Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 04
Lecture - 24

Lec 24: VI Protocol

Hello everyone. We are doing module 3 on Cache Coherence. Today, we will start with Cache Coherence Protocols. The first protocol is the VI protocol. We are going to actually see four types of Cache Coherence Protocols, the two state VI, the three state MSI, the four state MESI and the four state Dragon protocol. Today's lecture is going to cover the two state VI protocol. Okay.

So, for implementing the protocol, we need to understand the type of messages or interactions which happen between the cache, the bus, the processor, everything. Okay. So the types of messages are listed here. Whenever the processor sends a read or a write, this is going to go as a request to the cache controller. So, processor reads will go as a PrRd and the processor writes will go as a PrWr message.

We have two types of messages coming from the processor. When these messages read to, reach to the cache, the cache is going to either incur a hit or a miss. If the cache incurs a cache miss, then it has to go out on the bus to the memory to bring the block. Okay. So, there has to be a message sent onto the bus. So, in case of a cache miss, if there is a read request from the processor, the cache controller sends a transaction onto the bus and this transaction is called the BusRd transaction because I am going onto the bus for reading the data item.

Similarly, if the processor gives a PrWr type of a message and there is a cache miss for this, we have to again send a transaction on the bus and that transaction will be called the BusWr transaction. Okay. So, the messages are PrRd, PrWr, and BusRd and BusWr. So, these are basic messages which we should know. Okay. Now we will discuss the VI protocol. It is a two state protocol, V is for valid and I is for invalid.

It is an invalidation based protocol and not an update based protocol. The caches are write through and they are write no-allocate. If you recollect, we normally pair this, the write no-allocate gets paired with the write through. Write no-allocate is happening whenever we just do a write without the block being present. So, a write happens to the block, but the block is not there in the cache and the cache is write through.

So, why to bring the block to the cache? Simply send the data item to the next level. So, when I have a write no-allocate, we pair it with a write through cache and when I have a write allocate cache, it gets paired with the write update cache or no write back cache, sorry. Okay. So, this was the pairing which we had discussed during the review of the caches. So, this VI protocol is invalidation based write through using write no-allocate. So, we need to remember these terms. Okay.

So, I have listed them here as well. Invalidation based protocol that is when one processor writes to the data, others have to remove their copies. So, I have just truncated the FSM to two states. The blue block is present in P1. So, probably it is invalid state here, it is valid here, valid here and the blue color is absent in P3.

So, for this particular block, I will say that it is in state I in the processor P3. Okay. So, we have these FSMs interacting with each other. When P1 starts writing into this, so now P1 has the block, suppose it wants to write to this block, what should it do? When it wants to write, it has to invalidate other copies. So, it has to remove this copy. It need not remove the copy of P3 because it does not have the copy.

So, that copy will get removed and it is a write through protocol. So, any changes into this will also be sent to the memory. So, memory will be up to date with the value. Suppose I am making x equal to 2, that value will be propagated to memory because it is a write through cache. I will invalidate the block of P2 because it is a invalidation based protocol. Okay.

Now, suppose in P3, suppose in P3, P3 says I want to change x to 5. So, what should happen? P3 does not have the block, it is an invalid state, but it is a write no-allocate type of a design. So, P3 is not supposed to bring the block, only thing it will do is propagate this x equal to 5 to the memory and in this process, it also invalidates the other copies. Okay. So, when P3 changes the value of x, it does not have x in its cache, but it is not supposed to fetch x because it is a write no-allocate design. Hence, it simply sends the value of x equal to 5 on the interconnect, sends it to the memory and in the same process also invalidates the other two copies. Okay.

So, continuing the example, suppose P1 does a write, then it has to delete the block of P2 and P3 does not have the block. Subsequently, if P2 does a write, it is going to delete P1's block and if P3 does a write, it is directly going to write to the memory and it is going to delete the other sharers. Okay, now with this understanding, let us start drawing the finite state machine for the VI protocol. I hope you have understood how it functions and what are the messages I have? The processor sends a read, the processor sends a

write and on the bus, I can send a bus read or a bus write. Okay, so with these, with this information, we are going to draw the state machine for the VI protocol.

Okay, right. So I have two states. The first state is I for invalid and the second state is V for valid. The block can be in one of these two states only. And if you can see, there is no state called dirty because it is a write through cache and every update is anyway going on to the bus to the main memory. If the block is invalid and the processor wants to write to this block or let us do the read thing first. Okay.

