Parallel Computer Architecture

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

Week - 05
Lecture - 30

Lec 30: Correctness Requirements

Hello everyone. We are starting with module 4 today and the title of the module is Snoop-based multiprocessor design. So if you recollect, cache coherence can be implemented using either snooping or directory based protocols. So we are going to look into the details of how a snoop-based multiprocessor can be designed as part of this module 4. This is lecture number 1 in which we are going to look at correctness requirements. So when we are designing symmetric multiprocessors, there has been large difference in the performance, cost and scalability of various implementations.

But at the bottom if you see the cache coherence protocols, theoretically they look all similar. Processors also are same. So what brings out this difference in the performance and the cost? So it is mainly how we organize all these components and how do they interact with each other. So the design and implementation and organization of all these components will finally give result to a good performing system.

The latency and the bandwidth at which I am accessing the memory will depend on how your processor interacts with the cache, how the cache is designed, then how the cache interacts with the interconnect, whether it is a bus interconnect or a scalable interconnect and eventually how is the memory designed. Is it a single banked memory or an interleaved access memory and so on. So there are many factors which will affect the final performance of your system and also the end user cost. So when we are implementing, we need to keep track of three main goals of any implementation. One is it should be correct, it should be high performing and at minimal extra hardware.

So your overhead should be as less as possible. Right. So if you keep in mind that our three goals of implementation are correctness, high performance and minimal extra hardware. All these systems are going to implement the same cache coherence protocol with some variations, but how are we designing and integrating these components will affect the final performance cost and scalability of your SMP. So the implementation goals as we said are correctness, high performance and using minimal extra hardware. So correctness arises because when all these parallel components are interacting with each other or if you recollect the cache coherence protocol, we said that when a block is accessed, I am going to send an invalidation message on the interconnect or update message on the bus and we assume sitting here that somebody has invalidated. Right.

So we kind of assume everything is atomic, it happens instantly for us, but in real life it doesn't happen because you sent an invalidation to a node, but that node may be busy doing some processing or that snoop controller is busy accessing something else and it might take some time before that invalidation or the update takes place. So this sending processor, what are the assumptions it is doing? Am I assuming that it is happening instantaneously? Am I assuming that it will eventually happen? So all these will affect the correctness of your final implementation. Because at the abstract level everything happens instantaneously, but in real implementation it takes some amount of time. Again, high performance. We want the system to be high performing, but right now when we discussed cache coherent systems, we said that let one transaction happen at a time, we sent an update and we assumed that the memory will first write that data and then I can proceed with my next read or write action. But if you have a system implemented like this, it will be very slow. Because memory as it is going to take more time and if you are doing one operation at a time, there will be a lot of gaps and many processors will be stalled for doing that activity. Right.

So we want to do things in parallel. Can we pipeline the accesses to the memory? Can I have a bank level memory where all the banks can be parallelly accessed? So all these implementations or ideas if I implement, then it will also have to be guaranteed that they work correctly. Right. So high performance requirement also puts load on your correctness aspects of your system. And then again the cache coherence, there is a lot of interaction between the different caches, the snoop controllers on the bus and the interconnects. So several components.

If you recollect the FSMs which we discussed during the cache coherence topic, there were so many FSMs at each node doing their activity, interacting with the interconnect. So all this is a very complex system and it has to work correctly and at the same time with good performance. So earlier when out of order processors came into market, it was known that the correctness and testing of these were very challenging tasks. So these days, correctness of the cache coherence controllers has also become equally challenging and equally important. Right. So implementation goals of correctness, high performance are to be guaranteed because our theoretical understanding of atomic actions is no longer atomic in the real implementation.

So it is not necessarily atomic at the hardware level. We want to improve performance by pipelining the memory accesses and keeping several outstanding accesses in parallel. Right. So if a cache incurs a miss, we send that request to the memory but at the same time the cache can start servicing other requests. Same thing can happen with the memory that if the memory is serving a particular request on one bank, we can say that let the other banks process some other requests in parallel. Right. So we want several things to be happening in parallel to improve the performance of the system.

