Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 06
Lecture - 31

Lec 31: Single-Level caches with an Atomic Bus (1)

So, hello everyone, we are doing a module 4. On snoop based multiprocessor design. This is lecture number 2, where we are going to look at single level caches with an atomic bus. So, this is part 1 of the lecture or part 1 of the topic and then we will do lecture 3 as part 2 of the same topic. Okay. So, in this whole module what are we trying to do? We are going to develop snoop based multiprocessors and the various levels of abstractions as we open up the layers, we will go from single level cache to multi level cache, atomic bus to a non atomic bus. Right. So, we are going to see all those designs one by one. So, in this one we are going to check at how do we design a single level cache which uses or runs on an atomic bus.

So, while doing that actually how do we design the system? We have a processor, a cache that is the basic thing we study in a computer architecture or organization course. Now, when I want to integrate this for a multi core which is cache coherent system, what changes do I need to do for the caches, so that it becomes friendly to run parallel programs across different cores. Okay. So, for doing this we need to understand any changes to be done to the cache. Then we have been talking of various snooping protocols which communicate or inform each other about snooping results and they invalidate their blocks.

So, while doing this or actually how is this implemented, we are going to look at that aspect. Then what happens on writebacks, do we need to really care about what happens there and once we come up with a design that is any small changes to a single core system which we will adapt for the snooping based multiprocessor set up. So, once we have done that, then we will look at the correctness requirements which we looked in the lecture number 1. That is mainly cache coherence correctness, that is write serialization and sequential consistency correctness which is write completeness, write atomicity and then other progressive correctness requirements like deadlock, livelock and starvation freedom. Right. So, this is what we are going to see in this topic single level cache on an atomic bus. Okay.

So, the list of changes which I will want to do in my design mainly comes on what

changes am I going to do for the cache controller when I move from a uniprocessor cache to a multiprocessor cache. Then we always say snoop reserve. So, how am I actually going to implement this, are there any race conditions I need to handle and yes definitely all the memory accesses are in real implementation not atomic. So, when they are not atomic what problems do they create, are there further race conditions which I need to cater to. So, that we will look at and then finally, we will look at all the correctness aspects of serialization, deadlock, livelock and starvation freedom. Okay.

So, we will start with the cache controller. Right. So, we said that we are going to look at the cache controller, further we will look at snoop results, then any race conditions, lastly the correctness. Okay. So, cache controller and subsequently the tag design. So, whenever you are trying to understand this topic, keep relating to what is the basic design in a uniprocessor setup and what changes I am doing for making it adapt to a cache coherent multiprocessor setup. Okay. So, for that let us see what happens in a uniprocessor cache, so that we can incrementally change it. Okay.

So, uniprocessor cache if you recollect the lectures we have done, it consists of data, tag, states, right? it has a comparator which is used for address comparison and then definitely the controller which is the main part which does all these tasks and all this is finally connected to the bus. Right. So, the cache communicates with the bus and of course with the processor and the next level caches or the memory. So, that is the uniprocessor cache. Now this cache it interacts with the processor. Cache interacts with processor any request from the processor will be moved on to the next level of memory. So, what interaction happens at the processor level? processor sends a request, we will check whether the tags match, whether it is a hit or a miss, depending on the hit or miss we will either bring data or not and accordingly update the state bits.

If we need the block then we have to go to the memory using the bus interface and then fetch the data. If we are writing back a block again we have to go on to the bus interface to the memory to write back the data. So, this is the interaction of the cache with the processor. So, if I say this is the microprocessor, this is the interaction I am talking about. Now the cache is connected to the bus, so it also interacts with the bus controller.

Now when it has to initiate a bus operation that is to write back a data or sending a request to read, again it is not a single thing it does. There are several small things which happen or it is rather a not a complete atomic action. Okay. So, the cache controller when it talks to the bus it first has to assert a request. So, we have to send a request to the bus, then you will be granted the bus, after we get the grant then we have to send what action I want to do. Okay. So, first send the request, wait for the bus grant, once you get the grant, give the address you are considering to read or write and then the command what

are you supposed to do, are you going to read the data or you want to write to the data, okay.

