value. Can I say this in an update based protocol? Because can I say to a processor that I am giving you the new value but you do not read the new value. How to stop a processor from reading the new value? Because the idea is once the new value reaches to multiple nodes, for example, there were three nodes, this got the most recent value.

This is yet to get value. So, these two are slow in the network, they have not yet

received the value. So, I want to say that node 1, you do not read the value because 2 and 3 are yet to receive that particular value. How do I implement this idea? Okay. Well, it looks a weird idea, but that is what we will have to do, if I want to guarantee write atomicity. I have to prohibit the processor from reading the new value. So, we are going to actually do that.

We will send updates to the processor, then we will tell the processor do not read the value and wait until I permit you to read the new value. Okay. So, it cannot use the value x equal to 2 until the other nodes see it. So, that happens in the first phase. Second phase is, the home node or some controller would send the updates to all the nodes that is for example, 2 and 3. Eventually, 2 and 3 receive the data, they will send an acknowledgement to the home node.

So, once all the nodes have received the new value, they are going to send an acknowledgement. Once all these acknowledgments are received, so I have to collect all these x that is 2 and 3, both of them will send an acknowledgement back. And once I receive all these x back, I have finished the first phase and I will say ,now all three of you can start reading the new value. So, once all the acknowledgments are received, we have to send another message to all the processors giving them permission to start using the value from their respective caches. So, this is an idea which looks valid, it is correct, but it has its own overheads, that is you are prohibiting a processor from reading a value and it already has the value.

So, it is difficult, it has its own performance impact and therefore, in a scalable directory based protocol, we do not prefer update based protocols. We would rather go for invalidation based protocols which are more robust and give good performance. Okay. So, next correctness property is deadlock freedom. So, we have proved write serialization in coherent systems consistency, now we do deadlock, livelock and starvation. So, for deadlock freedom, when I have a big scalable network with distributors, several copies and so on, I have several buffers and requests and responses are kept in all these buffers and when buffers are in place, we have to worry for deadlock.

So, in case I have less buffers, there will be more chances of deadlock, if I can have enough buffers, ample buffer space, then probably I am able to service the transactions without falling into a deadlock situation. That is one option to solve deadlock. The second option could be I send negative acknowledgments once my buffers are full or beyond the capacity, we will keep on using NACKs to solve the deadlock problem. And the third aspect is, we have all these protocol optimizations, if it is a strict request response, then my request is going and my response has to come back and it affects

deadlock or related to lot of limitations on the buffer space, it might happen that my request is there, but I am not able to get the response back because of either buffer deadlock or because of protocol level deadlock. So, this can be avoided if I can have independent networks, one set of network for the request traffic, another for the response traffic.

For example, if you have a single lane road in any city, all cars are going in both directions. So, if there is no order, all cars will go in random directions and you are going to incur a deadlock. But to solve the same problem, if I divide the same road into two neat lanes, each can follow its own path, that is one in one direction, the other in the opposite direction. So, I will have two networks, one for the request and another for the response. So, if I do this neatly, I can avoid the deadlock.

So, for strict request response, up to two networks, it is easy, but now we have protocol optimizations. We went for intervention forwarding, we went for reply forwarding and both of these are not going to send a reply back, but they are going to generate further and further requests. So, how many such networks, virtual or real networks are you going to afford? Okay. So, if I just have traffic in two directions, left to right and right to left, it is all right, but if left to right traffic is going to generate further and further more traffic, which will eventually come back, there is no limit to how many virtual or physical networks you will be able to provide. So, this is the scenario of deadlock in case of protocol optimizations, which we have to address. Okay. So, for deadlock freedom, first thing is, let us provide enough buffers.

Second is we can use negative acknowledgments and third is use separate request response network, either they are physical networks or they are multiplexed over the same network using separate buffers. So, you can see the concept of a virtual network that is the multiplex network. So, two networks are sufficient, if I have a strict request response method, because every request, so I have two sets of networks, one in this direction, one in this direction and they both have independent buffers. So, this is a very neat design, which will avoid the deadlock. But now I do not have strict request response, we have intervention forwarding, reply forwarding and so on.

