Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 06
Lecture - 34

Lec 34: Multi-Level caches with an Atomic Bus (2)

Hello everyone. We are doing module 4, Snoop-based multiprocessor design. This is lecture number 5, where we are continuing the topic of multilevel caches connected to an atomic bus. In the previous lecture, we have seen that inclusion is a good property which helps us to avoid the bus snooper at the L1 cache, but it is not easy to maintain. And in this lecture, we will see various ways of maintaining inclusion. So the first thing we are going to consider is, can I maintain inclusion automatically? And if not, what extra things we need to do so that inclusion can still be maintained? So the easiest way to maintain inclusion is you use L1 as direct map cache because if you recollect all the set associative properties of L1 created problem.

So remove that set associativeness in L1 and make it a direct map cache. So when it is direct map, it always has a single block, it either removes that block or it keeps that block. So there is no history-based LRU decision logic and hence there won't be a conflict of decisions between L1 and L2. So direct map L1 cache and the L2 is definitely much bigger because if L2 is small enough, it is of no advantage to us.

We want the back end cache to be bigger than the first level cache. Right. So automatic inclusion is guaranteed when I have an L1 as direct map cache. It's very easy to do this. Here we can guarantee this by using the same block sizes. When you have same block size, it also says, inclusion says that when I am bringing a block M3 here, that is when I am bringing M3 inside this, I copy it in both. Right.

So you have to make sure that M3 is loaded in both the caches, the block sizes are same and the number of sets in L1 is much smaller than that in L2. That is L2 is much much bigger than L1. So if you guarantee these three points, then inclusion will be automatically maintained. Okay. So this is an easy configuration and so we can say that, if L1 is direct map, automatic inclusion is guaranteed. But in practice, if you see, making L1 as a direct map cache will put further restrictions on the hit and the misses of the cache and hence we want L1 to be set associative, not a very large set associative but a reasonable set associative cache.

So if we want to do this, then inclusion is not automatically guaranteed. We know that this will create problems. So how do we solve this? Okay. So can we enforce and make inclusion explicit? Okay. So I am not going to leave it to the system to manage, but can we be alert and keep maintaining inclusion as we keep accessing the blocks. To do this, we are going to use what we are used to doing now is the coherence related protocols and the actions. So the coherence mechanism is in place.

Can I extend that between L1 and L2 so that inclusion is guaranteed? So what do I mean by this? You evict a block from L2. When I evict a block, probably inclusion is maintained or not maintained but we enforce, we say that when I am removing a block from L2, I should also remove it from L1 because the earlier problem was M1 was there. So here M1 was there and here M1 was absent. Okay. This was the problem I was handling that L1 had a block which L2 did not have. Okay. So why did L2 not have? Because L2 deleted that block. Okay.

So it deleted this block. So whenever you delete a block from L2, you make sure that the same block is also deleted from L1. Hence, inclusion will be maintained. So when L2 evicts a block, you send that address to L1 and force L1 to also remove the block M1. Okay. So if L1 removes it, you are safe.

If the block sizes are different, that is L2 has a bigger block and L1 has a smaller block. Then when you delete one block from here, it might lead to eviction of multiple blocks at L1. So more blocks will be deleted. Of course, there will be a performance penalty but still it maintains inclusion. The third thing to take care is when L1 deletes a block, that is evicts a block and it has to be returned back, it should tell L2 and L2 has to take care of the remaining part of eviction because when you are writing back a block, L2 has to take care of the coherence aspects related to this block.

So now we are going to look at how do I maintain inclusion with a still L1 being set associative. So now imagine a setup when you have two levels of cache. So while guaranteeing inclusion, how am I going to handle the processor request and the bus transactions? L2 is doing the snooping on the bus and L1 is talking with the processor. So keep this picture always in your mind. So when L2 performs a bus snoop and it sees a BusRdX, what does this mean? That there is another processor in the system which wants this block and L2 knows that the particular node has this block.

So L2 has to delete this block. So when L2 has to delete this block, what should we do? We should also tell L1 to delete the block. We should, we should also tell L1 to delete the block because we want to maintain the subset property. So while doing this, we have some variations or design decisions which can make the task easier and optimal. So we

will look at them.

The one is when I am propagating bus transactions from L2 to L1. Okay. Let me give you the picture of what we are looking at. This is the bus and when something comes to L2, L2 has to tell that to L1. Okay. So this is handling the propagation of the bus information to L1. So should you propagate all transactions? The first question comes.

