Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 06
Lecture - 32

Lec 32: Single-Level caches with an Atomic Bus (2)

Hello everyone. We are doing module 4 on snoop-based multiprocessor design. This is lecture 3 which is the continuation of lecture 2 on single-level caches with an atomic bus. In the previous lecture we have seen the requirements we need to add to uniprocessor cache to make it multiprocessor cache coherent friendly. In this we are going to look at the organization that is the actual design, placement and other things and the correctness aspects. Okay. So we will immediately start with the organization. So I will give you a big picture here on this slide and the detailed design will be given on the next slide as a ready answer. Okay. So we will do it slowly and you can pause the video in the middle, try to connect all the components of which, which I am drawing here with the previous lecture. Okay. So base organization first thing we have a cache, processor cache which is connected to the processor. Okay. So cache is connected to the processor.

Then we discussed that we have two tag arrays. All the things which we discussed are going to be now placed into a design, okay. So, we have duplicated the tag arrays and this tag array I will say is for the processor and the second tag array is for the bus, okay. Then we have the FSM for the, the processor side FSM. Then this processor side FSM manages this side.

Then I have the snooping or the bus side FSM. So, we have the bus side controller which will look at this side. It is going to access that tag and the other things, okay. Now when a request comes from the processor, it will compare the tag and then it has to send the request onto the bus.

So, we need a bus. I will draw a bus at the bottom and the request from the processor will come. So, this FSM is going to send the command that is whether the processor wants to read or write the data item and then it will put this on the bus. And when it puts this, it also has to put the address. So, the address will come from the tag array and the original address.

So, it will combine and put the address onto the bus, okay. So, this is how the processor interaction happens. So, when you put the address and the command onto the bus, you are going to get an answer. Suppose you are reading the block, so memory which is connected somewhere here will give the data, okay. So, now that data comes and you have to store the data.

So, the data comes and sits here into this data buffer. So, this is coming like this and once you get the data, you directly send it to the cache, okay. So, this is the interaction by the processor. Now what happens on the bus side? Now the bus side controller is essentially the snooping controller. Now the snooping controller has to keep on checking what happens onto the bus and every time some address or the command is coming, it has to keep track.

So, any command which is coming from outside, any address which is coming from outside that is going a BusRdX or a BusRd which is going onto the bus, they both come, we store them here, okay. And then these two will be now used for comparison because do we have the data or not, we have to check. So, these two go here. So, I am drawing an abstract picture, okay. So, these two will come here, this tag which is a copy for the bus side controller.

So, this will be compared. Now along with this, if you recollect, we had a write buffer and this write buffer was storing the write back blocks from the cache which will eventually be sent onto the bus for the memory, okay. So, whenever we write back data, it is put into the write back buffer which is then connected to the memory. So, this buffer has multiple, probably multiple blocks and we also need to compare that, okay. So, all these 1, 2 and this 3 and 4, all these are used for comparison.

So, we are going to compare all of this and then decide to give the snooping result. So, the snoop answer is coming from the comparator and then put onto the bus, okay. So, this is the overall picture of the organization. So, the red part, except the bus controller, so the red part is what was existing in a uniprocessor design, whereas the blue, green and the bus side controller is what we have added, okay. So, this is a neater picture of the same thing, okay.

And here we list all the components in case you want a ready reference. You can keep checking this with the previous slide, we have a write back cache, we have dual tags, then processor side controller which checks the address and command, a write back transaction which goes via the write buffer, then the read transaction goes via the data buffer, we have controller on the bus side which compares the tags, then bus arbitration. So, definitely I did not draw this, but there is a bus FSM which will arbitrate for the bus, if it gets the grant, it will do the required transactions and so on. Finally, the snoop results which will come from the comparator will be put onto the bus. So, these snoop results will act as an acknowledgement to the initiator.

So, when we put snoop results onto the bus, so this when we put this onto the bus, you all understand that this answer of the snoop will help decide whether memory gives the data, cache gives the data and how should the requester behave, okay, okay. So, it also helps in my proof because when an act comes we can say that we have seen this request that is whichever cache was supposed to take action would have taken appropriate action as part of the snoop results, okay. So, this is the organization and then for this organization we are going to look at these various correctness requirements which we will do them one by one, okay. So, before we go to the correctness requirements one small thing we need to take care that happens because of this. That is the bus arbitration and putting the request onto the bus is not happening instantaneously but there are several small steps which are involved, okay, right.

