Parallel Computer Architecture

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

Week - 07
Lecture - 39

Lec 39: Multi-Level cache + Split transaction Bus

Hello everyone. We are doing module 4 on Snoop-based multiprocessor design. This is lecture number 10. Here we are going to discuss the split transaction bus connected to a multilevel cache. So the overall contents of this module, we are at point number 5 right now, that is multilevel cache with a split transaction bus. So what is the key problem here to handle? We have already taken care of the split transaction bus.

Now when I have a multilevel cache hierarchy, you may have L1, L2, probably an L3 cache in each processor. And whenever a bus transaction goes onto the bus, the bus snooper is going to see this and it has to act upon that. That is, does this processor node have the block in shared, owned, modified, whatever states and take appropriate action. So how would the bus snooper know? It has to ask the cache which is closest to the processor.

If it is a single cache, no problem. But if there are multiple levels of the cache, this check has to propagate up and up until all the levels are crossed. Okay. So you have to go all the way up to all the caches to find out what is the snoop state. And definitely this is going to take a lot of time. And again the correctness aspects and how do we allow multiple outstanding requests, all of these need to be handled.

Again we need to take care of deadlock, write serialization and many other aspects in this because the invalidations will take a lot of time to happen. Now while handling this, we have to make sure that we still need good bandwidth because multiple requests can be allowed to process individual units. That is the processor nodes are going to take their own time to process because of the hierarchy and other pending work at their respective nodes. And therefore there will be lot of queues and when there are queues, deadlock and serialization issues need to be handled. So we will take an example to understand how a multiprocessor cache hierarchy would function.

So here I am saying the processor on the left hand side sends a read request and it is satisfied by the second processor. So the numbers are showing the order in which things would happen. So processor sends a request to cache L1, it misses in the cache. So when this cache miss occurs, L1 has to ask the block to L2. So it puts a request inside this queue.

So all these are queues. Okay. So this is a queue or you can call a buffer. So it puts this request inside this buffer which is going towards L2. Now when it reaches here, it is an L2 miss also. So L2 wants to request the block, so it is going to send a BusRd transaction. Right.

It puts a BusRd request here which will eventually go on to the bus. And you can keep recollecting that by doing this, the request queues will get updated, everybody would snoop and appropriate actions will take place in the background. All right. So this BusRd goes and this cache snoops it. It has to identify whether it has this block. Okay. So that snoop request or I can say the BusRd for the particular block say A, gets entered here.

It comes to this L2. L2 has to check. It knows that it has the block A and the block A is present here, it is a hit, but it is in modified state. Okay. L2 will have housed it in modify but stale copy. It will be modified but stale because L2 may not have the updated copy.

L1 will have the data A in the M state. So L2 has to request L1. So here the request propagates up which says L1 you give me the data block. So that data block comes here as a response. So this is the response which is the data block for A.

So data block of A comes and sits in this buffer number 5. Here it sits here. Then it is taken by L2. So that is response from L1. Okay. Response from L1 goes to L2.

L2 updates its own copy and then puts the data response into this buffer which will eventually send the data of A. Okay. So data of A travels onto the data bus. Now this reaches here and this snooper of P1, the left hand side processor will grab the data. It takes the data of A. It writes the block into L2. Okay.

It is an inclusive cache and this is a response to this particular request. Okay. So this is this response and therefore L2 has to give an answer back to L1. So L1 sends that answer to, so, pardon me, because a pending request is there when L2 gets the data it hands over the data to L1. Okay. So you can see how these various buffers in the middle are utilized and how the hierarchy proceeds. So these buffers which we have inserted, we definitely cannot have a direct wire connecting L1 and L2 because L1 might want to send a request to L2.

So L1 wants something from L2. L2 also wants something from L1. At the same time the bus wants something from L2. L2, L1 want to give something to the bus. Right. So every time there is lot of give and take and the order is not guaranteed.

Hence we need to buffer these things pending before the answer is available. And answer is going to take time because of multiple cache levels. Because you have to travel through all this. You see the path from number 1, 2, 3, 4. It took 8 logical steps to finish this work. Okay. So it is going to take time and until then the request, all the responses need to be buffered on the way.

And now when there are buffers, there are deadlocks possibilities. So how do we handle the deadlock? We have two types of deadlocks to consider. One is a fetch deadlock and the second is a buffer deadlock. So fetch deadlock as we had discussed is a two-phase request-response protocol level problem. When there are requests sent and a particular node is sending requests out and it is also responsible to respond to other requests. Okay. So I want to send something here, but I am not prepared to reply to requests which are coming in the opposite direction.

So I am only waiting for others to serve me. So that is not allowed. If this is done, then you end up into a fetch deadlock. So solution to this is when you are waiting for your request to get serviced, you should be servicing requests for other transactions. Okay. So incoming transactions need to be serviced, otherwise we will have a deadlock. Okay.