So I am saying whatever happens on the bus, L2 sees and tells that information to L1. So should L2 keep on telling everything to L1? Okay. So all the transactions which are coming here for multiple blocks, L2 sees them and also tells L1 that this block is being read, this is being written and so on, so that L1 can take appropriate actions. Now what is the drawback with this is that when you have several messages coming onto the bus and all of them are sent to L1, you are actually disturbing L1 and L1 is closest to the processor. It is in high demand, it is very high performing module and if every request from the bus reaches L1, although L1 is not snooping the bus but L2 is propagating everything, the performance of L1 will go down. Okay. So we do not want that to happen.

To solve this to some extent, you can say that I will duplicate the tags of L1 and let the processor tags be different and the L2 side tags be different. Okay. So whenever a request propagates from L2 to L1, we will only compare the tags on the L1 side, on the L2 side and processor tags are different. But I do not want to do this, can I do something else? Can L2 decide which information to send to L1 and which to not, which not to send? So for this what we could do is if L2 somehow knows what blocks L1 has, then it will help us. So this can be easily done by maintaining an additional metadata information called the inclusion bit. So L2 keeps the inclusion bit with it, when that bit is set, it says that this particular block is also with the L1 cache. Okay.

So that optimization I can easily do. So instead of propagating all transactions, I am going to maintain an inclusion bit, so that I can selectively send requests. Okay. So not every information will go to the L1, but only selective information will be sent. Now the reverse direction, L1 to L2. Okay. Now we will consider L1 writes a block and how does it inform L2? L1 has modified a block, it has to tell L2 about it so that, why should L2 know about this information? Because L2 was talking to the coherence hardware and any transactions happening with respect to this block, L2 should be able to take the correct action.

Hence L1 changing a block, that information has to reach L2. Okay. So when I want to do this, there are two options, L1 is a write through and L1 is a write back cache. If L1 is a write through cache, for example, you want that, right. You want L1's information to

reach L2, all the writes which are happening, it should tell L2. And the easiest way to do this is use a write through cache, but if you have a write through cache, it is going to consume bandwidth of L2 because L2 will be busy here. Right. So this link will be very busy consuming substantial bandwidth.

Also to ease out the pressure or stall time of the processor, we may put a write buffer. Right. So we said that whenever I have a write through cache, let it go through a write buffer. So when I do this, I can avoid the stalls of the processor, but still I need extra hardware between L1 and L2. So write through of L1, if I use that, it comes with these two trade offs, that is the bandwidth reduces and the write buffer causes extra area. The second option is, can I use a write back cache? So if L1 is write back, now we have some problem because if L1 is write back, L1 cannot tell L2 that it has changed the block. Okay.

By default, it won't be propagated. So we need to ensure this explicitly. Okay. So in this, L1 has to purposely tell L2, without sending the data, it should say that it has changed the block. In write through option, the new data goes to L2, in write back option, only the information of change goes to L2 and the data does not go. Okay. So data is in L1, but the information reaches L2.

So I can add one extra state bit to L2, which says the data is modified, but do you have the correct copy? No, the copy is stale. So L2 says that I have the data in M, but stale state. So it is a modified stale state because the correct data is with L1. So anytime a request comes for this particular block, L2 has to speak to L1 to get the correct data. Okay, so this is how we can modify the protocol to update L1 and L2 and maintain inclusion.

Two side remarks are that there is also concept of exclusion, which is, we looked at inclusion. Exclusion is a cache property which says that a block cannot be in both the caches. So no block is there in more than one cache. Okay. And if L2 design has restrictions to maintain inclusion, but new designs want to maintain inclusion and if you do not want to do this extra work, which we have been discussing, then some manufacturers also think that can we still snoop L1. Okay. So we had discarded this option, but there are some designs which still wish to snoop L1.

Okay, but these are side remarks about the allied topics, but not the main flow of the lecture. Okay, moving on. Now we have maintained inclusion. Next thing is we need to handle the coherence transactions. So propagating the coherence transactions within the hierarchy.

So I am talking of intra hierarchy protocol, when the request coming from the processor, they go out onto the bus and request coming from the bus, rise all the way up to the processor. These two things we have to now cater to. Because now inclusion is guaranteed. Okay, so processor outside. Here processor requests, they go only up to L1. Right. So processor requests were going to L1 and I have to somehow guarantee that they will reach all the way to the bus.

So all these requests will first go to L1, then they will go to L2. L2's bus snooper will send them onto the bus and so on. So that green line is showing the request path going out from the processor to the bus. Now when I am sending a processor read, processor read goes, it goes eventually as a bus read. So when a bus read goes, you will get the data block, you will house it in L2 as well as L1, both of them either in the shared state or the modified state. Right.