So, we when we draw the protocol FSM we say that this is one state we put an arrow and then we go to the next state. So, if there is a bus read, from bus read we immediately go from an invalid state to a shared state or the state E, okay. So, if I say this is I, I can immediately go to S, okay. So, we just draw this as the theoretical protocol but you are not sure that when you send the request onto the bus whether you got the bus or not and definitely there are many intermediate actions which are to be done, okay. So, one is that you send the request onto the bus then the other coop controllers are going to do a cache lookup.

Then it your local controller you are going to do arbitration you may get the bus you may not get the bus even if you get the bus when you send a request to the next cache it has to do its own lookup, take appropriate actions that is each cache controller will have to take appropriate actions and then finally respond with their answer. So, this is going to take time, okay. So, other cache controllers will work then they will put the answer onto the bus, accordingly the sender will decide and finally, the block will be written or read by the requester, okay. So, when you are doing several of these actions definitely lot of race conditions are going to take place, okay. Now, we will see an example, that suppose I request a block B, okay and then request for a block B is coming to me, but even I want to change the state of block B.

So, this process of P1, it is working it is doing some work with B. In the meanwhile P2 says I want to work on B, okay. So, there is a race between P1 and P2. So, we will look at that example, okay. So, we have process of P1, P2 and both of them have the block A and now P1 says I want to write to A. So, it will send a bus upgrade, this is a variant of the MESI where instead of BusRdX, we send a BusUpgr if we already have the block.

So, this is the state of the block in both the processors, this one also sends a bus upgrade. Now, both of them are sending a request to the bus, but only one of them will get, okay. So, now you can realize that I cannot directly go from here to here just in a single step. So, P1 and P2 both contend for the bus, but P2 gets the bus here. So, P2 succeeds and P2 moves to state M.

When P2 moves to state M, P1 has to go to state I, it has to invalidate and remember this is still pending. I mean we can say that this was a pending work it had to do, but now it has to relinquish the block and subsequently again retry, right. So, this pending work had to be retried in step number 3 and in step 3 eventually it will be able to complete, okay. So, now this has to be handled by our controllers, okay. So, the same thing is written on this slide, okay.

Moving on. So, for handling such race conditions we need to do some changes to the finite state machine. So, for ready reference I have copied the MESI FSM onto the slide. We have seen this in detail in the previous module. Now, this is the FSM and I want to adapt all these race conditions or be able to handle those race conditions. So, I am going to do some changes to this.

So, we can go back and forth for the time being we will directly start adding the states, okay. So, I have just noted them here and we will try to add new states to handle those race conditions, okay. So, we have state I, the block is not present and when the block is not present we can either get a bus read or a bus write request. So, when we get a bus read request, bus read request what did we do? We went from I to S, okay. We went from I to S and on a bus write we went from I to M and depending on the shared wired-OR we could also go to E, okay.

So, let us look at I to S or I to E transition. So, from I, if you get a processor read we normally send a bus read and directly go to S or E, okay. But now if you look at the race condition example you may or may not get the bus, okay. So, just demanding bus is not getting the bus. So, I have to first request for the bus.

So, I am going to now have two small things split into this. Sorry, once you get a processor read, you send a bus request then you may or may not get the bus, okay. So, when you do this you have to go to some state. So, from this square state if you get a bus grant, we will send the bus read signal because it is a read request and then check for the S or S bar and depending on that we will either go to state S or we will go to state E, okay. So, now I am not directly going from I to S or E. I am going from I to the square state and from the square that is this box I am going to E or S.

So, this state I am going to rename this or give some name to this. This says I am going from I to either S or E. You can give any name, but we are trying to be as close to the implementation. So, we are saying from I, I am either going to S or E. So, I am going to name this square box as this particular notation, okay.

