Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 06
Lecture - 35

Lec 35: Split transaction Bus

Hello everyone. We are doing module 4 on Snoop based multiprocessor design. This is lecture number 6 in which we are going to discuss the split transaction bus. These are the main contents of the module which we are discussing. We are at point number 4 where we are going to overall discuss a single level cache connected to a split transaction bus. The previous two topics were related to an atomic bus.

In this topic, we are moving to a split transaction bus. So, I am going to cover this over two, three small lectures that point number 4. So, we start with the split transaction bus. So, before that let us understand what a split or an atomic transaction means. Okay.

So, for fun I have just created an example where there are three customers who are visiting a restaurant for eating and they have ordered some food. Right. So, almost all of them reach together and the person in the blue shirt places the order. Right. So, the waiter takes the order, goes to the kitchen of the restaurant and then waits for the order to get ready. The order comes and then it is given to this customer number 1. Then customer number 2 which is in the green color again places a next order, but then this order gets delayed. Right.

If you can see here, there is a delay inserted in this order because maybe a complicated dish was ordered. Right. So, then eventually the order is ready and given to the customer 2. What happens in the meanwhile? Customer 3 in the red shirt is unhappy. Why? Because his order is never taken because of this one by one or atomic way of executing the transactions. Right. What is atomic in this? That when one transaction is issued until that is completed, other transactions are not started.

That is the blue customer is serviced and the green one and then eventually the red customer will get the service. Okay. So, in this way we cannot have a best, best system performance because there will be lot of delay in the execution and customers will have to wait. Now if I change this scenario and want to go to a split transaction moving out from an atomic transaction. Okay. Now here again we have the same scenario. All three place orders one by one and these orders are then given to the back kitchen. Right.

Then of course, they are going to take their time to complete the items and then the items get ready and given to the service person. Now this service person has to give the items to the respective customers. So, what does this person think? He thinks that yes, my first customer gave me the first order. So, whatever is the first dish I get from the kitchen I should give it to the first customer. So, he gives that red color dish to the blue color customer and the blue dish to the green customer and the green dish to the red customer.

Now what happens in this? All get wrong items and they are all unhappy. Now why did this happen? This was okay till here because all of them were happy that their orders were taken, but problem started when they got the final answer in the wrong format. Now this happened because the service person had no clue that which order belong to which customer. So, how do I solve this? When I have several transactions happening in parallel I need to somehow keep track that which answer belongs to which question. Okay. So, we are going to now give labels to these transactions.

So, now when the orders get ready at the kitchen they come with the label of the customer ID. Okay. Now this helps the service person to handle the case very nicely. So, here the dishes come with the labels of the customers or the table numbers and then the service person takes the red dish gives it to the customer 3, blue to this one and green to the correct one. Right. So, now everybody has got their respective dishes and for them the order does not matter because the red person is also very happy because that person got the dish quickly. Right. So, in this scenario everybody is very happy.

So, this is the idea I am going to import now to a bus. So, you can keep on drawing the analogy with this example that now till the previous topic we had an atomic bus. Now atomic bus consisted of variety of steps before you can finish one transaction. So, what were the steps? We arbitrate for the bus, then we are granted the bus.

Once we get the bus grant we put the address and the command and once this goes on to the bus the responder whoever suppose the memory or another cache has to give the data. So, they will start the data transfer. Okay. So, these four things happen one after the other and until all four are finished no other transaction can finish. Right. So, this is the model in the restaurant example, until the first customer is completely serviced the order of the second customer is not taken. Okay. Now we move to the split transaction where multiple orders can be taken one after the other in succession without servicing the first order.

So, in this as we said first the order is given and the dish comes to the customer. So,

here again I split my transactions of my memory transactions into two parts. One is a request, you send a request and you wait for a response. So, we are going to split the five items which we did into two things mainly, a request and a response. Now what happens because of this? What are the advantages? You are going to get better bandwidth, that is more customers will get serviced in the restaurant whereas, in our example, many requests of different processors will get eventually serviced. They do not have to wait for each other.

Of course, you have to give the labels as we did in that example that you have to assign a ID with every request. So, that when the response comes it goes to the correct requester and again another third requirement is you need buffering. Because when multiple requests are pending they all need to be stored somewhere until they are answered. Okay. So, you need lot of buffering. The last question is what about serialization? Okay. So, what do you say about which write or which read happened before which one? We have been saying that the bus serializes the order and if I want to still retain that same argument in this split transaction bus whenever a request is issued that is when I serialize the request. Okay. So, I want to wait for the response, but the request will serialize the bus transactions.