So, how many networks would I need? It is equal to the longest chain of the transactions, that is I sent a request to some node, it sent an intervention, which is a kind of another request to this node, this node further sent another intervention to the third node, then to the fourth node, probably out or comes back. Okay. So, I had, I would say four different types of transactions before I finally get my response. So, here I would need four independent networks, one for the first request, second for the intervention one, then another intervention, third intervention, then the respective responses. So, you

can see that I would need virtual several such networks and this number will depend on the longest chain in my transaction. Well, this is workable, but not a practical solution, it is very expensive, definitely to have several physical networks, but even if it is logical network, still you will have crunch of buffer space and a lot of management to do.

If you end up having such networks, if at all using several resources, most of the time it is going to be underutilized. Alright, so this idea of having separate networks for your performance optimizations, although correct, but not a good solution. So what do we do? The idea is you do not assume strict request response, that is allow your performance optimizations of related to protocols, but try to detect potential deadlock. I am not saying detect deadlock, we have to detect potential deadlock, that is a possibility of deadlock much ahead of time. So do not allow deadlock to happen, try to find out before time. Okay.

So we will say, provide two real or virtual networks and then detect that the deadlock may happen and then try to avoid this possibility of deadlock by using certain methods. Okay. So how do you detect a possibility of a deadlock? How is a potential deadlock detected? Right. So we are now going to detect possible deadlock situations. So the first one uses, suppose I have these buffers, one is a set of input buffers and one is a set of output buffers. So this is where requests are coming from outside, this is when I am sending requests to the network. So both input and output request buffers will keep on filling and I will say that we have a potential deadlock, if the buffers fill beyond a given threshold.

So you can find out a threshold Th using empirical analysis and say that once my buffers are full, suppose this is my threshold Th,, once all these are full, then I can say that probably there is a chance of deadlock. The other possibility is, this is the head of my queue, this is the head of the buffer queue and this head in the buffer, that request is of a type which will generate further requests. So this request at the head is going to generate intervention or reply forwarding which may or may not eventually generate further requests. So if this was strict request response, that was all right, as soon as I pop this request I can send the reply. But if this request is of the type intervention or reply forwarding, it is going to generate further requests, so it is going to again add to the same buffer. Okay.

So if the buffers fill up beyond a given threshold and the request at the head of the queue is one, that may generate further requests like interventions or invalidations, we can declare that there is a possibility of deadlock in this scenario. Okay. What about the output queue? The output queue is where we are sending out, our requests are going out. Right. If we are not able to succeed in sending our request on to the network, we can say

there was a potential deadlock, if no progress happened for example up to T cycles. So if these requests were sitting for several cycles and T cycles have happened and still this request was not popped from the buffer.

So you can say that there is a possibility of deadlock. So this is how we can say potential deadlock situations can be detected. Once you detect such a situation, we have to do something to avoid deadlock. Now what actions to take here? So the actions depend on different manufacturer's design decisions. If I take the Stanford DASH system, it says it will send NACKs. So once it detects a potential deadlock, like here, this was the input queue, it was full and it says now I detect a potential deadlock, what will it do? It will keep popping requests from here and send a NACK.

It takes this request, says NACK, I am not going to serve you. It takes the next one, says I am not going to serve you. This is how it is going to empty the buffer space here by NACKing requests one by one. And how many will it NACK? Either until the threshold Th is coming down, that is your Th is not reached, you delete several requests, so that you have space in the buffer or you reach a request which is a strict request response type, that is it is not going to generate future requests. So first thing is you send NACKs and second thing is you can do this NACKs until your head request is of a strict request response type, that is for input buffer.

For output buffer, you keep on revoking or NACKing the request until the output queue is no longer full. So you can remove a few requests and make space. Definitely all the NACK requests will be retried in future. So detect a potential deadlock and NACK requests, that is the solution for the Stanford DASH system. Okay. Now, what does SGI Origin 2000 system do? It does not use NACKs.

What it does is it says, once I detect a potential deadlock, I am going to convert myself from intervention or reply forwarding type to a strict request response. So I will dynamically back off and become strict request response. I will become strict once I detect a deadlock. Okay, how do I become strict? I have these requests and these requests were going to generate interventions, that is, I was going to send an intervention to some node which will then send me an acknowledgement and I will use this ACK to send my response. Okay.