So when you do a read, it will go all the way to the bus and you will load the block in the shared state at both the levels. When we do a processor write, now during write, we write the data to the nearest L1. So data gets written here. Right. Data won't be written at L2. But I want, first thing is I want the block in writing permission, that is in modified state. So we are going to send a bus request when the block comes, that response has to go all the way in the hierarchy and it will be in M state in L1, but it will be in modified stale state in L2.

So this is how I will distinguish between a read request and a write request. Read is common, both have the same state, whereas in a write, you will have L1 in modified and L2 just indicates that the block may be stale. So that is processor going outside, that is request from processor traveling out. Now request from the bus traveling in. So bus requests have to propagate upwards, then they have to go in the up direction until they find the block in the correct state it is asking for.

So if you, I can reuse this figure and say that if a request comes onto the bus, L2 snoopers sees what is required. If it is simply a read request, it can take care, but if it's a write request and L2 or this node has the data in a modified state, then L2 has to go up, up to L1, look at the data and bring that updated data and send it out onto the bus. Okay. So you have to go up and bring the new content. So that's what we are going to see in this slide. If the request is to flush the block that is we have to provide the block, then we have to travel in the upward direction till I find the block in the correct state from L2 to L1 and if you have three levels, then three level travel. Okay.

So that's what we will do going up. The other possibility is invalidations. For invalidations, you have to reach all the levels to invalidate the block. So block is there in

L1, block is there in L2, invalidation request comes, it has to be sent to all these levels and until you finish all the invalidations, you cannot give the acknowledgement. You can't say that you have finished the invalidations and until then definitely the bus is held up. So when invalidations come onto the bus, you propagate them upwards, everybody finishes, then you release the bus. Okay.

This is slow. How do I optimize this? I can optimize this by using the concept of commit which we discussed in the previous lecture. Here we say that L2 which sees the invalidation request, commits onto the bus that yes, I will invalidate and it releases the bus that is it sends the acknowledgement and in the background L2 will invalidate its own copy and also propagate information to L1 to invalidate its copy. So they will be invalidated later on but we commit in the beginning. So we commit the invalidation. While doing this, it is possible that other requests come onto the bus while we haven't done the invalidation, some local requests also come.

So the local node has to guarantee that the invalidations will maintain the order. Okay. So you need to maintain the order of invalidations with respect to the other transactions coming on the bus. So I said invalidate block B, maybe read block C, many things are coming onto the bus but this was pending for us. Okay. Because our invalidation was pending, we first make sure that we finish that only then cater to the other requests and why? Because order of the bus has to be maintained for correctness. Okay. So this optimization is okay if I have a single transaction bus and not so many requests are happening within the node also.

But when I release this assumption of a single transaction, multiple transactions can be pending at the same time, some of them half done. So how do I solve this problem? So we will leave it for now but when we take up that topic, we will try to solve that issue. This is very simple that earlier we said, we will have two tags, one for the processor and one for the snooper. But we are going to get rid of this by maintaining inclusion. When I maintain inclusion, only the L2 caters to the bus, L1 caters to the processor. These tags are mainly used by the processor, these are mainly used by the bus snooper.

Definitely L1 also can use these tags but only sparingly. Okay. Whenever there is a miss, it will be required to access those tags and therefore I do not need to duplicate L1 tags, area is saved. I do not need to snoop on L1. Okay. So the bus snooper also we have avoided and this way we have tried to solve the multi-level cache hierarchy problem. I have multiple caches, everybody has their own tag storage but extra protocol interactions between L1 and L2 which will guarantee inclusion and coherence. Okay.

So all the correctness requirements are very straightforward as long as inclusion is

maintained. So our discussion said how to maintain inclusion. They are also guaranteed because all the transactions are propagated up and down the memory hierarchy. Then we also say that the transactions need to be held up until the propagation happens. That is L1 and L2 if everybody has to invalidate, let everybody invalidate, only then you release the bus.

If you do this, then the bus transactions are also taken care correctly and the serial order of the operations will also get maintained. But if I hold up the bus transactions, I am going to incur a performance penalty. It is okay if I have a single transaction bus but when I relieve this constraint, we will have to solve the newer issues which will come up. Okay. Right. So multilevel caches are guaranteeing correctness because I wait until the information percolates through the hierarchy and then only we release the bus and we are guaranteeing inclusion. Okay. So with these two things, this multilevel cache with an atomic bus topic is complete. Thank you so much.