We are in the invalid state and the processor sends a read request. The block is invalid, processor sends a read, what should we do? We have to send a request on to the bus telling that I want this data. So when this BusRd request goes, the main memory will provide data to this block. Okay, and once we get this block, what should be my next state? It will become valid. When you have a PrRd, you send a BusRd request on to the bus, you will eventually get the data and you shift to the V state.

In the I state, suppose there was a processor write request, processor write request comes, block is invalid. Are we going to bring the block? No, because this is a write through protocol with write no-allocate. So this processor write is simply going to do a BusWr. What is happening inside this BusWr? Actually the value gets transmitted. I am going to send this value to the memory in the bus write transaction.

Will this change the state of the cache block? No, it won't because we haven't brought the block, so we will stay in the invalid state. So in PrWr, you send the new value on to the bus to the main memory and don't change the cache state. Coming back to the valid state, we have come to the valid state because of the read signal. So there was a processor read and we came into the V state. Subsequent processor reads, what will happen? They will stay in the same state and we don't need any transaction going on to the bus because this is a local hit and we can cater to this data.

What happens if I have a PrWr? If a processor write happens and we are in the valid state, we will definitely remain in the valid state, but we also need to send the new value on to the bus using a BusWr transaction because this is a write through cache. Right? So bus write, the value is going to be sent to the memory. Okay. So this is the local behavior. Now suppose a request comes on to the bus, there is another cache which is also reading and writing to this block. So when the other cache wants to read, are we going to change anything? No we are not, because multiple readers are allowed.

But if the other cache is going to modify the data, that is another cache receives a processor write request. Okay. So let me say there is a cache, I can use another color for

it. So there is another cache which sends a write. I get a processor write from another processor.

So this one comes here. So what should this particular processor do? It will send a bus write signal. The BusWr signal will go and this bus write signal will reach my FSM. So I am in this red FSM and the green FSM is sending me a BusWr signal. So what should we do? This is telling me that there is another writer into the system. Our protocol is invalidation based protocol and hence we should remove this block from our cache.

So if we are in the valid state and if we see a bus write happening, this is the input to us, then you simply go to the invalid state, that is remove the block from your cache. Okay. So a neater diagram is shown in this slide. PrWr, send a BusWr, PrRd, send a BusRd, become valid. In the valid state all the read writes are locally catered to but every write will go on to the bus. Whenever you receive a BusWr onto the bus, that is a BusWr signal is going, traveling onto the bus, you have to invalidate the block.

I am using different colors. The blue one is for the request coming from the processor and the red one is used for the request coming onto the bus. That is we have a processor, the cache and the bus. This was the design. So requests going onto the bus are shown in red color and requests coming from the processor are showing in the blue color. So this is shown by blue text and this is shown by the red text.

So the cache snooper has to see both sides for the implementation. Okay. We elaborate this with a small example or a state transition diagram. Right. So this table is showing the cache state invalid or valid and the blue columns are belonging to this particular processor, the processor I am interested in and the yellow one is for other processors. So what happens with respect to them? Okay, so let us fill the blue part. The blue one says there are two columns load and store.

So this particular processor sends a load request that is a read request on the block. So what are the options? Either I have the block or I do not have the block. So for a load request, if the block is invalid, what should we do? You have to read the block, bring the block from the main memory and change state to valid. So I will start filling the table and you can also pause the video and try it out for yourself. Okay, so here I am going to incur a cache miss.

There was a load request, the block was invalid and hence it is a cache miss. What happens on this cache miss? Go bring the block from the memory and then change your state to V. So we will become valid after this. If a store comes, the block is invalid. What do we do? We do not bring the block because it is a write no-allocate protocol.

In write no-allocate, the stores will directly go and modify the memory. So we will simply do a write through. We will do a write through of the data and what is our state after this? It remains I. We are still in the invalid state.

We did not bring the block to the cache. Right. So that happens. Now we will cater to the second condition. This particular processor sends me a load request and we have the block. What do we do? We simply send the data to the processor and no changes happen into the system. So this is a cache hit and your state remains V.

No changes in the state happen. If there is a store request, the block is present. So this is again a cache hit. So write will happen here but at the same time you have to send the updated value over the bus. So write through also has to take place.

The state will still remain V. So this is the behavior with respect to this particular processor. Now what happens with the other processors? Another processor issues a load request and it misses in its cache. So there is another processor which wanted to read the data item. It did not find the data item. Hence, it sent a bus read transaction because it wants this data.

But it is wanting it for reading and it does not affect the blue processor. Right. So let the yellow processor read the data item and the blue can still keep its own copy. And who provides the data to this yellow processor? Does the blue processor give? No, because the memory is anyway up to date. So the current processor has to do no action. So this no action is with respect to the blue processor.

