Parallel Computer Architecture

Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 07
Lecture - 38

Lec 38: Path of a Cache Miss

Hello everyone. We are doing the module number 4 on snoop based multiprocessor design. This is lecture number 9 and in this lecture we are going to discuss path of a cache miss. That is in detail we will see all the steps which a particular transaction goes through before the processor finally gets the data block. Okay. So, the overall contents of this module are these and we are still on point number 4 single level cache with a split transaction bus and this is the last lecture in this particular fourth point. Okay. So, moving on path of a cache miss is the topic to discuss.

There are two types of misses. Path of a cache miss is what we are going to discuss and there are two types of misses, handling a read miss and a write miss. So, first we will start with a read miss. So, we all understand that read miss means that the processor wants to read the block and the block is not there in the cache.

So, when such a block is not there it sends a bus read transaction onto the bus. In a normal single core system we go out from the cache to the main memory, bring the block and start using it, okay. But in a cache coherent multiprocessor system, we have to use various transactions to finally obtain the block. So, we are going to send a bus read transaction and right now we are dealing with a split transaction bus. So, when we send a bus read we are not expected to get the response immediately, okay.

So, what happens? Overall when I send a read miss transaction onto the bus, an entry will be made in the request table. But before that am I permitted to put this transaction, has to be checked. As we said we cannot allow conflicting transactions, but mine is a read, is there another write pending? If there is then we cannot issue the transaction. If there is no write pending, but if another processor is also wanting to read and that particular request is still in the pending queue then probably yes, because I want to read that, processor wants to read. So, both can get the data in the read form because sharing is allowed for read content, okay.

So, what type of things I need to do so that I can also grab the same data when it arrives. And the third case is, when there is no pending request on this particular block. So, we can go on to the bus and then put the request. But while doing this also we need to take care that just by chance has another conflicting request arrived in the meantime. So, this is how we are going to handle the read miss, okay.

So, we will start with the bus read, right. So, read miss we are going to use the bus read transaction. While doing this before we put this transaction on to the bus we will check the request table and inside the request table, is a request for this particular block present or it is not present we have to check that, okay. If the request is present then we have to take certain types of actions, if the request is not present that is there is no other processor already issued a request on this we can issue the request, but still watch out for any race conditions that, that may arise in the meantime, okay. So, prior bus read exists, that is when we issue the bus read, there is already a pending bus read transaction for this particular block.

So, this table on the slide is the request table belonging to processor P1. The three entries are various transactions BusRd, BusRdX and this originated by different processors on these particular data blocks, okay. Now my processor P1 sends a BusRd on A, okay. So, when we get a BusRd on A we will compare all the addresses that is we are going to compare this, this and this and check whether A exists, okay. So, if A is there we have to check the type of request, then once we get a hit here we will check the request type which is bus read in this case, okay.

So, we get a hit in the request table the request is type of bus read and what do we decide, that P2 wants to read the block A and P1 also want to read the block A. So, why not both of them can get the data block whenever the response happens, alright. So, we should queue up ourselves in the entry number T1, okay. This can be done by adding two more information bits to the request table which say that whether we want to grab the data, okay. I will set this bit to 1 in my case because P1 wants to grab the data.

So, it sets it to true and is P1 the originator? No, it is not, okay. So, this bit is 0. So, with the help of these two bits, what will happen is, in whenever the response comes P1 can also grab the data, okay. So, we can grab the data when the answer comes on to the bus, okay. So, these two bits are there, in my processor P1 they are set to 1 and 0.

What would the bits be set in processor P2, if I ask you, okay. So, think and find out what are the bits for the request table of P2, because of P2 wants to grab the data. So, inside P2 the bits will be, the first bit grab data will be definitely 1 because it wants the data and the originator bit will also be 1 because it is the originator, okay. Now you will say why do we need the originator information also because end of the day everybody is going to grab, alright. Now suppose the current scenario is for P1. In case of P3 also comes and wants to read A.

So, in case of P3 the bit will be it wants to grab, but it is not the originator. Even for P1, we had said it wants to grab it is not the originator. So overall we have three processors which will grab the data when the answer comes, okay. Initially only P2 had asked for the data that is the first transaction which took place. When the response comes P2 will consume the data.