When you have such complex interactions between the cache, the memory and the interconnect, we are going to be concerned about the correctness of our system. So definitely the recent cache coherence controllers and the modern out of order execution processors, they both are complex systems and they will have many outstanding memory requests pending and processors will have many outstanding instructions going on together and therefore the correctness of this whole system complex controllers is of utmost importance for us. So what are we going to see in this module? So first thing we are just going to list and understand the importance of correctness requirements in this lecture. Then in the next lecture, we are going to look at how a atomic bus, that is system which works on a bus but the bus is atomic, that is it takes only one transaction at a time finishes that transaction and then takes the other one. So with such a single transaction bus and if the system uses single level caches, how can we implement a snoop based design? We will move one level up and say that I will go for a multi level cache but still use an atomic bus.

Next is we will again go back to a single level cache but this time we will try a split transaction bus which means I can have multiple transactions happening on the same bus in parallel. Okay. Again next level up here is split transaction bus with a multi level cache. So through these four designs, we are going to see in detail how a snoop based system can be developed. Okay. So coming back to our goal of this lecture is the correctness requirements. So in a cache coherent system, to guarantee correctness, we have to make sure that all the conditions related to coherence and consistency are maintained.

So what are the conditions to ensure coherence? So if we can recollect the topic on cache coherence, we said that coherence can be maintained given that whenever a write is done to a shared item, everybody else should be able to see it and nobody should get the stale data items. Right. So if there are other copies in the system, either they get invalidated or they get updated. In the end, all the stale copies need to be found and either updated or invalidated. So that is what we do in coherence. And while doing this, we need to guarantee write serialization as well as write propagation. Okay.

So these are the requirements of a coherence correctness system. Then we also have to guarantee sequential consistency. So sequential consistency, this topic we are going to see later in a different module. But for the time being, I will simply say that we need to cater to two main properties for sequential consistency correctness. So for sequential consistency, we are going to look at or rather understand what is write atomicity and detecting the completion of write.

So if I can guarantee these two, I will say that my system is sequentially consistent. And if I want to guarantee coherence, I will say that all the stale copies are identified, they are either invalidated or they are updated. Finally has the write been serialized and has the write been propagated. Right. So every time we need to check whether all these properties listed here are satisfied or not. Okay. So for sequential consistency, we discussed that we need to prove write atomicity and detecting completion of a write.

So what are these two things? So I will say that a write completes when the write which is done by one processor is seen by every other processor. Okay. So suppose processor P1 does a write of value 5 to B. Right. So it does write B equal to 5. And there are other processors P2, P3 and P4 in the system. So what we are saying is that this effect reaches every other processor.

This write reaches every other processor before P1 is permitted to do another write or another read. Suppose it is also wanting to read something. So it can do any other operation only after everybody else has seen the effect of write B equal to 5. Okay. So when I say write complete, it says that write should not complete and it should not allow the next write to occur until all the processors have seen the effect of this write. Right. So they all should have seen that B has become 5 only then other writes can happen in the system.

So if we guarantee this, we are guaranteeing sequential consistency. Then the second condition, so we have done this one. The second condition is write atomicity. So write atomicity says that after a read operation is issued, the issuing processor waits for that read to complete and for the write whose value is being returned by the read to complete before issuing the next operation. Right. So complicated definition, I would request you pause the video, read through it and try to understand.

Okay, so let us repeat this. What is write atomicity? That is once we issue a read operation, we get some value. Suppose I did a read of, if I said read C and I got a value of 2. Okay. So we would say that my read is complete and after this I want to do something else. I want to do a write to X something. Okay.

So step 1, I did a read. Step 2, I will try to do a write. But I am not permitted to do the step 2 until, until this value C equal to 2 is effectively seen by every other processor. That is I got C equal to 2 from some other processors. This is where, this one, this is the write. So there is a processor which made C equal to 2.

We got the C equal to 2, but we need to wait until everybody else in the system sees C equal to 2 before we can move to my next instruction. Okay, so that is the meaning of write atomicity. And if you see it as at an abstract level, what you will understand is, the processor say P3 which did a write of C equal to 2. If there is a processor P3 which did this write C equal to 2, this write has atomically happened for everybody.

Atomically means instantaneously. Actually in the real world, it is not happening instantaneously. But how do I guarantee this logical instantaneous behavior in a real system? By making sure that whenever anybody accesses C, they all should see C equal to 2 before they can do the next action. Okay. So this is how we have implemented or we can say that we have managed to implement write atomicity in a non-atomic real world system. Okay. So in other words, we have written here in the bullet point that if the write whose value is being returned has been performed with respect to this processor, then the processor should wait. Right. It should wait until the write has been performed with respect to all the processors. Okay.