Let us do the write part, processor write. When a processor write comes we send a BusRdX, but now we cannot directly do that first we have to request. So, let us send a bus request. So, when you send a bus request again I am going to another new state. From here you will get a grant or you may not get a grant.

If you get a grant you go to bus to the state M. So, bus grant send a BusRdX and then go to M, okay. If you do not get a grant then you will keep waiting there for the grant to come, okay. So, these are two states what should I call this? This is on the way from I to M. So, I can simply say I arrow M for this box, okay.

There is one more variation that says if I am in state S, I will use another color for this. If I am in state S and we get processor write request. Because from state E if you get a processor write you directly go to M because there is nothing other sharer in the system to be informed, but in S you have to inform others. So, when we are here, we again need to send a BusRdX, but I cannot do that now. I have to first send a bus request. So, when I send a bus request I need a square box, okay. So, we will go to a square box from here and then if you get a grant you will go to M, okay.

Once you get a bus grant we will send a BusRdX and then go to M, but if we do not get a bus grant, okay. Let me call the state as S arrow M, okay. When I am standing in S arrow M this square box, in the meanwhile here because the block is shared it is possible that some other cache requests this block, okay. So, from somewhere else. So from elsewhere you get a some request for the same block, okay that request comes and now if it is a BusRdX what does this request mean which is coming from outside? It says that somebody wants to write to the block. So, we have to remove this block and where are we right now standing here, okay.

So, because of this green request which comes, we would have to go out of this box and invalidate the block, okay. So, now the block will get removed whereas we were assuming that we are in this intermediate state waiting for the bus grant. So, while you are waiting for the grant, our block was removed and next time when you retry, your retry cannot begin from here because this state represents that you already have the block, okay. So, you cannot retry from here but you have to retry from a state which is remembering what you are going to do. I can either go to I or I can go to a more useful state which is this.

So, what I will do? I will go to a more useful state which is this because now once I see a BusRdX which is the green transaction coming, I have to invalidate the block, probably also give the block if it is cache to cache sharing and then go and stand in this red rectangle I to M, and once we go there, we know that we do not have the block and we are probably going to eventually move to M. Okay, so, this is how we have added these new states to the MESI protocol. Even here this one, instead of BusRdX because we already had the block you could have sent a BusUpgr, okay. So, you can also send a BusUpgr here, okay. So, these are the intermediate states, also called as transient states.

So, apart from the cache organization we have also added few functionalities to the MESI protocol, okay. So, again a neater diagram is given here. The changes we have done is we have added three states 1, 2 and 3. Three transient states have been added which are shown dotted in that figure, alright. So, this slide is showing the complete picture integration of the MESI as well as the transient states.

Previous slide only highlighted the changes whereas here it is a complete picture. So, you can cross refer to the pure MESI only protocol that is MESI states only and these three new added states 1, 2 and 3. So, I had shown them as rectangles in my explanation, but they are shown here with these highlights. Now, these transient states need to be called something. So, we call them transient or intermediate states and the other as stable states, okay.

So, we have two names, stable states that is the original MESI 1, 2, 3, 4 these are stable and then the three new which we have added as transient. You all understand it is going to add complexity to your system and this problem would come if I am using bus upgrade. So, if I am using bus upgrade let me go back to this one, okay. So, I said that here when we did the design we said if you get a grant you can either send a BusRdX or a BusUpgr. But when you send a BusUpgr, it creates more issues because you assume that you had the block. But in the meanwhile if somebody takes away your block then the BusUpgr which you are taking the action which you are taking gets translated to a BusRdX when you have to flush the block in the middle, okay.

So, when you are moving from S to M, if some other cache wants the same block you, invalidate your block and when you invalidate your block you can no longer send the bus upgrade, but you have to send the BusRdX instead of BusUpgr. So, your FSM has to take care of this. It has more complexity that initially it was supposed to send BusUpgr, but now suddenly it has to change its decision and send a BusRdX. So, this adds more complexity and therefore some implementations say I will never use BusUpgr, let me always be consistent and use a BusRdX, okay. So, yes, so I am not going to use BusUpgr, but only BusRdX. Because I do not want to change the decision of whether to send this or to send this depending on which intermediate state we are standing, alright. So, this is how the non atomicity that is the bus having several other steps before we get the grant, is handled by the protocol by adding more intermediate transient states.