Now whether P2 should load this block in state shared or state exclusive in a MESI protocol. So, we need to decide this and it will load in shared provided other readers are there, it will load in exclusive if there are no other readers. So, how is P2 going to know whether to load in S or E because these things happen later in time, after P2 had issued the request, okay. So, we have to make sure that when the result comes, P1 and P3 will proactively say that yes, we are grabbing the data, hence all of us should load into the shared state, okay. If P1 and P3 were not active, only P2 will load in the E state, okay.

Now how do I manage this? Okay, so for this we will say that the non-original grabber that is the processor which is going to just take the data, it is not the originator, will raise the shared wired-OR line, okay. So, the S signal the shared wired-OR signal which was there, it will be set to 1 by P2, sorry P1 and P3 in my example, okay. So, both of them will make the shared signal 1, this will guarantee that the originator will not load it in E but in the state S, so state S is loaded because of this arrangement, okay. So, in the picture you can see P1 is the new requester, it wants to grab, P2 is the originator and so when the data starts coming onto the data bus, P1 will make the snoop shared signal equal to 1, making both of them load it in the shared state. So, this is about bus read.

Now a prior BusRdX transaction exists, now BusRdX means a conflicting transaction is present. If we see here, this one. And now my P1 processor is asking for a BusRd on block B, okay. The block has changed for block B. We already have a T2 transaction which is a type BusRdX originated by P3, what should we do? The design specification or what we have decided says that you cannot allow conflicting transactions. So, we cannot allow this to go on, right. Here the two bits are not much of a significance because anyway this transaction will not get issued and we have to cancel this request because of the conflict transaction existing in the queue and we will retry this transaction later. So, when do we retry? Whenever T2 completes.

So, after T2 completes, we can again try to issue this new transaction. Now the third case is when there is no pending transaction on that block. So, now P1 says BusRd on B but there is no entry for block B into the request table so we can move on, good. So, we will say there was a request table miss hence I can issue the request on to the bus, okay.

So, before issuing the request we have to request the bus that is arbitrate for the bus and maybe we are unlucky and we do not get the bus this time, okay.

So, P1 goes out to issue BusRd but does not get the bus. In the meantime, processor P3 wants to do a BusRdX on B. it wins the bus and adds T2 entry. So, see now the table has updated like this. So, T2 has suddenly appeared and P1 had not seen that T2, okay. P1 did not see T2. It went to the bus, did not get the bus, so it is waiting for the bus and in the meanwhile P3 comes and puts the T2 entry, here. So, can P1 move ahead with the BusRd because it is already standing at the bus it had never seen T2.

Well, P1 is not aware of this but we know that we cannot permit this. So, how to implement? So, here even after that you have decided that you can go on to the bus, just before you get the bus that is or even when you get the bus grant, you check the requestable entry once more because it is possible that in the meanwhile a conflicting entry has already occurred into the requestable. As soon as you see that T2 is conflicting, we cannot cancel the request because we have already got the bus. So, what we do we essentially only put a null transaction onto the bus and do not do anything further and again retry. So, this will say retry later.

So, we can retry about this after some time. Okay, so eventually P1 will be able to put its transaction onto the bus. Okay. So, even if we see that there is no pending request, so we need to take care of race conditions because even if there is no conflicting transaction at the moment but before we get the bus, a conflicting request may be granted by the bus and therefore we have to be careful and check the requestable once more. Okay. So, if we find such an entry, issue a null request, and withdraw from further arbitration and retry later. Okay, so now we were succeeded in putting the BusRd or BusRdX onto the system.

Now what is its effect on the other processes? Okay. Till now we were discussing P1 wants to send something onto the bus. Okay, so P1 wanted to send a request onto the bus, BusRd,BusRdX. There were conflicts, there were no conflicts, race conditions. So, all three cases we have handled. Now once this transaction goes onto the bus, what happens to the other processes or what actions do they take? Okay one action is that there is a sender which has to provide the data be it BusRdx or BusRd somebody has to provide data to P1. Now who will provide the data? It will be the memory, by default if there is no other cache which has modified data or it is another cache which has the data in a modified state.

So, one of these two. Now when the transaction goes onto the bus, each of these processor cache controllers will do the snooping inside and they will check the state of the block in their local caches. Memory definitely doesn't need to check, it already has the data. So, memory can decide, let me start fetching the data to be sent as a response. Other caches will start comparing the data or comparing the state of the block and one of them may decide that I have the block in dirty state, so I should provide the data. Now there are two parties wanting to send the data. One is memory which is always proactive and one is this cache which wants to send the data because it has a dirty block.