So if I am getting the value of C equal to 2, then the processor which wrote C equal to 2, that write should be propagated to every other processor before I can say that my C equal to 2 is complete. Right. So for write atomicity, I will just take this example. Here, we have written these two values and suppose I am concentrating on this write. Write C equal to 2 was done by P1 and now P2, P2 reads C.

So it is definitely going to get a 2. But are we saying that this is complete? Can it go to read B? It cannot. It should not go to read B until it is somehow sure that P3 which is the other processor in the system has also seen C equal to 2. Okay. So the write of C equal to 2 in P1 should reach everybody before P2 can proceed to read the next item. Right. So P2 actually waits for P3 to see C equal to 2 before it can proceed to the next instruction. So it can only do read B after P3 has seen C equal to 2.

So that is the meaning of write atomicity. Okay. Then the next correctness requirement is the system should be free from deadlock, livelock and starvation. All of you are very familiar with this, but let us do a quick recap of what these three terms mean. And apart from these things, if there are any other error conditions in the system, our system should be able to control them or get rid of them in a proper manner. Okay. So deadlock, so deadlock means there are several outstanding operations, that is there is work to do, but I am not able to do the work.

That is the system activity has ceased, it has stopped, everything is stalled in spite of the fact that there is work to be done. And why does this happen? This happens because there are multiple entities in the system. Everybody is incrementally acquiring resources, like you are standing and you are acquiring books from a library, your friend is also acquiring books from the library. Both of you have got five books, but you want a book which your friend is holding and the friend wants a book which you are holding, but none of you is ready to share the book with each other. You are saying you first give me the book, then I will give you my book and you only have capacity to hold five books. Right.

So in this case, even if there is work to do, you still want to read the sixth book, you are not able to read the sixth book because you are not willing to relinquish one of your resources and you still want the new resource in your hand. Okay. So when multiple concurrent entities hold incrementally more resources and they are not going to release them preemptively. Right. So it is a non-preemptive system, so we cannot progress in this scenario. So deadlock is outstanding operations and system activity has ceased. Multiple concurrent activities, entities are there, they are acquiring resources but not releasing them.

So thus there is a cycle of dependencies. So there is a cyclic dependency between the system components and we are going to see an example what happens at a traffic light junction and in a hardware system. Okay. So at a traffic light junction which is seen on the right hand side, here we have all these cars waiting at the junction. Now these cars are waiting because how can I say what resource it is acquiring? This orange car has acquired the lane, if I say this is lane 1. Okay. It has acquired lane 1 and I can say this has acquired and it is asking for the lane, green lane if I call it. So it is holding the orange lane and then it is asking for the green lane. So this wants to go straight, so it is saying I want access to this straight lane and I am currently holding the horizontal lane.

So it cannot go further because the red car which is standing here, it is holding this lane, if I call this lane 2, it is holding lane 2 and asking for access of this lane. Okay. So it wants this lane. So there is a circle of dependencies. What is blue car doing? Blue car is holding this lane and is asking the grey car to move out and the grey car is holding this lane and wanting the orange car to move out. So there is a cycle among them because none of the cars is willing to back off and they both are saying I am going to stand here, you move back and if everybody says this you will get a cycle and hence the system will be deadlocked.

Why deadlocked? Because the work is not over and we are not progressing. So this can happen in a traffic light intersection. What happens in a hardware system? So in the case of hardware when I am discussing two controllers A and B, both of them are holding certain resources and they want more resources from the other one. So A says that I want a resource from B and only then I will accept requests from you. Right.

And B says the same thing. B says I want A to give me something, only then I will give something to A. So this way I have a cyclic dependency between A and B because none of them is willing to preemptively release their resource and accept the outstanding requests. Then the next scenario is live lock. So this is opposite of deadlock but even here the system makes no forward progress. Right. So here the processors make no forward progress and actually there is lot of activity happening here.

So if I give you an analogy at the traffic junction, all the cars in the deadlock scenario were waiting still. Right. So they were all standing but in the live lock what we are going to understand is all these cars will decide that they want to release the resource. So they will proactively move back. Right. So they will all move back giving way to the other cars to proceed. But coincidentally what happened is all of them moved back to the same time.