So, the address and command go onto the bus, once this goes there is some receiving device which will understand that command, that is if it is memory, memory will understand that this is the address it is supposed to cater to and do read or write. So, there will be somebody who accepts your request and this device will then be involved in your data transfer, either receiving the data or sending the data. So, just one bus operation is now divided into five things and you will now realize that all these five things must happen one after the other and in an atomic bus I assume that all of these five will happen together without any other disturbance in the middle, right. So, in this topic I am assuming an atomic bus. So, all of them are going to finish in one go before a next bus transaction can take place, okay.

So, this way we are going to extend the uniprocessor cache. So, uniprocessor cache is nothing but a finite state machine. Okay. Every hardware the system has to be implemented using a state machine. Then the transactions which happen on the bus you will also realize that there are five things and lot of interactions so that is also an FSM. So, the sequence of steps related to the bus, sequence of steps related with the processor, both of these are finite state machines. And now, do not confuse this with the coherence protocol FSM, right.

So, coherence protocol was a third FSM which was a block level FSM for every cache block . Okay. So, overall there are now three FSMs which we are considering here. Then the uniprocessor cache has to monitor the bus as well as respond to the processor request. So, till now we were only considering that the uniprocessor cache accepts request from the processor and the response to the processor and then uses the bus to do some data transfer, right, bringing data or sending data. But when I put this uniprocessor cache in a multi core cache coherent system, there are requests coming on the bus which I have to cater to, right, because some other processor wants a block and this current processor has it.

So, if a BusRdX transaction goes on to the bus, this uniprocessor cache which was initially not worried about looking what happens on to the bus. Now it is supposed to look on to the bus, snoop on to the bus and see whether the request is applicable to this processor. So, the controller also has to look at the bus happenings right now, okay. So, we have to look at the processor side request and also monitor the bus while implementing the protocol. Okay. So, to implement snooping protocol, we essentially will have FSM, which looks caters to the processor request and an FSM which caters to the bus request.

Now when I have these two FSMs which I am discussing, they will be wanting to compare the addresses. So, processor sends the address request as well as something goes on to the bus and that also reaches to the controller. Now the controller needs to compare this address. Now processor wants to compare as well as the snooper or the snooping controller also wants to compare. So, the tags is one storage which is in high demand.

So, if both of them want to compare, they will do it sequentially. So, this is going to add to the latency. If the processor is stalled, it is definitely not desirable. Hence we conclude that when I move a uniprocessor cache to a cache core and multiprocessor system, I need to do something for the tag storage. And what we are going to do, we are going to duplicate the tag storage. Okay.

So processor is comparing, snoop is also comparing the tags and hence I am going to use two storages that is use a dual tag array. Definitely I am not going to duplicate the data. Okay. Data is a single copy, but tags will be duplicated or we can use a dual ported RAM for the tag. So you can have two ports which will give you parallel access to the tags. Okay. So while we are doing all of this, we will guarantee that the processor is not stalled for long time.

There will be only cases when there is some, these both tags, I have two tag arrays. So these two tag arrays T1 and T2, when they need to update each other, only those cases is where everything will be stalled, but otherwise routinely it will go faster. The common case is very fast. Okay. So this is how the design will look like. We have the cache data and then we duplicate the tags.

So this is a set of tags on the bus side and this is a set of tags for the processor side. So when processor says, it goes to the blue box to compare and when the bus snooper wants to see, it checks the yellow box. And at the bottom, I have just for representation given that there are some FSMs. Right. I have just left them empty to show that this is some abstract diagram. There is a bus side finite state machine, there is a processor side finite state machine and definitely a block level coherence finite state machine. Okay.

So all these machines have to interact with each other to make the whole system work. So now the cache controller is extended with two caches, sorry tag arrays and it also has to do something more. Now what is that extra feature? So initially we were only comparing addresses here. So when an address comes, we were comparing the tags there and now something comes onto the bus. So now when an address comes onto the bus, we are also supposed to compare those addresses with this yellow storage. Okay.

So we need extra comparators to do this and not only compare if there is a hit, that is if we have the block in our cache, so the tag say there is a cache hit, then we may have to put the data onto the bus if we have modified the data or we can simply change the state in case it is a read only block. Okay. So in an update based protocol, I also have to pick up the data from the bus. So the two added features to uniprocessor caches, the first thing is I should be able to put data onto the bus which the uniprocessor cache never did. What did uniprocessor cache? It only put the data on write backs, but here in a snooping friendly cache, I have to put the data on demand from the bus snoopers. When bus says give me data, we have to give data and the second thing it has to do is pick up the data also.

