Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 09
Lecture - 49

Lec 49: Correctness Issues

Hello everyone. We are doing module 6 on directory coherence case studies. Currently, we are looking at the SGI Origin case study. This is lecture number 3 in which we are going to look at correctness issues for the SGI Origin protocol. So, correctness aspects as you can recollect, we are normally going to discuss about serialization to a location, across locations and then deadlock, livelock and starvation related aspects. So serialization of operations across locations when the variable names are different is used for consistency aspects and the arguments of correctness for this is similar to the arguments we did where we discussed the directory coherence protocol in a generic manner.

So there is going to be no change in the proof of correctness for serializing operations across locations. For serializing operations across the same location, for coherence, we are going to discuss two examples to see why it is very much required to follow certain discipline when we implement the protocol. Okay. So, serialization has to be managed by some entity. It was the bus in the snooping protocols whereas in the directory protocol, we were discussing, can the home node become the serializing entity.

So yes, it can become a serializing entity provided it is going to handle all the requests in a disciplined one by one manner. So if the home node is going to buffer all the requests in a first in first out buffer, that is a FIFO buffer and service them in that particular order, we can say that home is a serializing entity. However, having buffer has its own problems, that is you will have buffer overflows, you will have to maintain the flow control and various other aspects. So Origin is not going to do this but there are other architectures which follow a FIFO based buffer queuing up of the requests. Okay, right. So for serialization, we need an agent and if we consider home node to be a good agent for doing the serialization, we require that the requests are buffer in a FIFO order at the home node.

This is good enough but we would have problems related to the buffers because once you have buffers, we need to take care of the capacity, deadlock and other reasons to be handled. Okay. So the problems with buffers are definitely overflow because if your number of requests are more than the number of buffers available, what do you do with the new request? Because your protocol would say do not NACK them, you have to buffer them. So these extra requests that overflow your input buffer capacity can be moved to the memory as a separate storage. So this idea is used in the MIT Alewife processor. The other idea is when your buffer capacity gets full, instead of sending it into the memory, what we could do is, do not buffer it at home at all and forward all the requests to the owner node.

So they get queued up at the owner and this is implemented by the Stanford DASH protocol. Now you would say they would also pile up at the owner, but the requests which are coming to a home node for variety of blocks, the owners for these blocks would be disjoint and hence the overflow problem would not be as severe as the previous case. Even here, if the block is clean at the home, then it will be serialized at the home node because home would do the service. If the block is dirty, that is there is an exclusive copy of this block elsewhere in the system, then the home forwards it to the owner node. In this case they are queued up at the owner.

So serialization is done by the home for clean copies and it is done by the owner for the dirty copies. So at the owner, if you will say that we forward the request to the owner and in the meanwhile, the owner has for example written back the data block. So home forwarded the request to owner, but owner wrote back the data block and it cannot service this request any further. So what would it do? It simply NACKs this request and once a request is NACKed, the requester would later retry it in the same or a different manner. So this is done by the Stanford DASH system.

What is done by the Origin protocol which we are discussing? We are going to use the concept of a NACK that is the home would go in a busy state and then we will NACK any requests which are coming to it and it is not being able to satisfy them. So what is the order of serialization? The order is the order in which the requests are accepted by the home node. It is not that they are NACKed. Suppose I have, this is my directory and to this directory request 1, 2, 3 and 4, these all come in this order, whereas the request number 2 gets NACKed. So what is the order of serialization? It will be 1, then 3 and then 4, because 2 will get eliminated and maybe 2 does a retry later and that succeeds.

So then 2 will come in this order. So the order of serialization is done by the home node in the order in which the requests are accepted and if it cannot handle a request because it is busy servicing a particular request, it simply sends a NACK. So the NACKed requests get retried and hence we do not need to buffer them. So that is the Origin way of working. The fourth idea is, I have a FIFO buffer, but this buffer is distributed, because if you have FIFO buffer you have problems related to overflow and capacity issues. But if I have a FIFO buffer which is distributed in the form of a link list, then it is easy and this is done by the SCI protocol which is the cache based flat memory directory style design.

So we are going to see this SCI protocol in this module towards the next set of lectures. Alright. So this idea of serialization is, I hope it is clear and we also discussed in the previous lectures that just having one single entity as a serializing agent is not sufficient. Because if this entity knows that my involvement with this request is completed, it does not mean that request related to this particular block are completed elsewhere in the system. Because there could be other requests in the same block which go to some other nodes, they service them and they get replies, whereas this particular entity is not aware of that thing happening. So having a single entity and knowing its involvement getting over is not sufficient. So what do we do? We make sure that all the nodes in the system which are involved in answering or catering to this request, they also should follow certain disciplines. What should they do? They have to make sure that until that request is completed in, on behalf of them, they should not service future request for the same block.