So, we have seen how to add transient states to the MESI FSM to handle the non atomicity and next with this base organization and all the additions which we have done, we have to check whether all our correctness requirements are satisfied. So, what were the correctness requirements? They were cache coherence correctness, that is write serialization, write completeness, write atomicity and then deadlock, livelock and starvation freedom, okay. So, we are going to see whether all these properties are satisfied by our designs. We have made the design and now we will check whether it satisfies them, okay. So, we are going to quickly do a recap of how bus orders reads and writes.

So, these are the four processors, color codes say the request coming from the particular processor. So, three are already ordered by the bus, next P3 sends a write. So, this is going to invalidate that block across all and then P3 does some reads and writes. So, they are drawn little above because they are local hits, okay. So, here this will invalidate whether these are write hits within P3.

So, they will not go on to the bus, but subsequent reads writes will go on to the bus by another processor, okay. So, these are following the program order and a read by P2 will now go on to the bus and this is going to get the value written by this one, okay. So, this is how the bus is going to order the reads and writes, okay. So, for write serialization that is the writes get serialized in my current design where I have a single level cache, atomic bus. So, there may be conflicts that is race conditions or there may be no race conditions.

So, if there is no race condition, what happens? So, we have processor P1,iIt wants to do a write x equal to 2, it wants to do this and presently the state of the block is S. It has the block in shared state, maybe there are other processors sharing it P1 wants to write, but what it would do? You would say that why not write, but we have to go to the M state for writing, presently we are in the S state. So, for doing that the protocol tells us first send a BusRdX or an BusUpgr on to the bus, okay. So, first we are going to send this and when you send this all the other processors will snoop, when they snoop the bus they will be seeing that write to x has happened, okay. So, write to x is seen by everybody because of the snooping controllers of P2, P3 and P4.

So, once they have seen this, the snoop results will come on to the bus, right. So, the snoop results will come and once the snoop results come, P1 will know that yes everybody knows about x being changed it will go to the state M and then actually it will perform the write, okay. So, x will become 2 only after it moves from S to M. So, that this is when there were no conflicts in the system, okay. But now if there is some competition happening, right, there is a race condition.

Look here P1 also has the block x and P4 also has the block or wants the block, okay. Both of them are going to race with each other. P1 has in state S and P4 does not have it, but wishes to write to it, okay. Now P1 says I want to write to this block.

So, first thing it tries to send the bus request, okay. At the same time P4 also sends the bus request, both of them send the request at the same time, but P4 gets the grant because P4 gets the grant it will send the BusRdX and the BusUpgr and this will result into, what will this result into? It will invalidate the block in P1, okay. So, P1 has to remove the block and then go to state I instead of S and once this finishes P4 will move from I to M. So, in this slide we have seen a race condition where both P1 and P4 wanted to write and P4 gets the bus. So, P1 has to relinquish its block and this is how the protocol will manage that exactly one of them gets the chance to go onto the bus and we know that the bus is going to serialize the writes, okay. So how do we establish that serialization is taking place? So, processor and cache must have a handshake which preserves the program order, right.

So, processor and cache are going to manage that properly according to the program order and once they go to the bus, the bus transactions are going to serialize. So, once it reaches the bus, bus will serialize all the reads and the writes. If we have the block in state S, it is very tempting that we start writing because the block why not write it, okay. But we cannot start writing until we get ownership of the block. Now, why are we imposing this condition? We are imposing this condition because if you recollect this race condition, P1 had the block.

It could have written directly the value of X, but you never know that another processor might want the same block and that processor might get access. So, if P4 got the access, it ended up writing X and if P1 also had modified X, it will result into an inconsistent state of the system. So, P1 even if it has the block, it is not permitted to change the value until it gets the bus transaction completed, okay. So, the temptation of writing is not permitted before acquiring ownership, okay. Because if you do this in case of conflicting transaction that is a race condition happens, you will have an inconsistent system state, okay.