Uniprocessor cache only brought data from the memory whereas a multi core cache coherent processor cache has to pick up the data on demand from the bus in case an update for the block is going. Okay. So these are the two added features and that will extend my cache controller. Okay. So we have added two features to my cache controller. Now we will look at the next aspect of snooping results. Now what is this? We were saying that among the shared data when we transfer, sorry when we have to read or write, we will send a bus request, a BusRd or a BusRdX transaction and subsequently some other cache is going to say that whether they have the copy or not. Okay. Because we need to know whether there are other sharers.

Multiple occasions would need this information. The simplest example comes to my mind is when you want to read and nobody has the shared data, then we can load the data into the E that is exclusive state in the MESI protocol. Other uses also of this. Right. So when a transaction goes on to the bus, all the caches have to respond to this transaction. Either they say they have the block or they have to say they do not have the block.

So this is called snooping result. So we have to declare the snooping results whenever a bus transaction takes place. Now how are we going to declare this? So the first question is when should we tell this answer? and then how should we tell this answer? So when is time dependent and how is, what is the design change I am going to do. So we are going to see these two aspects. And, the first question is when do we declare snoop results and why is this important? This is important because if the memory which is connected to the interconnect, it has to decide whether it should give the block or some other cache is going to give the block. All right.

So if there is a sharer having modified the data, that cache is going to provide to the new reader. If there is no sharer with changed data, then the memory has to give. So in any case, the memory has to always be alert to decide whether it should give the data or

some cache is giving. So this decision has to be done quickly. Okay. So that is why we have to have a method for doing this.

So I am going to consider three options of this when answer. Okay. So I am going to say that let me keep a fixed delay before the memory decides that it has to give or not. Let the delay be optimistic that is variable and third option is immediately. Okay. So these are the three options we are going to consider. So using this, we are making sure that the memory will know whether it is giving the block or some cache is giving the block.

Let us take the first option of fixed delay. So now the snoop results will come in a fixed delay. When we say fixed delay, we have to say how many clock cycles. So you will have to find out how many clock cycles does it take for the snooping controller to go to the tag array, compare the address, declare a hit or miss and then send the answer onto the bus. Okay, so to reduce this delay, we are definitely using dual tag so that we do not contend with the processor, we have our separate tag storage to compare.

But we still need to be conservative because in case we went to the yellow side tag array, the processor might be changing the tag or state so that both the tag arrays are currently locked. So we have to be slightly conservative when the CPU might be changing some state of the tags. However, we can still derive a reasonable conservative assumption about the number of cycles required to do this. So once we have this, we know that after n number of cycles, my snoop results will be available onto the bus. Okay. So what is the advantage of this? The advantage is that the main memory is not affected at all. Because the main memory simply waits and I am saying suppose you wait for 8 cycles, just an example. Okay.

So memory waits for 8 clock cycles and until 8 clock cycles, if no snooping result come onto the bus, it decides that yes, it has to give the data. If some answer comes, then memory does not do anything. Okay. And the cache to cache transfers are also easy. So if there is a cache which has the block, it can simply put it on the bus and the reader can utilize this data. Okay. Disadvantage is that you need some extra hardware for doing this and potentially some longer latency because we are doing a conservative estimate of the delay.

This is a very useful method and definitely implemented in real systems like the Pentium Pro, the HP server and the Sun Enterprise. These three machines use this fixed delay based snoop result. Okay. Second option, variable delay. Now variable delay, how much different and how do you decide this? So here we, this is the best thing or the most optimal answer we can arrive at. So how do you decide this? Mainly the memory is

waiting, if you recollect, we, the memory is waiting to decide whether the cache is giving the block or memory is giving the block. Okay.

So memory says, okay, I will not wait for example 8 clock cycles but I will wait until you tell me you are not going to give. Because if I keep a fixed delay, maybe some caches are fast, maybe some caches are slow but you always have to wait for that much time. But in case the caches are fast enough and I get the snoop results in three cycles for example or three time units, then the memory can quickly decide in lesser time than the fixed delay. So here the memory says I will wait until all the caches tell me that whether they are giving the data or they are not giving the data. So this is of course less conservative, hence we do not have to wait for the worst case delay and hence most fastest. So it is flexible but once you bring in flexibility, the design becomes complex.