So not only the home node should wait for a request to complete before taking care of further requests in the same block, at the same time either the requester or the owner all have to make sure that the request completes with respect to them as well, before they service request for the same block for future accesses. Okay. So that is what this text in the blue is indicating, that we should not apply any incoming transactions to a block, while we are still waiting to have a transaction outstanding for that block. So not only the owner but also the requester and the home node, everybody should follow this procedure, that unless one request is complete do not service future requests. Okay. So to get a feel of what would go wrong, we are going to see two examples which will show you the need for the second condition. So second condition was that every entity should wait for the request to get complete with respect to that particular node. Okay.

So the first example is we are going to see that a single entity for serialization is not sufficient. Now my serial entity is the home. So this example says that, we have a home node, if I assume this is going to serialize and I receive the request in the order, P1 processor sends a read request for block A. Immediately next is, P2 processor sends a write request for the same block A and the third request reaching the home is, P1 sends another read request to the block A. Because if you can understand, first P1 sent the read request, it will reach the home node, then P2's write reaches, so because P2's write reaches the home node it is logical that P1's block would get invalidated and hence P2 would also want to issue a read request which will again go to the home node. So this is the order in which they reach the home node.

Now let us see that if the home does not wait for a previous request to get over, what can go wrong. The protocol guarantees that home would wait for the request to get over but if it does not we are going to see what can go wrong. Okay. So we will do this example slowly. So I am going to take the home node and the two processors P1, P2. Okay. First request is this one, P1 sends a read to home.

Which sends read A. So this is a request going to the home node. What will home do in this case, if you recollect the Origin protocol, it is going to send a speculative response to P1 if there is another sharer in the system, but right now because there is no sharer there is no intervention required, it simply sends a reply back to P1 with the data. So if this is arrow number 1 as a response, the arrow is going to send the data and I will assume that the data value was A was equal to 2, say for example. So if I say that here value of A is equal to 2 and for writing maybe A becomes 4 in this case. In the meanwhile what happens is P2 sends a write to A with a value of 4.

So it wants to write value 4 to this. So first it sends a write request. Now this write request when it reaches home, home is going to give a reply to P2 with a speculative data and it is going to send an invalidation to P1. So let me name this as arrow number 2 because request 1 reaches, arrow 1 reaches and arrow 2 reaches. Because arrow 1 reached to begin with, home sends the data to P1 in the third step. Now in the fourth step it has to cater to arrow number 2.

So what does it do? It sends a speculative response to P2 as step number 4 and it sends the data with value A equal to 2. I am just writing the data to see the consistency aspect. Okay. Now it also has to send an invalidation message to P1. So home as part of transaction 4, so I will call it 4B, it sends an invalidation message to P1. And what argument I am going to make here is, this message number 3 is slow.

So although numerically it is ahead of 4, so this message 3 which home sent is going through a different path in the network and it is slow to reach P1. So before 3 reaches, 4B has already reached. If I say like this 1, then 1 results into 3 where 3 is slow and then 2 results into 4A, 4B which both are fast enough to reach P1 and P2. As a response of 4B which is an invalidation message, P1 will delete its block and it will send an acknowledgement back to home node. So it sends an ACK back as action number 5 as invalidation acknowledgement.

It sends an invalidation ACK here and when 4A reaches P2, what is P2 going to do? After this, it will simply update the value of A to 4. Okay. Now you imagine after doing all of this, the message 3 reaches P1, what will happen? You can pause the video, think and come to some conclusion and then let us see if it is going to be valid. So suppose this message number 3 reaches now to P1, what would P1 do? P1 receives this message 3 which is containing the data A equal to 2, it updates its own cache and it is done. Because 3 came with data, it took that data, wrote it in its cache. Because of this green action, the third read, that is read A of 2 by P1. So when P1 does a read, what value will it get? It will get the value of A equal to 2 whereas globally or in, if you see the serial order, the value of A which P1 should get is equal to 4.

So why did this happen? Because the message 3 got delayed into the network and it reached P1 late. Okay. So with this scenario, what was done improperly? That is, which entity was impatient to take actions? See what happened here is P1 got a response to its data. So P1 sent read A request as action number 1. What was it expecting? It was expecting the response of data, that is data A equal to 2 should reach to P1. Before that, before the 3, the 4B came to P1 because 4B came to P1, P1 invalidated its data block.

But actually P1 was waiting for its data. So P1 sent a request, it was expecting the value A equal to 2 to come to it, but this never came. Instead invalidation came. Now when you send a request, you are not expecting an invalidation as a response. So when you send a request as P1, you have to wait for the response for your request before you can cater to a new request.