So all the incoming things need to be buffered while we have outstanding requests. We have requests but you also buffer the incoming request responses and take appropriate action. Now when I say buffer, how much buffer do we need? So if I do a rough calculation, suppose I have P processors in the system, every processor might send one request. Okay. If we allow only one request per processor, exactly one per processor and then it is expecting one reply. So the buffer space it needs is approximately P plus one size. So I need, if I have 10 processors, I need 11 size buffer everywhere in the worst case because I would assume that I need to house responses and requests from everybody else to my node. Okay.

In case the buffer is smaller, then you have to do flow control and definitely you can use a NACK to solve this problem if you do not have P plus one size buffers available to you. So this is how fetch deadlock can be solved by servicing the requests which are coming to you even when your requests are pending and use a proper size buffer or a good size, good method of flow control. The second type of a deadlock is a buffer deadlock. Here we can see a scenario. I have taken a part of that diagram between L1 and L2, we have these buffers.

Here L1 sends a request to L2 and L2 is sending a request to L1. And both of them have only one position of buffer and L2 says you give me the data, L1 says you give me the data and none of them is ready to provide the data to the other one until its request is serviced. Okay. So this way we have a circular dependency causing the deadlock. Because the L1 to L2 is filled whereas the L2 to L1 queue is also filled with the request and they are both are waiting for responses from each other, without servicing another's request because the buffers also have no space. So how do we do this? Okay. So solution for this is we need to increase the buffer capacity, we cannot have just one. So increase the buffer capacity. This will add up to your hardware real-estate. Each request may need two outstanding buffers.

Why two? Because one is the request itself and one is the write back that is the data block which it might generate. So if you are saying one request, you have to store the actual request as well as if it is a write back request then the space for its data also. Now that is for one. If you have large number of outstanding requests and they will come from the bus side. We will say that, let the processor have only one outstanding request but from the bus side many outstanding transactions will come to you and you have to buffer all of them. Okay.

If you see here, I can have limited buffer. Suppose I can also limit the way we handle the processor side. But from the bus side many requests will come and you will need a huge buffer capacity to handle the incoming request. So what do we do? Either you give enough buffers from the bus to the cache, if you can afford it. Otherwise you simply limit the number of buffers and use standard deadlock avoidance methods to solve this problem. Definitely we need buffers, separate buffers for request and response. Otherwise you would have issues related to how easily you can handle the fetch deadlock problem. Okay.

So in the case of bus transactions, if I allow multiple cache misses to go out and bus transactions are coming in. Right. So if I have incoming request to me are 10, that is from the bus you are getting 10 requests coming to you. And your cache here is sending out some n number of requests. Okay. So n are going out and 10 are coming in. All of them get buffered and now I am assuming that n is greater than 10, that is number of bus transactions is less than the number of pending cache misses. So cache says, I have these many misses pending. The bus should service them for me and the bus says, I already have 10 maximum transactions I can handle.

They are already pending so I cannot service any more requests from the cache. So this is a type of a deadlock because cache misses will be waiting for the bus and bus transaction is waiting for this particular cache to service. Okay. So we should be able to say that this cache services the bus transactions, only then the request table will get free because here the request table has got 10 entries and all these entries are full. The cache miss which is getting generated wants to put a new request, the 11th request cannot go in and already this cache is supposed to serve one of these 10 requests. So it should service one of these 10, so that its request can get a chance to go on to the bus.

How to do this? We need support of bypassing the responses and the request. So I, if I have request, response, response, request, all queued up inside a buffer, I should have some way of bypassing the response on to the bus even if my request cannot get the bus. Okay. And this is going to help us solve many problems. Okay. Now proving write completeness. Here the arguments are same as earlier where we said that I can safely replace the commit with a complete and in the case of multi level cache with a split transaction bus, we definitely want to replace completeness with commitment. Why? Because there are multiple levels and it is going to have more latency.

The scenario is multiple levels, more latency, hence the problem says that you have to use commit and not wait for completeness. But again all the arguments which we used for split transaction bus with a single level cache, they all hold in this case and hence write completeness can be guaranteed. Okay. So next we are going to discuss about having multiple outstanding requests from the processor. If the processor had a single request, we were able to handle multiple levels of cache with one request coming from processor. But it is not true in modern processors because all these are high performance and hence they do out of order execution, out of order completion.

They also use write buffers to hold the writes when not to stall the processors. So we make the writes go into the write buffer and release the reads and write to the processors next request and so on. So lot of things are happening in parallel, many things are pending and out of order. So how do we think of serialization here. And we are saying that, until a write is serialized onto the bus, it should not be made visible to others. At the same time ,we are saying, let the writes happen to the write buffers let them happen out of order. So to make these two things go hand in hand, we say that, let the processor writes go to the local cache, and let it write into the local cache, don't stall the processor until you get the ownership, modify the local cache contents and when you get the ownership only then you can respond to request on this particular block. Okay.