Now two responders are there and they have to again win the bus for data response. Two possibilities here. If the memory wins the data bus to send the response, memory will start sending the data which it should not have been because the cache had the dirty copy. So, how do we stop this? So, even if the memory starts sending the cache which had the dirty data should do something to stop the memory from doing this. Okay. This is easily done by using the snoop signals, that is we have the inhibit signal and we have the dirty signal. I am going to use these two to stop the memory from sending the block. And the other cases the cache having the dirty block is successful in getting the bus.

So, it sends the data and the third case is that there is no cache with the dirty data and hence the memory has to send the data. Okay. So, we are going to see these three cases. So, these are the three scenarios we are going to consider. First scenario, cache has the dirty block and cache wins the bus. So, this is good situation because the original or actually the authentic sender has got the bus.

But in the meanwhile definitely memory would also have started loading the block because memory did not know whether cache will give and memory takes more time to load data. So, it starts fetching the data. In the meanwhile, the cache wins and it sees that the block response has already started from the cache the memory aborts its fetch. So, it just stops fetching the block and does not do anything.

Okay. So, the cache controller was able to get the bus and it wants to send the data. Seeing this memory does not send the data. There is one more case to think that there are other cache controllers which may not have finished snooping. That is if I say, P1 is the one who asked for it. I can say P3 is the one who wants to give the data.

But P2 is a slow processor which has not even finished snooping. Okay. So, it is still snooping and finding out what it should do with this current request. And because it is slow, it has to give the answer of the snoop. So, it extends the snoop transaction by raising the inhibit line.

It says that wait, I have not finished my snoop. So, it raises this inhibit line, finishes the snoop and then disables the inhibit line. So, this can be easily extended as the part of the request and response transactions. Okay. So, when another cache controller is not ready it can use the inhibit line.

With respect to memory, we have to take care that if the memory's buffer is full. Remember that the cache has dirty data. It is sending the data onto the bus for a BusRd transaction. This implies that when this updated block is going the memory, is expected to take this data, so that it has the updated copy. Now if memory's buffer is full, we have to do the flow control by sending the NACK signal. So, memory says okay, data is coming onto the bus but my buffer is full. So it raises the NACK signal which will make the cache with the dirty block, retry in future.

Otherwise everything goes smoothly, cache puts the data onto the bus and other caches memory pick up the data block. Okay. Second situation, cache has the dirty data but memory wins the bus. Now this is a problem because we do not want memory to send the data. Now what should we do? The cache which had the dirty block may have finished snoop or may not have finished snoop, but in any case, before it finishes snoop it would assert the inhibit line.

It will complete the snoop. While doing the snoop it understands that it has the block in the dirty state so it raises the dirty line. If you recollect we had three wired-OR signals, the shared, the dirty and the inhibit. Okay. Shared means somebody has a read copy, dirty is a dirty copy and the inhibit signal says, wait until I have done S and D.

Okay. So the cache controller raises the inhibit, completes the snoop, asserts the dirt,y and then releases the inhibit. So during this the dirty line is raised, so memory will see the dirty line and it has to cancel putting the data onto the bus, because it understands some cache has the block in dirty state. Okay. So memory cancels sending the block but that particular phase is occupied by memory. Even if it does not put the data, the cache which has the data has to again retry for the response phase.

Okay. So it has to again arbitrate for the data response and then put the data onto the bus. Okay. So it will acquire the bus later and then send the data. Third case, cache has no dirty block and memory has to give the data. So very straightforward.

Memory acquires the bus and sends the response. In the meanwhile, if any cache has not finished the snoop, it will keep the inhibit line raised until it finishes the snoop and then makes the inhibit zero before the data transfer can begin. So when inhibit is back to zero, memory can start putting the data and then the caches can grab it. So here you must have realized that we are not making the shared caches skip the data. If the block is clean everywhere, only the memory gives the data and hence cache to cache transfers are not permitted in this particular SGI protocol.

Right. So that was read miss. Now write miss is when processor wants to write the data. So it will put a BusRdX transaction. Whether to reach the bus or whether there are conflicts about this particular transaction, all those steps are same as in the read miss. So nothing new here to discuss. So once the BusRdX transactions goes onto the bus. So for a write miss, the BusRdX goes onto the bus and when the response comes, if the response comes from memory, no problem, directly take the data. But if the response comes from a dirty cache then that cache has modified the data. The current requester is also going to modify the data. So when the data transfer happens from the dirty cache to the new requester, memory is not going to pick up the data block because anyway it is going to change. Right.