So I will say invalidation is a new type of a request, whereas this one was my old request. So my old request of data should get a response of a data packet or it should get a NACK. Only then I should be ready to cater to my new requests. So P1 did not wait for its involvement to get over to solve this problem. Hence it ended up reading a old value. Right.

So what was the effect of this whole transactions which we saw because P1 did not wait? The end result was that the second read resulted into a stale value. A got the value of 2 instead of 4 which was the newest value. So the effect of write due to P2, so P2 made the value of A equal to 4. So this value got lost with respect to P1 because P1 read some old stale value which reached it in the network little later. So what are the solutions to this problem? One is that which we discussed that P1 should wait for its read to complete.

It should wait for its read to complete, wait for the data or wait for the NACK and only then try to cater to the invalidation which has reached to it. That is one solution. The other solution is that home does not allow P2 because home has to wait for a acknowledgement from P1. Let me go back. Here ,when home sent the data to P1 in this step, before it sent a reply to P2.

So before it could do this, it should not send the data to P2 unless P1 says it has received the data block. So P1 requests, home replies, the data may or may not reach, it does not know and in the meanwhile it sends the same data to P2. So how does home come to know that the P1 has received its packet before it can start catering to P2? Well that can only happen if P1 again sends an additional acknowledgement that yes, I got the data which you sent to me. So if, this was my P1 and that was the home, so P1 sent a request, home sent a reply with the data, this was the slow link and to solve this problem, home should somehow know that P1 has got the data. So there has to be an additional acknowledgement from P1 coming back to home.

So this will work correctly but it is going to violate our strict request response behavior. Because for one request you get a response and that is it. You cannot send an acknowledgement as a third message to your strict request response protocol. Additionally, it is going to cause more buffering, lead to deadlock problems and definitely longer delays to take place. So, if I force home to take care in the scenario that P1 will send an acknowledgement it is going to violate my request response protocol. Hence we cannot take this approach and therefore our solution should depend on P1 to make sure that P1 does not apply an incoming invalidation message, before it receives a response for its previous transaction.

So P1 is a node, it has a request which is outstanding for a block, it does not allow access to another request to apply until the outstanding request completes. So in this case, P1 would do an invalidation only after it gets a reply or it gets a NACK. In any case it should wait for a response before catering to the invalidation request. Alright. So my second example is that, again same thing single serializing entity is not sufficient, so here we will see how the home can handle a particular problem. So scenario here is that, there are again two processors P1 and P2. P1 is holding the block in dirty state and there is P2 who wants to start writing to this block.

So P2 will send a read exclusive transaction to the home. Home is going to send an intervention to P1, P1 will send the data to P2, it will not send the data to home because it is a read exclusive transaction and home can still remain in a stale state. But P1 has to send an ownership transfer revision message to home, telling that home, now P2 is the owner and remove P1 as the owner. So P1 is not the owner, P2 is the new owner, this will be done and P2 gets the data from P1. Okay.

So this is the scenario. End of all this P2 ends up deleting the block which it just now got. Now this deleted block has to be written back because P2 had updated it. It had asked the block for writing, it wrote the data and it has to write it back to the home node. So in this scenario, if the home does not follow certain discipline, we are going to have a problem in the correctness of the protocol.

So let us do this example. So again I have a home node of P1 and P2. Right. And here I will say the home knows that the block is dirty with P1. The block is dirty with P1. The first request is, P2 sends a write request. It says I want to write to the data block A. What should home do? Home knows that the data block is dirty with P1.

So it sends an intervention, sends an intervention to P1 with the identity of P2. So that P1 can give the data to P2 as the next transaction. I will call it 3A. So it gives the data and the value let me say it was A equal to 2 with P1 and it sends an ownership revision message to home, which is my message number 3B.

So I will say this ownership transfer. I am using some short forms here. So it uses an ownership transfer revision message to the home node. Now I am going to say this message number 3 is slow. This is a slow network.

So 3B never reaches home before other things happen. Now P2 after receiving 3A, it does a writing. So it does write A equal to 4. It writes, writes A equal to 4 in that and then it has to evict the data block. So at P2, all these things are happening.

It first finishes writing then it evicts the A. So when you evict A you have to write back A and this write back message goes to the home node as transaction number 4. So write back A and what value will go back? Here, A equal to 4 will go back to the home node. So when message number 4 reaches the home node, home is going to update its content with A equal to 4. Now after 4, 3B reaches. Okay. So if I just write the order for you this is message number 1, then 2, then 3A, these things happen.