If the other processor sends a BusRd request, the current processor has to do no action. Similarly, if the other processor sends a BusWr signal, the current processor has to do no action. It has to take no action in these two cases. Why? Because the block is invalid. Right?

The block is not there. So any reads or writes happening on the bus, I can ignore them. But the problem comes here when the block is valid. If the block is valid, another processor wants to read. This is the cell I am talking about. We have the block invalid state and the processor wants to read, a bus read comes.

What do I do? Do I provide the data? I need not. Why? Because memory will give the data. So here I will say memory gives data to this yellow processor. The blue processor can keep its data, keep its block as it is and the memory gives the data to the yellow processor. Does it change the state of the blue processor? No, it still remains in the valid

state. When the other processor sends a BusWr, so it is wanting to change the value.

So the change will go onto the bus as an updated data item. So here is that data goes to memory because of the bus write from the yellow processor. So that data is going on going to the memory and the blue processor has to remove its block and become invalid because it is a invalidation based protocol. Any new writer into the system is going to remove all the other copies of the block. So whatever we have written is with respect to the blue processors.

So in this, whatever yellow processor asks, I am not going to take action. Here the memory is going to provide data to the yellow processor. The blue processor need not do anything. So that is the meaning of how we have filled this table. Okay.

So a neater version is given here. That is the state transition table for the VI protocol. Okay. So next, once we have designed a protocol, our next task is to prove that it satisfies coherence. Okay. So the problem is complicated, but what we are going to do, we are already saying that I will handle consistency later and presently I am going to use a very simplistic system. That is I am going to have several assumptions defined. So what are the assumptions I am defining? This is with respect to the bus.

So I am saying that my bus is atomic. That is it does not delay anything whatever work is given, it completes that work in almost no time. So bus is atomic. At a time only one transaction can happen. So bus can handle one transaction until that transaction finishes, it does not start the next transaction.

All the actions related to this transaction will finish. Suppose it is an invalidation message, so a bus write is going, all the caches have to remove their copies. So I am assuming that when a bus write travels onto the bus, all the other caches will invalidate their copies only then this particular transaction completes. So all small small actions will complete before the next transaction can begin. And because I can handle only one transaction at a time, all the writes which are happening, they will happen in serial order over the bus. Okay. So one transaction everything completes before the next transaction, invalidations also applied and the bus order make sure that the reads and writes will be handled sequentially.

So these are the assumptions I am going to do before I start proving coherence. So to prove coherence what we need to do, we have to prove either the write propagation that everybody has seen the value and then write serialization that is all the writes are serialized. For doing this in a VI protocol what I am going to attempt is can I construct a total order of all these operations which is coherent with the complete system. So can I

serialize all these operations happening to get a final hypothetical serial order and then check whether everything is falling in place. Okay, so if I have a write through cache, all writes are going on to the bus.

So all the writes which are going on to the bus will be seen by every cache. So write propagation is taken care of. The bus is going to handle one transaction at a time. So if you are doing multiple writes or if multiple processors are doing write to the same one they will be going in a particular order. So all the writes which are going on to the bus they will follow the order in which they acquire the bus.

So bus decides the order of writes. So even if two processors want to write at the same time the order in which they get the bus will be the serial order of writing to them. And similarly for invalidations that the bus order will decide which processors get invalidated. Okay, so I am going to try to construct a total order. The bus order is going to decide the order in which the writes are happening and the invalidations get also performed in the bus order.

Okay, so we will see some example. So bus orders the writes, that is the writes get ordered because of the bus ordering is maintained among the writes. So I have shown four processors, the memory and the bus. Now the color coding is that this is with respect to one block, but the colors will say requests coming from those particular processors. So the green and the blue are the same block, but green color denotes the request coming from P2, blue says request coming from P1. Now how is the bus going to order the writes? So let us see what is happening in the system.

So I get R, so time is going in this side, so this is the order, so this is the first transition which you can see, then the next action and the next action. So a blue color R is seen which means processor P1 sends a read and it did not have the block that is why the read went on to the bus, the memory will satisfy this request, so memory will provide data to this one and eventually P1 will finish its read operation. Subsequently P3 sends its read request, again the memory is going to give the data because P3 did not have the block and that is the reason why it came on to the bus. So once the memory gives the data, then the P2 reads and now P3 does a write. Now when P3 does a write, invalidation based protocol, so it has to invalidate all the other copies.