So that is the small change in this write miss transaction. The third type is BusUpgr which is taking place when the cache has already got the block but it wants the writing permission. Now there is no response for this and so the bus controller itself sends the acknowledgement and in case there is another cache which would have sent a BusUpgr for the same block, it would have to translate its request from a BusUpgr to BusRdX because this particular BusUpgr succeeded. Okay. So these are some minor details. But overall write miss is similar to the read miss in terms of the actions to be done. Only thing is that the dirty data when it goes memory, will not take the data and definitely no other cache will also grab it because it is not permitted. Only one can do the writing, others have to invalidate and this is how we will handle the bus upgrade. Okay now proving write serialization. So we have discussed the example path of a cache miss read and write miss.

Now we have to prove write serialization, completeness, atomicity, and variety of other properties. So write serialization is that, are the writes to different locations serialized? and are the reads which are coming after these writes getting the correct value? So if I have a read x, first I read a read x and then I did a write x. So will the new write of x affect the value the previous read has established? okay, it should not because the read should have ideally finished before this write happens. Now is my current setup guaranteeing this? So we can say that yes, it does because we do not allow conflicting transactions. So when the read x is a pending transaction, a write x to the same block cannot take place and hence when the write x takes place, the read would have finished and the write will not affect the read. Okay, same argument holds for the reverse case. When I have a write followed by a read, what should this mean? that the read x should get the value written by the previous write and this is again possible because we do not allow conflicting transactions, that is until write x finishes read x will not be sure.