With this design decision I have to cater to some issues in the future, but my design decision is serialization happens at the time of request. Okay. So, we saw normal bus, split transaction bus divided into request and response. It improves the bandwidth, we have to match the requests, we need extra buffering and we decide that serialization will be done when we win the bus during the request phase. Okay. So, in the request transaction whenever we win the bus arbitration that is the serial order. So, because of the split transaction bus, I need to split the bus also physically. Okay.

Initially I was saying that we have a common bus on which everything happens, but now that we have request and response as two different phases, I would need two different buses because we have to arbitrate each phase separately. I need to ask for the bus for sending the request, I need to ask for the bus for sending the response. Okay. So, I have a separate address bus and a separate data bus. Okay. Both the buses are now treated differently. Definitely initially also we had all these wires as separate wires, but we bundle them together calling it one bus, but now we divide it into two buses address and data. Fine.

So, what we have decided this, now we go back to our original set of transactions which we were doing and then we will understand how to split them and what are the request and responses related to each of them. The first transaction is the bus read. So, bus read was sent by a processor which wanted to read a block. So, when that transaction goes on

to the bus what is the response? So, if I say bus read is my request the response is the data. So, you get data in return the data could be sent by a cache or by memory.

So, we do not know whoever they will send it. So, that is about bus read. Then bus upgrade, bus upgrade takes place when we have the block we have one sorry the processor has the block it wants to write to the block and it sends a bus upgrade on to the bus telling others that they can now invalidate their copies. Okay. So, in this case what is the request bus upgrade, what is the response? Actually there is no response because the assumption is in an atomic bus that when I send a bus upgrade all the processors will invalidate their blocks. So, in this case I have to have some kind of a response because I have a split transaction bus which expects a response. Okay.

So, what we can do is whenever a bus upgrade is sent by a controller, that bus controller itself sends back an acknowledgement to the same processor. This way the response is going and when the bus is granted for this bus upgrade, it also serializes it in the bus order. This bus upgrade transaction request will be seen by everybody and then they will commit to invalidate their blocks. Okay. So, the commitment will come from their side, they will not invalidate immediately, but they will eventually commit it before the next transaction on the same block. Okay. Third type of request is BusRdX.

BusRdX expects a data response, but apart from data response it also expects other processors to invalidate the block. So, you need two things, data and acknowledgement of invalidation. So, both of these two things will be part of the response. The fourth type of transaction is write back and write back is only a request because we are sending the block as part of the request and there is no response. So, we have these four options 1, 2, 3 and 4.

Each of these transactions is now divided into or identified into the request and the response phases. Now when I divide this, it leads to some complications which we need to handle. First is conflicting request. When I have multiple transactions on a block, is it allowed that I can have read as well as writes for the same block happening because not everything is finished there are some pending transactions and can we allow this to happen. So, that is called conflicting request. Then we need to handle flow control and then snoop results. Okay.

So, these three things I will elaborate them one by one. So, the first one is about conflicting transactions or conflicting request. So, how do I define a conflicting operation? It is the operation on a same block, out of which minimum one is a write. So, I can have two write operations or one read and another write operation. So, whenever I have such operations where one of them is a write and others are reads or write, I will

say that these operations conflict on the same block.

And why is this a problem? Because if P1 and P2 both want to write to block A, what would they do? Their P1 will send a request, whatever the write request for block A onto the bus, as soon as it gets the bus grant for example, in a bus upgrade type of request. If you have a bus upgrade you will quickly get an acknowledgement and you will assume that yes, I have the block others will invalidate it and P1 can start writing. The same thing P2 will also do the same thing in parallel. Maybe this is the first request and that is the second event and they will happen one after the other, oblivious of each other and both of them will end up modifying the block. So, this is possible to happen hence we cannot permit it. Okay. So, we cannot permit conflicting operations to do because it will result into wrong coherence results.