Now what is the complexity of the design? Here the cache has to talk with the memory which is an added feature. But as it is modifying the uniprocessor cache to be able to adapt to a multi-core system, now we are saying the cache also should talk to the memory which is connected far off onto the interconnect. So we need to establish a handshake between the cache and the memory. Okay. So this is extra work. The other optimization people try here is that the memory is very impatient, it says let me bring the data because I am a slow person, let me bring the data in the meanwhile the caches will decide what they want to do.

So the data is almost ready with the memory. If the caches say they are providing the data, memory says okay I will not give you this data, otherwise memory puts that data onto the bus. So we can have this optimization also implemented. So for a variable delay, the memory assumes that the cache will give until they say otherwise, until they say I am not going to give. It is less conservative, more flexible, complex implementation because now the cache and the memory will talk with each other.

So extra work to do. Optimization is we can start fetching the data from the memory bank and if the caches give the data we can simply discard the memory fetch or stall the memory fetch. This method is implemented in the SGI Challenge processor. Third is immediately. Now this is coming like a magic. How do I do? Because initially I was saying it will take some 8 to 10 clock cycles or lots of time.

Then in the variable we are saying that let the caches tell me whether I am giving it or not. So how do I come up with the third option of immediately? So immediately is the memory decides whether it is giving or not immediately. That is the memory knows whether the cache will give or not. Now how will the memory know this? Right. So memory is here. It has to know whether the block B which I am accessing or which is

being sent for whom the request is sent on to the bus, should I give the block or not? That is will the cache give this block or not? Okay. So for memory to be able to decide this in one clock cycle or immediately is that memory maintains a map as to which caches have this block.

So if the memory knows which cache has the block, memory will know that yes that cache can give. If no cache has it then memory has to give. Right. So immediately I can decide if I put one bit per cache block indicating whether a cache has it and whether it has been modified. So if a cache has it and in a dirty state then that particular cache is ready to give that data. So this option is good but it is expensive because this ends up adding this extra hardware to the memory and memories are normally considered to be more commodity devices and we do not want to change them based on the type of protocol and based on the type of variations which we are discussing. Right.

So we do not want to touch the memory. Let it be as generic as possible and hence mostly we do not prefer this option but this is also one good option to think of. Okay. So disadvantages, we need extra hardware and we have to modify the memory subsystem which is a big disadvantage. Okay. So these are the three ways in which I can report the snoop results. The next question is how am I going to give these results? We discussed when the results will come. Now how are they going to come? How are they going to come meaning I have a MESI protocol and in this MESI protocol when a request goes onto the bus, suppose it is a BusRdX, then we were saying that if somebody has the block put that block onto the bus, if nobody has then we can load it into the E state and so on. Right.

So the result has to come in some format. So whether the block is dirty, whether the block is shared all this information should come so that I can take appropriate actions whichever are required. So I need to know whether the block is shared so that we can load in the correct state. I need to know whether somebody has the block in dirty mode, why? because that cache is going to give me the data and not the memory. So memory should also know that it should not give the data.

So we have to give these two answers onto the bus. Okay. So for that if you recollect we had discussed the shared wired-OR signal earlier that every cache if it has the block in S state it will say yes. Suppose this says no, this one says yes. Okay. So everybody will put an answer and you can say that these all answers are collected to generate this final signal. So we have a wired-OR shared signal.

This will say whether the block is shared or not. We can have a similar design for the dirty signal. Again wired-OR. Any cache which has the block in dirty mode will raise it

to 1. So I have all these blocks, sorry all these caches sending the answer. Suppose he says I am the block and they do not have then this bit will become 1.

If this is 0 then the answer will be 0. Okay. So depending on the dirty status across the processors your dirty signal will be generated. Okay. So this way I have integrated the two. So we have the blue line telling about the dirty state, red telling about the shared state. Okay. And now the memory is waiting for this and accordingly memory should either give the block or not give the block. But when should the memory decide that the D and the S signals are ready? Are these signals ready to be sampled? Have all caches finished the snooping? We do not know. Okay.

So how long should we wait for this D and S to come? So for that we add a third signal which tells that whenever D and S are ready it will say now D and S are ready you can sample it. So the third signal is called the snoop valid signal, this pink line. Okay. So this pink line is normally set to 1 by default and when this is ready and this is ready when both of them are ready it will make this 1 to 0. Okay. So when the pink line becomes 0 memory will decide whether it should give the block or not. Okay.