Then after 3A, 4 takes place and after 4, message 3 reaches. So when message 3 reaches the ownership transfer message reaches to the home and home says, okay, now the owner for block A is P2. So home node, if I say, initially, the owner is P1. In the meanwhile, there was this P2 who took the data block, made A equal to 4. It deleted the data block and then as a response to message number 3, the new owner becomes P2 because home says now P2 is the owner. So initially P1 is owner, at the end of all these transactions P2 is the owner, but in the meanwhile, in this stretch P2 has already deleted the block.

So now home is pointing to P2 for block A whereas that block is not there in P2. So we have now reached a place where my directory information is corrupted. So why did this happen? This happened because the home node was already catering to the write A transaction from P2. The home node had a pending transaction of write A from P2.

While it was still yet to finish this, it took a new transaction of write back of A from P2.

It started catering to this one without finishing this transaction and that is the reason why our directory information got corrupted. So this illustrates that, what should we do? The home should first wait for its previous transaction to complete before catering to new requests. Okay. So I have written the same thing here. So in the previous slide we did a run through of the example and here is a point of reference for you to clarify. So at the end, when the revision message reaches, the directory makes the new owner as P2, whereas you already know that P2 has deleted the data block.

So what is the solution for this? And why did this happen? This happened because the home did not wait for its involvement to get over with respect to the previous transaction. So there was a previous transaction of write P2 going on and it had not complete. Why was it not complete? Because this message, the revision message from P1 had not yet reached home and home knows that it is going to get this 3B. So it should wait for 3B, before it does an invalidation request from P2. Okay. So this happened because home did not wait for its involvement to get over and it started servicing the write back.

The solution for this is what does Origin protocol do? It prevents this by going into a busy state. So when it is servicing a write P2, it goes into a busy state, that is see this one here. Suppose a write request has come, so this is your P2 sending a write of A and that has made the directory state busy. So when the directory is busy, the new request of write back from P2 will be ignored and P2 will be sent a negative acknowledgement.

So we are going to NACK that request and not service that request. Okay. So that is the reason why Origin uses a busy state so that it does not cater to newer request for the same block ahead of time. Okay. So we need to note down two points here. The first one is there in this orange color. This says that, if P2 sends a write back to the directory and the directory is made busy because of P2 itself.

That is the case we are looking at in my current example. In my current example, P2 had sent a write request. It was not complete and P2 again sent a write back request and we had to NACK this request. Okay. So this orange text is saying that if P2 sends a write back request and P2 was the one who made the directory state busy, then directory has to ignore it and send a NACK. Allright. Because previous transaction is yet to complete.

So we have another scenario where P2 sends a write back. P2 sends a write back to the directory and directory is busy. But here the directory was made busy by some other node and not P2. So here node P3 had made the directory busy. So P2 is sending the data back to directory.

Directory is busy because it is catering to node P3. So should we NACK P2? We should not because P2 is sending a data block and if we NACK that data block, we are going to lose out the only copy in the system and P3 is likely a new reader or writer to that data block. So P3 would want the request and home would have forwarded the request of P3, going up to P2. So this was the case we took up in my previous example where two requests had crossed and what we do is, the write back from P2 we take and give that data directly to P3. So we forward the write back from P2 to P3, whereas in the case in the orange text, the write back is ignored or NACKed whereas in the blue text the write back is forwarded from P2 to P3 because in both scenarios the processor which made the directory busy were different. Okay, so we have looked at serialization and now the deadlock and livelock aspects.

So deadlock aspects, Origin has got finite buffers and if we keep requests pending and if the buffers become full, we could fall back to strict request response type of a behavior. For handling livelocks, we are sending NACKs to requests which are busy and so busy states are NACKed and NACKed would again be retried and the first request which reaches to the processor will make progress. Now would this guarantee starvation freedom or not because I am going to use NACKs, NACKs will be retried and retried NACKs would maybe lead to starvation because there could be some processors which do not get serviced. So for deadlock we are, we keep pending request and fall back to a strict request response method on detecting network contention. For livelock we can avoid by using busy states and NACKing the subsequent requests and they will be retried in future but if I am using several such it can lead to starvation.

So NACKs can cause starvation, how do I solve this starvation? By maintaining a FIFO order among the requests or by associating priority with each request, that is if I am going to NACK a request some X number of times, then we increase the priority of such request, so that they will definitely get serviced in near future. So the number of times I am going to NACK the request, that is the order in which the priority of the request will be increased. So overall the Origin philosophy is that, we are going to have a memoryless system, that is don't maintain any history as such of the local state and don't hold up resources. That is simply when you cannot service a request you NACK the request instead of buffering it to prevent deadlocks. Okay. So we have looked at starvation freedom using priority and overall the correctness aspects are discussed in this lecture. Thank you so much.