The second thing to take care is flow control. Now what is this flow control? When I have split transaction bus, we are going to buffer the request. Here some request will be buffer with every processor and I need to limit on how many requests I am going to buffer then the responses will come. So, in any scenario whenever you have storage elements buffers or first in first out FIFOs, you need to worry about what happens if they become full. So, if the FIFO becomes full and the system is still sending you more and more requests what are you going to do? So, this concept is called flow control. How am I going to manage the flow of data through these limited buffers? The third issue is when I have these buffers and request and response happening far apart in time, what about the snoop results? Because snoop results, say that whether I have the block in shared state or dirty state, right.

So, if you recollect we had the shared wired or free signals. Now when do I send these three signals as part of the request or as part of the response? So, I have to also consider that. So, overall this is the picture atomic versus split. The top one is showing an atomic bus where the green transaction begins and it finishes before the orange transaction can start whereas, in a split transaction bus if you keep track that you can have multiple of those happening in parallel. Okay

So, that is split versus atomic. Okay. Now what is the design space for this or what are the options I need to take into account? So, one is about snoop results as we were discussing. When should I send the results? During the request or the response, that has to be decided. Second is how many requests can I keep outstanding? For example, in the example of my restaurant, how many customers request should the service person take in one go? 3 or 10 or 30? right. So, it depends on your capacity of processing. So, I need to see how what is the capacity of processing and then decide the number of outstanding requests I can handle.

The more requests you can handle the more bandwidth utilization happens, but the more buffering is required and so it implies flow control issues will come up. Okay. Then the third is what is the order of the responses? So, are you going to reply in the same order of the request or the responses can come in any order depending on when they are ready. So, either it is in order response or out of order response. Okay. So, we have two examples in real implementations. The Intel Pentium and the DEC TurboLaser, these two processors do in order response and the SGI that is Silicon Graphics Processor and the Sun Enterprise processor, these do out of order response. Okay.

So, we are going to see a detailed split bus based example using the Silicon Graphics Challenge processor. So, before we look at the split bus, let us have a feel of how big these systems are and what are their parameters. These are little older systems, but you can still get a feel of their scale and definitely the recent systems are bigger than these. So, the SGI challenge which is the Silicon Graphics Challenge Processor, it looks like this. It consists of a board and each board has got four processors.

You can see here, I have shown three boards. They are all connected to the power path bus, then connected to the memory and then the I/O subsystem. Okay. Some parameters are shown here. So, we will see them in detail. So, here you see that every board has got four processors and overall this system has got 36 R4400 MIPS processors.

What is their performance? It is 2.7 gigaflops is the performance. It uses a 16 GB interleaved memory. There are four I/O buses. Of course, these parameters are not directly related to the topic, we are discussing, but you will get a feel of how big and scalable such systems are.

It is connected to a 1.2 GBPS power path bus. Cache lines are 128 byte. It can handle 8 outstanding requests at a time and all the transactions take 5 cycles. The operating system is a variant of Unix called the IREX. Okay. So, this is the system and if you see power path, this is the power path bus on which we are going to discuss the split transactions. The next example, we are not going to use this directly, but it lies in the same lines of out of order response.

So, this is the Sun Enterprise system which is again a big system with 30 ultra spark processors and there are these cards. And every card has got two processors 1 and 2 and there are several such cards to finish the 30 processor system. Then you are connected to a Giga plane bus. In this case, instead of the power instead of the power path bus in SGI, you have a Giga plane bus in the Sun.

The memory here is distributed. If you can see it is not seen here. It is not connected on the bus, but it is on each of these cards. So, we can say that the memory is distributed, but it is still treated as a central memory. In the SGI, the memory is treated central because it is outside in one location not with the cards, but in Sun Enterprise, the memory is with the cards and the operating system is Solaris Unix. Okay. So, that just gives you a feel of how these processors are.

Now, we will go back to the technicalities of split transaction bus. So, the SGI power path-2 bus is a split transaction bus. It takes 8 outstanding requests at a time. It handles flow control by giving a NACK. NACK is negative acknowledgement, that is if the buffer is full, it refuses to take further request.

We are going to do these in detail in future, but right now a quick list that 8 transactions flows flow control using NACK and responses are in a different order than the request. The total order is maintained by the bus request phase. So, as soon as you request, you will have serialized the transaction. So, in this what are we going to look at? We are going to look at the bus design, then the snoop results and how do I handle conflicting request, then the flow control and lastly example of path of request through the whole system. Okay. So, we are going to see all these three topics in the next lecture. Thank you so much.