So this is how the snoop signals are implemented. So you can see what all additions we are doing to uniprocessor cache. Earlier we simply had the processor, the cache and the bus. Okay. Now we have these three colorful lines getting added plus more decision logic also implemented. Okay. So a summary here, so we are going to use three wired-OR signals, shared, dirty and snoop valid which is an inhibit signal. So this signal will remain high until the caches have completed the snoop and when the signal is de-asserted when it becomes low the memory as well as the requester can then sample or examine the other two signals, that is the D and the S signal they will sample to decide what to do.

Next is who will provide the data? So if some cache has the data dirty that cache has to provide and if the cache does not have the data dirty that is every cache has got the data in shared mode, right everybody has in shared mode plus the memory is also connected to the bus then who should give. So in this case it depends on the implementation. So here we say that the initial Illinois MESI protocol allowed cache to cache transfers. So block was provided by the cache and not the memory. So here one of these caches will give the block to the new requester and the memory is not bothered. Okay.

But when you have this, which of these caches will give? So you need a priority scheme and a selection logic to do this. So it makes the protocol complex. Okay. So implementation of the protocol will become complex because memory is not going to give the data you have to select among these three processors who will give the data. So

therefore these commercial systems do not use, so they avoid using cache to cache transfers because the implementation will become complex. These two implementations SGI challenge and Sun Enterprise, they use the cache transfers only for modified data, that is if the data is changed only then they will use.

So SGI challenge says the cache which has the block in state M will give the data. But when it gives the data, it also changes the memory so that memory is up to date. The other variant is Sun Enterprise which says that the block which is in M state gives the data but memory is not updated. So you will not update the memory here. So if you do not update the memory then there has to be an owner which is implemented using the MOESI protocol with the O bit. Okay.

So only dirty blocks will be provided by the caches, clean blocks will be provided by the memory. So that is the overall conclusion of who provides the data. Okay. Now dealing with writebacks. So writebacks actually do not interfere with any of your coherent logic because coherence comes into picture whenever we have something shared. Right. So if there is nothing shared and I am writing back a block because I am the only owner of this data, then there is no interaction with any other caches.

So dealing with writebacks has to be handled because it is going to generate bus transactions and actually two of them. So what do we do here? We first have to write this block to the memory and also bring the new block because I am this block is being evicted from the cache. So cache wants to remove the one block and bring the other one. So there are two bus transactions happening here and to reduce the processor waiting time, we want to do the reads fast. Okay. So what we will say write the writeback happen later but let me do the reads first and then this will add to some complications. Okay.

So let us see what is the scenario. So this is the cache, it has the block x equal to 5 and now a read miss comes. So when a read miss comes before bringing y we have to write the x but I would not write the x all the way to the memory but I will say put the write x equal to 5 into the write buffer. So what happens here? x equal to 5 comes and sits here, eventually it will go to the memory and in step number 2, I am going to bring y so this gets replaced with y equal to 88 and then the processor can begin using this. Okay. So now what has happened? I have this extra entity which is holding some data which is not there in the cache and also not there in the memory. So this adds to more issues because now the coherence logic or the snooping controller also should look at the write buffer. Okay.

So we have the processor, cache the write buffer and the snoop controller which was

initially only considering cache tags, it was only interested in looking at the tags of the cache, it is now also supposed to look at the write buffer. So it has to check two things. Okay. See now we are adding more and more features to the cache controller. Okay. So the snooper has to check the write back buffer as well as the cache tags. Another thing which can happen is, this x was sitting here and in the meanwhile because of some request coming onto the bus, this x was given by the write buffer.

Okay, so the x is no longer present inside this. So here the processor or the controller was supposed to write back this x, eventually it had to write back this x to the memory, but in the meanwhile somebody else took this x because of cache coherence. Right. So this was taken away and hence we have to cancel the write back. So write backs also will have to be canceled if they are picked up from the write buffer. Again more logic to be implemented in the controller. Right. So with this I have given you an overview of what all things we need to make a uniprocessor cache capable of getting integrated into a cache coherent multiprocessor system.

So in part two of this lecture we will look at the basic design organization and then the correctness requirements. Okay. So we will stop this class here. Thank you so much.