So who gets invalidated? The green and the blue because the pink remains as it is. So this invalidation is actually going to remove the green color and the blue color. Right. So these two processors will remove their data item because this write has happened. Next the blue and the pink again try to write, again try to read. So when blue tries to read, does it have the block? No because we recently invalidated its block, so it goes on to the

bus and who will give the data? Let memory give the data because it is a write through cache. Right.

So the pink processor need not give the write, the memory can give the data to the blue processor. The read by pink processor also succeeds after this and now the P4 does a write. When P4 does a write, it also has to invalidate others. So in this case, who all will get invalidated? In this, the green already is invalidated.

So here the blue and the pink get invalidated when the yellow write happens. Now again the green read happens. Now green read happens, memory gives the data and what data will it get? It is actually going to get this particular value. Okay. So what you can see is the order in which the writes are going. So this is one write and this is another write.

So the pink write and the yellow write. So the order in which they went decided the order in which the writes got serialized and the order in which they got read. So here this pink is going to decide the values these two are going to get and the yellow write is going to decide what value the green is going to get because the read is following the write. They will happen in this particular order. Okay. So the bus is going to order the writes.

So moving on and understanding the write serialization. So writes are seen by the processor by issuing reads. So in this slide we saw that yes the bus is ordering everything but how is the write serialized and how do I know that the writes were serialized? That is by issuing reads. So when I issue a read, the value which I get will decide which particular written value am I going to obtain. So the reads are going to tell what the writes were. But not all reads will go onto the bus because if I have a local read hit, I am not going to go onto the bus.

So how do I say that it gets serialized? But only certain reads which incur due to the cache miss will go onto the bus. So we have to handle these two cases that is reads which go onto the bus because of a read miss. Right. So here we have again the four processors and this is the partial work already done. So this much work is already done. Two, Three reads have happened, one write from the pink processor, another read from the blue processor.

Now after this, suppose the pink processor reads and after this the blue one also reads. So what can I say? These top three, they do not go onto the bus because they are read hits. They are read hits, they will not go onto the bus. Later the yellow write happens and because of this yellow write, any subsequent reads from another processors will go

onto the bus but this is a hit. The yellow gets a hit probably if it has brought the block but it, the pink one may or may not have this. Right.

So only the miss goes onto the bus and the hits do not go onto the bus. So how do I order these reads? How do I say that R1, R2, R3, suppose I put these numbers, what is the order in which these reads happen? Is a particular order there or any order is okay for me because as long as we sit between these two boundaries, all the reads which are happening here between the two writes, they can happen in any order whether they are hits or misses because if it is a miss, it is going to get the most latest copy. If it is a miss, most latest copy, if it is a hit, then you have already cached that data item. Right. You have already brought this latest copy and going to do a local read to that data item. So the read hits have already the most recent data item and hence they can be ordered in any way with respect to the bus order. So between these two boundaries which we have constructed, all the reads can happen in any order. Okay.

So that is about reads that do not appear on the bus. So these are read hits and read hits, how did you get the value? You got the value from your previous write or your previous read which went on to the bus. So you have already brought the block, you are doing a local read, you have already updated the block locally, so you are going to get the most recent value. Okay. So overall the bus order is ordering the reads writes properly and every program is following its own program order. So if we have these two things, they give us enough constraints to say that the VI protocol is following and satisfying write serialization. So having proved the write serialization with respect to the VI protocol, we are just going to see a generic method of how do we construct a total order.

Because no matter what the protocol is, we should be able to see what is the final sequence of reads and writes which have reached memory. No matter it is VI, MSI or MESI protocol. So the method of constructing the total order is by constructing small partial orders and then stringing them together. So how do I construct a partial order with respect to reads and writes? So we have four situations in this case.

So we are going to see them one by one. The first situation is when I have two transactions, two memory operations M1 and M2 happening in the same processor. Okay. So we are discussing with respect to the same processor. The second condition is or second scenario is when I have a read operation which happens after a write and they generate a bus transaction. So read followed by a write and then I have a write which is subsequent to a read or a write which again go onto the bus. The fourth one is we have a write operation which is subsequent to a read but in this case we are talking of local hits. Okay.

Four scenarios we will do them one by one. So the first scenario, a memory operation M2 is subsequent to a memory operation M1 if the operations are issued in the same processor and M2 follows M1 in the program order. Simple, we have M2, this will come after M1. So what is this arrow showing? This arrow says that M2 comes after M1. M1 goes out to the memory first and then M2 will go out onto the bus.