Okay. And BusRdX and BusUpgr, for these there are invalidations sent to all the processes. So now what about them? now when these invalidations are queued up. Do we permit them to be queued up? Should we allow the writers to start writing? So we can say that the read cannot happen until the invalidations associated with that particular block or all blocks are actually done. So therefore, before every read, we have to check whether the invalidations have finished or not. Okay, so that was write serialization. Now write completeness. Okay, so if you recollect the previous discussion we had said I am going to replace the completeness with a commit. Commit means that once I get into the bus there is a commitment from all the processors that they will invalidate the block. And because of this commitment, the sending or the original requester can proceed with the writing. assuming that others will invalidate but these invalidations may still be pending in a split transaction bus. right. So if you can visualize that several of these will be piled up and they will be done eventually. So can we say that commitment of write guarantees that the new value will be visible. So does it guarantee completeness? So actually difficult unless we put some restrictions. Because multiple operations are outstanding that is they are already buffered they are not finished and we are saying that just because you promised that you will invalidate, I will start writing that does not help here, because if we start changing and that particular processor also attempts to read the same data which I am modifying, then we will lose on the coherence and consistency property. Okay. So what can we do to solve this. So I would say that the processor should not be able to see the value. We have buffered several things. So buffering changing the cache comes later. But our main concern is the processor should be protected correctly, processor should see the correct value. Now it can see the correct value if I do not allow reordering of these buffered transactions, that is all these pending transactions are there in this queue and I will say that you cannot reorder them that is if this is done, do not allow this to finish, let them finish in a particular order of appearance, okay. So we do not allow reordering, if we do this then I can replace commit for complete and the other is I can allow reordering but try to ensure some more conditions on that, okay. First option we will take, do not reorder, that is all the transactions on the bus go from the cache to the processor, right, I mean bus to cache to processor in the FIFO order and they will be seen in the FIFO order, okay. So everybody sees the value in the order in which they have appeared onto the bus there is no reordering. Although responses are allowed to be reordered but we will make sure that, all these inside the processor they will go in the FIFO order. So processors will be able to see this data block when the invalidation is applied and invalidations will be applied in that particular order. I am going to maintain invalidations. Hence if I say, I did a write A and then I did a write B we can say that the order is maintained because the invalidations happened in the correct order, okay. While doing this if a response comes to a data block ,that is we have not done invalidations but a data block comes, should we allow this data block to overtake the invalidations and update the cache? no, we should not. So make sure that the data never overtakes the invalidations. First apply the invalidations and then if applied I mean if the data is still relevant use the data otherwise you cannot use the data. So in pending invalidations have to finish, before data replies can be consumed. So do this and then see if the data reply is useful, okay. So if we do this then commit can still replace complete. Next option is do I allow reordering but take some extra care? Here the idea is that I will allow reorder but make sure that the processor always gets the fresh value. Let the request come in any order, I will reorder, I will affect the cache, or may not affect the cache, that does change the data of the cache. But when the processor comes to read a particular data item, I am going to check the queue. Okay. So because the queue might have some blocks. So A, B and C suppose these are the things and processor has come for B, but we don't know the order in which this has finished. So when the processor comes, I am going to check this pending queue, and take appropriate action before servicing the processor's request, okay. So proactively we come and check what is there in the pending queue, so that processor always sees the correct values. So how do I implement this idea? So if the processor wants to access a value and that particular entry is already in the pending queue, we haven't finished its invalidations yet. So we will check the queue and use the queue's information to act upon the processor's request. So processor wants and depending on the queue's information. The queue may have the data, so you give the data. The queue may say that this block has to be invalidated, so you refuse the processor to use that value, okay. So when the processor asks, you check, give the data or you invalidate that block at runtime. So at runtime you quickly do this depending on the processor access pattern. The second option is, we don't keep it in the pending queue, we may write it to the cache. But whenever the processor wants it, so before the processor wants, don't directly answer to the processor, service it to the cache and then let the cache answer to the processor, okay. So if I say here, when the processor comes, it goes to the queue effectively and the queue will reply to the processor, okay, in an abstract sense. Here when the processor asks, you go to the queue, then the queue goes to the cach,e and then the cache replies to the processor, okay. So in short, this is how the idea is, okay. Then other properties right atomicity. Now this is easily provided because of the broadcast nature of the bus, that is when the transaction goes onto the bus everybody has seen it, okay. So bus implies that the rights are committed in the same order, correct they are committed in the same order by all the processors and the reads cannot see the value produced by the write until the write has committed with respect to all processors. So this is perfectly happening with respect to the other or the previous slides arguments which we said, that commit can replace completeness. The same arguments if they hold then we can say, a broadcast based atomic bus can also guarantee write atomicity, okay. So write atomicity falls through very easily in a broadcast medium. The other three properties deadlock livelock starvation. So these are yet to be done. So these last three properties. We will do when we discuss the multi-level cache with a split transaction bus because the arguments are similar, okay, okay. So the three properties of deadlock livelock and starvation, we will discuss when we do multi-level cache with a split transaction bus. We will quickly wrap up with two small points, that is if the responses come in order so how am i going to do that instead of out of order, okay. So when the responses are coming in order, we have a FIFO of a request table instead of a fully associated table. And the request will come and sit in the FIFO and they will be answered in that particular order, okay. So if several requests come, the top request which was the first request needs to be serviced. So therefore, the cache which was supposed to service this request has to take extra care, use all the snoop signals and provide the data proactively. Because until it provides the data, the other requests cannot be serviced, okay. So they are going to be serviced one after the other. This particular constraint puts a performance limitation, that if you have a banked cache and suppose a, b and c are the blocks you want to access. Because of the serial order, you might have to wait for certain accesses even if the banks are free, okay. So for in-order responses, I am going to use a FIFO queue. The dirty cache has to proactively send the response because other requests will be only serviced later. It issues a performance and latency aspect here because if this is the order and c is possible to come quickly, but we have to wait for a to finish then b to finish and then only we can do the c. What happens if i have conflicting requests in an in-order? should i allow? you would say yes, why not because now the responses will go in order. So one two three, so if this is write for x and even this is for write for an x, I can permit both of them because they will finish in that particular order, okay. So we are going to see this example, P1 writes, P2 writes. We can permit this because responses will come in that particular order, right. So P1 sends a BusRdX for a, P2 also sends a BusRdX for a, one by one. So when P1 sends BusRdX, P2 would have invalidated its block. So P2 invalidates. But it has already put its request. So this has already happened. After this happens or in parallel, the invalidation happens for P2. So after this when P2 sends the BusRdX, when this transaction goes onto the bus, P1 sees this transaction. So P1 should invalidate its block? yes or no? It should not, because P1 is using the same block and it has yet to receive the data. So let it receive the block, use it itself and only then it will relinquish the block for P2, all right. So eventually P1 will take the block, modify the block and then flush the block onto the bus for P2 and invalidate it, all right. So we have P1 followed by P2, both want to write, so P1 first takes writes and then short circuits the block to P2. Because that's how we are going to optimize, that is I am not going to make P1 keep the block or if P1 had invalidated it when seeing P2's request, the block will keep on doing ping pong between P1 and P2, okay. So to handle this, these are the steps, right. So with this we finish the topic on split transaction bus for a single level cache. Thank you so much.