So inside the cache, you have the variable x, you keep on updating it in the local cache and when you get exclusive ownership for this or your BusRdX is committed onto the bus, future accesses to the x, then you can flush these answers. So overall I do not wait to change x but I permit x to change in the local cache. And do not make it available to others until we get the ownership. So that is the idea here of optimization. The other option is instead of changing in the local cache, I will write them in the write buffer and whenever I get exclusive ownership only then I will utilize the writes from the buffer and update the cache or provide it to other caches in the system. Okay. So with this my processor is not stalled, it is permitted to update the local cache temporarily until it gets the permission to write or update the write buffers again temporarily until exclusive ownership is obtained. Right.

So again, writes to the same location can happen in succession, that is we keep on writing and these writes go to the write buffer, so that the processor is not stalled and same problem that I am issuing multiple writes to the same location, should I continue doing this and I have not got the BusRdX permission yet. Okay. So we could do this, provided my block is in state M. You already have the block in state M. So keep on changing, accumulate all the new updates and then service them onto the bus together. Okay. So here the cache must process all these writes which are coming in the modified state from the write buffer and before we service the request onto the bus. Okay. So when bus says give me the data, the cache controller makes sure that all these small small updates which have taken place onto the block, probably they are sitting in the write buffer, take them all, apply onto the block and then send this block as a response onto the bus. Okay. So this way the write buffers accumulated values can be correctly utilized. All right.

So again multiple writes to the same block are allowed if no other memory operation for that block occurs, right, in the program order. So even here we can say that you accumulate all the writes in the cache, if not in the write buffer, you can also accumulate them in the cache and make the entire sequence visible at once. So variables could be different but the same block gets changed and we know that this block is getting changed and there is no other block in the program order which is going to be affected. And hence we can permit such changes and then make all the changes visible to the system once we get exclusive access. So multiple outstanding requests from the processors can be handled this way if we are able to preserve the program order, if we have the correct state of the block. So when we are allowing multiple outstanding transactions or requests from the processor, what about serialization and consistency. Okay.

So who should guarantee that the writes get serialized? If I say the processor should do this because it issues the request, you imagine the complete hierarchy, that the processor goes to the L1 cache then to L2 cache then to the bus, eventually it gets the bus, and then the responses. So it is going to take a lot of time and we cannot afford that the processor waits all this while. So we can say that, I outsource the requirement of processor waiting to the, to somebody else. Now this entity can be the write buffer or the reorder buffers in my out of order processors and the associated controllers. So I do not say that the processor waits because it is going to be inefficient method of handling and let the write buffers and reorder buffers take the responsibility of waiting and handling serialization issues.

So what do these buffers do? They take the charge and make sure that all the write operations which are happening, the processor is writing but they are not made visible to the memory or the interconnect until the correct serial order is happening. Okay. It also applies to local writes. So if the current processor also tries to read, certain things which are there in the write buffer but they are not yet serialized with respect to the bus, it does not respond to the processor request. Okay. Processor is anyway out of order. So if a particular request is not serviced, it will move on to the next instruction. So it is not going to slow down the processor but it is going to guarantee correctness.

For example, if there is a process variable X which is there in the write buffer, it is not yet serialized onto the bus and the processor wants to write it again or read that variable again. but it cannot allow this read to happen because it is there in the pending queue until the correct order is happened onto the bus. The write buffers and reorder buffers will make sure that even the local reads cannot complete. Right. So for a multilevel cache with a split transaction bus as you can now see that we need deeper buffers at every level. We have buffers between L1 and L2, between L2 and the bus and there are requests going up to down and down to up and then responses. So lot of things are happening, many buffers are there and so you need to handle deadlock. To handle deadlock we need to make sure that I have separate request and response queues, so that they do not mix up, we can still provide responses even if the requests are pending or if you have a same queue then you should be able to bypass the responses from the request, that is if you are supposed to reply, that is respond to a request, you make sure that you send the response even if your requests are pending.

So if you do this we are able to handle deadlock in a neater way. Okay. Now when I have a processor with multiple outstanding requests, all these outstanding requests will go to the cache of the processor and this cache should be able to handle them. Now how should the cache handle multiple requests that is request for block A, B, C, D, all are going to the cache and if one of them is a cache miss. Suppose I send request A then I send B then I send C and if B happens to be a cache miss. So it will wait until the B is serviced and only then C will be allowed. But I am saying let A, B and C happen all together. Okay. So if I am doing this then I have to make sure that my cache is lockup free cache and not a blocking cache.

So with this we have finished the topic on multilevel caches with a split transaction bus. The main problems to handle is good buffer management, good fetch deadlock management and making sure that all the write commits will eventually guarantee write completeness and atomicity. Thank you so much.