So, if other transaction comes and it completes, your answer will be inconsistent. So, we do not permit this to happen and if we maintain the proper order that first go onto the bus and then change, we will be able to serialize all the accesses, okay. So, we can say that write serialization is established by properly following the protocol, going through all the intermediate states and not writing the block even if we have the block in hand earlier, okay. So, everything which appears on the bus will get correctly serialized by the bus because the protocol says that even if here, if you see here, because the BusRdX happened on that block, P1 had to remove its block even if it had the block, right.

So, the order was first X equal to 4 and then X equal to 2. So, this is the serialized order of the writes in this example and it will be correctly established if we follow the protocol properly. That was write serialization. Now, write completeness is required for sequential consistency. Completeness says that whenever a write is happening, everybody in the system should have seen the write, right. If P1 changes X to something, it makes sure that everybody has seen X equal to 2 before it starts using that X or doing any other action.

So, that is write completeness. So, write completes with respect to all the caches, okay. So, now we will say that how strict is this? So, if I change X equal to 2 here, I have to tell everybody that I am changing X and this is a long process, okay. So, should I wait for everybody to invalidate and tell me that they have removed it or can I do it little early, okay.

So, we are going to see a small optimization in this case. We are saying that the cache need not wait until all the invalidations finish, okay. We do not wait for them to invalidate because maybe those caches are busy, they will take some time to invalidate, but we should have somehow guaranteed that they will invalidate the block, okay. So, how can we guarantee this? We are saying that I am using an atomic bus. When the transaction goes on to the bus, we are sure that every cache is going to follow it properly and invalidate.

So, they are going to listen to us. That is the underlying assumption. So, in this way, we guarantee that they have seen the write, okay Although they have not invalidated, but because they came to know that a BusRdX has happened, they will sincerely delete the block from their cache without doing any further changes to that block. Hence, we can say they have seen the write. So, once they have seen the write, we can start updating, okay. So, our base design, what does it do? It puts a transaction on to the bus and once it puts an invalidation transaction that is a BusRdX, the BusRdX will eventually go to others, invalidate, get an acknowledgement and so on.

So, we need not wait. So, once we get a bus grant and we send a BusRdX, we can say yes, the work is done. That is we somehow assume that there is a commitment from all the processors to invalidate. So, there is this concept of write being committed, it has not complete, but it has committed to happen, okay. So, we are assuming a commit instead of a complete. So, write completeness is replaced by write commitment, because as soon as I send a bus transaction, the invalidations will eventually take place, but they will definitely take place before the next transactions happens, okay.

And we are not bothered about the, in the remote processor, where is that invalidation put in its local order because these will be local hits, right. So, the local hits and misses do not appear on the bus. So, I am not concerned about the local order. I am only concerned on the bus order and I am sure that this invalidation will have definitely finished before the next bus transaction, okay. So, the protocol guarantees that it will first finish the task only then the new transaction can take place.

So, this way a commitment of invalidation implies a completeness of the write, okay. So, that was write completeness. Next condition is write atomicity which says that once you have written a value, everybody sees the same value across all the processors, okay. So, a read returns a value of a particular write means that the write transaction is gone onto the bus and everybody has seen it, okay. So, we have seen that with respect to if I prove serialization plus completeness, okay, these two things are kind of implying my write atomicity requirement.

So, write atomicity is also guaranteed, okay. Now, one small side note about writebacks. What about writebacks? But we say that writebacks can happen anytime because there are no sharers for such blocks. They can go anytime onto the bus then nobody is interested in seeing them because there are no sharers. So, nobody sees them, memory gets updated. Now, the question is what about reads to such things? So, I had a writeback going onto the bus, writebacks for x and if there is a read x coming. Now, what happens to this new read x? Now, this new read x will go onto the bus and definitely this time the memory will provide the data for this and because the writeback has updated the memory.

Okay. So, the writebacks which go onto the bus although we are not interested in the serialization, but we can definitely say that the future reads will get the most recently written value from the memory, okay. The next requirement is deadlock. We are not talking of the deadlock like which happened in the case of buffers where two controllers are waiting for each other's buffers or the traffic light junction, but in protocol we are discussing about a protocol level fetch deadlock, okay. So, this is a special kind of deadlock where I have a controller, suppose this, which is saying I want to send something onto the bus, okay.