So this was the order. To convert this into strict request response, what we will do is, this portion of intervention I have to avoid. So we will instead of this, we will say the requester for this. So we will send the ID of the owner to the requester and tell that now you go to the requester and serve yourself and I am not going to do the intervention and reply to help you out. Okay, I hope it is clear that instead of sending an intervention, we

revoke the intervention and we send the ID of the owner to the requester, so that the requester can go to the owner and get the answer back. Okay.

So this way, this particular entry will get freed up. So the advantage of this is that the NACK based protocols are not normally robust solutions because once you NACK request, they are going to retry and once the request retry, it may cause congestion, several retries will come probably all together increasing the network traffic and overall latency. And so therefore, the Origin 2000 processor does not use NACK, but it converts its protocol from intervention or reply forwarding to a strict request response. Right. So the next property to prove is live log freedom. Now a quick recap, what was live lock? That there are a lot of activity happening, but there was no progress happening in the system. That is requests are going, but they are probably getting NACK, they have to retry and overall no system progress takes place.

So if I have a system where I can avoid deadlock using enough buffer space, then and each buffer is following a FIFO order, then here the problem of live log will not happen because every request will get a chance to get served and so on. But if this is not the case and we use a NAKC based solution, then all the requests which are going to NAC Kare eventually going to retry. Probably they will retry at the same time and in such a pathetic situation, we will again NACK all of them and they will keep on retrying. Okay. So this is the scenario of a live lock and let us see how do we solve this. So if I have enough buffers, then live lock and starvation are taken care of automatically, if they are following a FIFO order.

Otherwise multiple nodes will try to write to the same location at the same time. When multiple writes happen, some will get served, that is the first one gets served in the first come first served order and the others are going to get NACK. So if I do a NACK, one gets serviced, the others do not get serviced. So NACKs are good to avoid live lock due to race conditions. That is, if I have 1, 2, 3, 4 of them coming, I am going to service this and NACK the others.

Next time maybe somebody else gets a chance. So slowly one node will get a chance at a time, but several will get negative acknowledged. And if you remember, we have used the NACK based idea even for avoiding deadlock. If I go back here, the DASH system is going to use NACK to avoid potential deadlocks and all these NACKs are going to be retried later. So if NACK is used for avoiding deadlock, then all the retries might lead to livelock. So it is kind of a circular problem. That I need NACK for avoiding deadlock, but the NACKs will do a retry which may cause a livelock.

So hence, NACKing the request is not a very good idea. However, the other solution for

deadlock, which was used by the Origin protocol, that is we do not NACK, but we dynamically back off and become strict request response. If I am using this, that is becoming strict request response to avoid deadlock, this will also guarantee forward progress because every request will get served and it will never have to retry. Okay. Next was starvation freedom. Starvation is the case when several requesters are there for the same block and some get serviced and there are few who never get serviced for a long time.

So they are kind of starved from getting served. If the protocols are designed neatly, then it is unlikely that starvation might happen. If I am using FIFO buffers, it is best because everybody will be queued up in the order they reach the home node. So I have a FIFO queue, here no starvation happens because the order in which you are reaching, you will get serviced in that same order. So FIFOs are a good solution for starvation. If I am using NACKs and retries, then there is a possibility that starvation may happen because if all the retries happen at the same time, it is possible that some of the nodes will not get serviced.

So what do I do in this situation? Easy thing is, do not do anything. Well because the situations or scenarios when this would happen are very rare and we can definitely depend on the variability of delay in the network. That is the pathetic situation that all these nodes come to the home node at the same time. Right. Several of them come to the same time and they keep on coming again and again, getting retried at the same time is very rare and there will be network delays and it may happen that such situations will be solved automatically on their own in the execution time. So we can wait or bank upon the variability of the network delay and let the system solve the starvation on its own. The third solution is, add random delays between the retries so that the retries do not occur simultaneously.

The fourth solution is assign priority to the requests. That is, if I am NACKing a request say 50 number of times, the next time it comes, I will give a priority and solve this particular request. Alright, so we have proved all the properties in the scalable cache coherence system. We started with write serialization for coherence. Then we did write serialization for consistency.

We looked at deadlock, livelock and starvation freedom. So with this we have completed the correctness aspects of scalable protocols and with this we finish this module. Thank you so much.