So that is the meaning of this arrow. Okay. So as long as the program order is maintained, in the program order if I have M1 then M2 then in this total order which I am constructing, I will put them in this order. So here I will first put M1 and in the total order then I will put M2. So if I am constructing the sequence. Second condition, a read operation is subsequent to a write operation W.

So if this read generates a bus transaction that follows that of W. Okay. So I am saying that this read transaction is coming after the write transaction only if both of them are separated by a bus transaction. Okay. If they were not separated by bus transaction then they would be probably within the given process but if they are across two processes then definitely a bus transaction should take place. Right. So this write, the blue write generated a bus transaction and the read also generated a bus transaction. Why did it generate a bus transaction? Because this read was not from the processor which did this write.

Okay. So if I am again stringing them together I will say that this write happens and after that the read happens. So this is how I am going to put them in the total order. Third condition, a write operation is subsequent to a read or write operation M, if the M is generating a bus transaction and the bus transaction for this write is following the M's bus transaction.

Okay. So I have the M which could be a read or a write. So I am not restricting myself to only reads or writes. It could be either transaction. When either of this transaction is generating a bus transaction. This M produces the bus transaction.

After that I have a write. So I am going to say that this write also goes on to the bus. So we have either a read in the blue color or I have a write again by the blue processor. So both of them could go in any order.

So this is just one box. It is either this or this. We have a read or a write and they go on to the bus. So I am just drawing the total order and after this I am going to put the green write. Okay. So this is the order in which I am going to string them together. Fourth condition, a write operation is subsequent to a read operation M if the read does not generate a bus transaction, that is it is a read hit.

It is a read hit and it is not already separated from the write by another bus transaction. Okay. So now let us see this pictorially. I have a transaction M which is a read hit. Read hit that is it is a local hit you already have to block and then I have a write which generates a bus transaction. So we were doing reads in the local processor and then a write came on to the bus. So when I generate the total order I will first put this particular read which I was doing and only then I will put the write because this write came after I had finished the read.

Okay. In case the bus transaction had not happened. Suppose this green processor does not generate the bus transaction it would rather mean that this is a local write hit within the blue processor. So I would not have the write in the green color it will be the blue color because only when a write is from a different processor it is going to generate the bus transaction because otherwise you have it in the block and you have the local read hits happening.

So probably you can also do a write in the program order of the blue processor. Okay. Okay. So these four conditions have given us how do we sequence or order things. So when I put this first, second, third, fourth, right when I construct the total order which is going to the memory, how do I put them in sequence, right? What goes in which order? So these four conditions have given us how to put things in order. So to construct the total order I need to first get all these small small partial orders and then definitely all these small partial orders are coherent and we can construct a serial order because every partial order will follow some transitive relationship, right? So there is a transitive relationship among them and we can utilize this to join several partial orders together. Okay. Then several read transactions that happen within the processors they happen in the program order they do not go onto the bus, so we can order them in any way, right? So there are several reads R1, R2, R3.

So all of them can be stringed together in any order into my total order and all these interleaving read operations will be in some segment so they can be put in the total order but definitely you have to follow the program order. Okay. So if you see here onto the bus I have some reads and writes and on the top line you can see some read hits, you can see some read hits here. If I ask you all these read hits the pink, the blues where will you put them in the total order or where will you put them in the bus order? Okay.

So this read it could have happened here, it could happen here. Similarly this blue but these blues have to follow this order because they need to follow the program order of the blue processor. Then the green so this follows like this. Subsequently these two can be put in any order, right? So both the reads, the blue read and the yellow read can again

be stringed together in any order. Okay. So what am I trying to construct? I am trying to bead all of this together to construct a total order from my partial orders.

Okay. So if I start reading this, right? So I will put this first, second, third, fourth. After this definitely the blue will come but I can take this or I could have taken the pink one also. But once I take the blue path I will have to finish this and in parallel the pink will also go into this. So if I would have stringed it like this, suppose let me do it again.

So I will take the blue one and then let me take the pink. Definitely the blue should go in the program order. So this is the program order which I have to follow. The pink could have happened initially here or I could have done pink in the third position after the two blues. Then again I have to come down, take up this green, take this one and here I have two options either I take that or this because both of them will be.

So this is how I will string these several partial orders together to construct the final total order. Okay. The same thing, several read hits, they can happen in any order but they all merge when there is a write transaction onto the bus and again several parallel reads can happen and they again merge when a write transaction happens onto the bus. Okay. So this way we can use several partial orders and string them together to construct a total order following those four conditions which we have discussed. Thank you so much.