So, it has sent a bus request and it says until I get a grant I will not do anything. I will just keep waiting here for me to get the bus. In the meanwhile, some request comes onto the bus to the same controller. It says I want some help from you. So, but this one says until you give me the bus I want to do anything. So, you can see this results into a deadlock because the blue one says I want the bus and the orange one is not ready to respond.

So, nobody is releasing the bus and hence they both keep waiting. So, this is called a protocol level fetch deadlock, okay. Why is this happening? Because I want to issue a request, but I am not willing to respond to the incoming request, right. So, if there is an incoming request, I ignore it whereas my outgoing request I am only interested in sending my request outside. So, this creates a problem. I want to issue it, but I am not servicing, but to come out of this, we need to start servicing the incoming request.

So, even if the blue one is pending, I should be catering to the request coming onto the bus, okay. So, cash controller is waiting for the bus grant, but at the same time it must snoop that is snooping should happen in parallel and if you snoop you may also want to give data if you have the block, okay. So, your, the request which you want to send outside, they may be pending, but you should be ready to send responses to the outside world at the same time, right. So, you should be able to cater to this otherwise there will be a protocol level deadlock, okay. Example, so here P1 sends a read exclusive for A, it has in S it wants to go to M state, okay.

And P2 says I want to go to M state for block B. So, both of them want different blocks, okay. And it so happens that P2 gets the grant and it sends a request for B and if you see B is in state M with P1. So, P1 has to give the data to P2, but P1 says I want access for A, until you give that I will not move ahead. So, this will create a deadlock. So, what should happen is once P1 sees that a read X on B is pending it should ignore or it can keep this pending, it need not keep waiting here, but cater to this request, okay. It should flush the block B onto the bus so that P2 can finish its work and eventually P1 will get a chance to go on to the bus, okay.

This is how while our requests are pending we should be ready to keep servicing requests coming onto the bus, we will avoid the deadlock. Live lock happens so when multiple processors want to change the same block, everybody wants to change the same block they will want go on to the bus, one of them will get the access, they will move from I to M, okay. So, the block move from I to M, but while the processor was about to write, that is the processor and the cache will now try to write to this block, but in the meanwhile another processor comes and says I want this block back, okay. So, the block moved here and before it went to the processor, before it could go there, the block was taken away by this again, okay. So, we kind of have a ping pong effect of the ball or the block moving from one processor to the other, okay.

So, if there is a simultaneous access this processor has to release it and then again there will be a cache miss. So, we have a cache miss, we go on to the bus, we get the block, before I change the block somebody else takes away the block from me, okay. So, lot of activity, but no forward progress and that is what live lock means. So, how do you solve this? So, to solve this we need to, this is the solution you have to design the cache and processor interaction properly. This portion, you have to say that once I get the block until we finish the write I am not going to service request on that particular block.

So, if this is implemented in the processor FSM sorry the coherence FSM then live lock can be avoided. So, live lock is avoided by implementing a proper processor and cache handshake. Last requirement, starvation, now starvation happens when multiple processors are competing for the bus right. Everybody wants to go on to the bus. Only a few of them get and it is possible that only selected few get the bus grant and certain others are stopped. So, for a fair solution that is to avoid starvation I need an arbiter which is fair or I can use a FIFO where the services are queued up or first come first serve type of implementation policies. So, to solve starvation we can have fair arbitration FIFO queues and FCFS scheduling, okay.

Now, for FIFO you will need buffers. So, this might be expensive. So, certain implementation say, I will have no buffers because starvation does not happen very often, but I will have counters. So, whenever you do not accept a request from a processor we can keep a counter with that processor saying that I have refused this processor n number of times and then check for a threshold. So, once you have refused a particular processor say n times then after that you can give a priority to this processor, okay. So, this way we can avoid starvation, right. So, with these all discussions we have established that write serialization, completeness, atomicity, deadlock, livelock, starvation are possible and correctly implemented in our discussed design, okay. So, with this we have completed the single level cache with atomic bus. Thank you so much.