Then they saw that okay the lane is free, I can move forward. So they coincidentally again moved forward together. So we are again in the same situation and this can repeat forever. They all back off at the same time and then again come in front at the same time.

So this is the situation of a live lock. So live lock is when there is no forward progress happening but there is lots of activity going on into the system. So in the traffic analogy they all back off and then again try again at the same time and repeatedly they do this without making real progress. So there is no real progress in the system. What happens in the hardware? Hardware you can have independent controllers. They are holding some common resources but it the situation is that they snatch the resource from each other.

That is I have controller A and B and there is a common resource. So A says I want the resource, it it gets the resource. Therefore it can consume this resource, B pulls the resource back towards it and then A pulls it back. So you can say that in the context of cache coherence. Okay. In cache coherence if a processor sends some request to the memory or to another processor, let us say P1, P2.

So we can use this example. We send an invalidation request from P1 to P2 for a particular data block. So P2 says okay I have to invalidate this block. So it invalidates.

Then P1 gets this block B. Then P2 again wants this block B. So it sends an invalidation. So this is invalidation by P1. After a while invalidation from P2 goes onto the bus. Okay. So when invalidation of P2 reaches P1, P1 has hardly even accessed the block B but again it has to relinquish that block for P2. So this way the block will keep on doing a ping pong between P1 and P2 and none of them will be able to utilize the block.

So once we request ownership of the block by invalidating others and we lose the ownership immediately before having finished usage of that resource. Okay. So here if you understand there is lot of activity on the bus and the system but none of the processors is able to progress. So this is the example in hardware scenario where I have controllers A and B. They are using a common shared resource.

So A pulls the resource from B. So it pulls that resource and when A gets the resource B thinks I want the resource. So B also pulls it back from A. So this resource keeps on doing a ping pong between A and B and none of them is able to use this. So that is the situation of a live lock.

The third scenario is starvation. So here starvation means there are one or more processors which are making no forward progress. So there are one or more. It is not all processors. Okay. Because if all processors make no progress then it is either live lock or deadlock. But here among several processors or several modules certain modules are not able to make progress.

Let us take the traffic analogy. Here how do I solve the live lock? So to solve live lock what you could do is give priority to some direction. So there are cars coming in all directions and what we can say is give priority to the northbound traffic. Okay. If I give priority to this what would happen is all the other traffic or we can say the eastbound so this one, this traffic has to back off because the northbound traffic has got priority. So this traffic backs off and the northbound traffic can continue. But if there is heavy traffic what would happen is the eastbound traffic would never get a chance to cross the junction.

The other example is you have a busy highway, a busy highway connected to a tiny countryside road. Right. So if you have this scenario the highway always has vehicles passing through and hence the countryside road traffic will never get a chance to cross over. So these are the scenarios of starvation where the countryside is starved and here the eastbound traffic is starved. In hardware what is the example? We can say that I have an interleaved memory system where I can send a NAC.

NAC means a negative acknowledgement on a bank being busy. Okay. So I have these four banks and requests keep coming to them. Okay. And several requests come. Suppose this bank is busy, it will keep on sending a negative acknowledgement and it may so happen that the NAC which is going, it keeps going to the same processor and that particular processor will never be able to finish its memory access. So this is the example of starvation in a banked memory system. Another example in a bus based system is on the bus if a certain processor gets stalled of accessing the bus or going on to the bus to the memory, how should we solve this? So here it is rather easy because I can use a FIFO of a queue, that is I queue up the requests of everybody. So all the requests are queued up here hence at some point of time everybody will get a chance to progress.

So I can avoid starvation using FIFO type of queues in a bus based system. What happens in a scalable systems? That was the interconnect is not bus but a scalable interconnect. So here solving starvation is more complex but the philosophy used in this cases is that instead of solving it we believe that the scenarios or the situations of starvation are very less that is there are, they are very less likely and even if they occur they are not catastrophic. So they are not going to create a big harm and eventually the starved module will get a chance to progress. Right. So we want to eliminate adding the complexity to the system and hence in most of the cases apart from the bus where I can use a FIFO in complex interconnected systems, I am going to ignore the aspect of starvation because it is not catastrophic, it is not permanent and the effort required to solve that is really large. Okay.

So this is the concept of starvation. So we have seen in this lecture the various correctness requirements and so with this we finish this lecture. Thank you so much.