Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 06
Lecture - 33

Lec 33: Multi-Level caches with an Atomic Bus (1)

Hello everyone. We are doing module 4 on snoop based multiprocessor design. This is lecture number 4 and between lecture 4 and 5, we are going to consider the second variant of our module where we have multiple levels of caches with an atomic bus. So the module overall we were targeting to consider correctness requirements and then these four design options, single level cache with a single transaction bus and the multilevel cache. So I am going to do this point now, multilevel cache with a single transaction atomic bus as part of this lecture. Right. So there has been a trend of multilevel caches because why do we need to consider this variation because we just not have a single level cache from the processor. Okay.

So in the previous module or previous lecture, we saw that there is a processor, a cache which is connected to the bus and then we also understood the interactions of this cache with the processor and this cache with the bus. That is how does the particular cache controller handle the snooping related requests and responses. But now when I have a multilevel cache, how do I handle the same problem is the objective of this lecture. And why do we have multiple level caches? Because they help in faster processor request or response time because when you go to a first level cache, it is supposed to be much faster and to reduce the memory latency gap, we want to introduce more and more levels of the cache in the memory hierarchy.

So the first level cache is smaller, this next level is bigger than the first level and so on. So most of the recent systems have multiple levels. They could have the second level on the chip or off the chip and even the most recent ones have a third level cache which is also possible, it is on the chip or it is off the chip. So how do we handle coherence compliance in this scenario of multiple level caches? Because when a request comes on to the bus, the cache closest to the bus is going to see it and not the cache closest to the processor. Whereas all the requests from the processor will be going to the cache closest to the processor and they will not be visible to the bus transactions. Okay.

So we have to handle these problems. So we have recently multiple levels of caches, either first level, then a second level, probably also a third level cache. For coherence

compliance, we have to handle situations when the processor makes any changes to the first level, how does the bus see it? Right. So the bus operations would not be able to see this happening and the bus transactions which are coming, they are directly visible to the cache closer to the bus and definitely they are not visible to the first level cache. So how am I going to handle this problem? Okay. So we have seen this in the previous lecture that I have a single level cache and this cache has a bus snooper which snoops onto the bus and then all these colorful lines are the snoop related request response and wired-OR signals and so on. So that is the bus interconnect and the blue color snoop controller snoops onto the bus.

The processor side FSM is here and that directly talks with the cache. So these two FSMs are tightly coupled because they both work with the same level of the cache. Okay, now when I introduce another level, so for understanding the concept we will only use two levels but it applies to a multi-level cache as well. If you have three or four level caches the same logic arguments will apply but we will take for example a two level cache. So from this slide where I had P0 followed by a cache, now I have that cache replaced by two caches.

One is the L1 cache and the other is the L2 cache. Mostly the L1 cache is on the processor chip, the L2 cache may or may not be on the processor chip. Okay. The same two FSM's now you see they are separated, the bus snooper is still with the bus or closer to the bus, the processor FSM is closer to the FSM but now this only speaks with L1. In the previous slide the processor side FSM was in contact with the cache whereas here the processor side FSM is in contact with the level 1 cache and not with the level 2 cache.

Okay. So, do these two communicate information across each other because they are disconnected. So that is the problem we are trying to handle that how do I connect these two types of informations coming from the processor to the bus snooper which is attached to the L2 cache. Okay. So there are many reasons or many solutions. One solution is that can we make the bus snooper snoop the L1.

Okay. So, the blue snooper which I have attached with L2 can that also go to L1 that is one possibility. When you attach a snooper to L1 what does it result into? One is it will need to do tag comparison so you will have to duplicate the tags at the L1 level and because L1 is closest to the processor you don't want to stall the processor. Hence duplicating the tags will be required. And the other thing is if you have a snooper attached to the L1 the snooper will need extra pins to the IC to connect to the bus which will add to the real state of the IC. Okay. So, if I have a multi-level cache hierarchy then can we independently have two snoopers instead of just the L2 snooping onto the bus connected. But there are problems with this. So, we are going to discuss three reasons

why this is not good.

Okay. So, if L1 on the processor chip has to snoop then I need a hardware which would need extra pins. Okay. Right. So, here I have added a new bus snooper, the green color one which also snoops onto the bus. So, the blue snoops onto the bus here and the green also snoops onto the bus.

So, when the green snooper has to access the bus it would need extra pin connection. So, from this I will need, I will need extra pins to connect to the bus apart from those which were required by the snooper. Okay. So, there were existing pin connections of the snooper now I need extra connections. So which is not a desirable option. So, adding another bus snooper adds to my cost.

Okay. Then adding another bus snooper would also consume more area because I will have to duplicate the tags of L1 and you recollect why do you need to duplicate because snooper as well as the processor are going to do tag comparisons and hence having duplicate tag helps so that the processor is not stalled during the comparison. Okay. So, the green bus snooper is now given a dedicated tag array for L1 and the processor side FSM also gets, this is a processor side FSM which has its own tag. So, these are the duplicate tags we have added. Now when you have duplicate tag, it helps you in the latency, but again adds to the area.

So, this is going to consume extra area on the chip. So, duplicating level 1 tags is performance wise okay, but costly in area. The third reason, third reason is we are actually duplicating our effort. Why? Because most of the blocks which are there in L1 are found in L2 as well. Okay. If you have a block sitting here, very likely that particular block is also available or its copy is also available in the L2 cache.

So, why are we doing the repeated effort both for L1 and L2 when we can do it only at the L2 level and manage the correctness aspects. So, that is the third reason that we are duplicating efforts because most of the blocks in L1 are also present in L2. Okay. So, this property if you have heard earlier, this is called the inclusion property where the L1 cache is a subset of L2, okay, right. So, the last observation of inclusion, can I use that as a solution for my multi level cache hierarchy. Solution in the sense I am not going to have a bus snooper for L1 and can the bus snooper for L2 alone help me in doing this.

So, while doing this I need to ensure that the inclusion property is preserved across any designs, okay. So, if a memory block is in L1 then it must also be in L2 that is the definition of inclusion which we said that L1 is a subset of L2. Okay. So, if my L2 is big enough and definitely it has to be bigger if it is an inclusive cache then my L2 definitely

fits inside this box, okay. So, that is the subset property. The second thing to be guaranteed while doing inclusion is that if there is a block in state M.

So, right these are my two caches and I have a block in state M in the L1 cache. So, this is a modified block in MESI protocol. It could be ownership state in the MOESI protocol or it could be Sm state in the Dragon protocol. So, this state says that the block is dirty and modified inside this L1 cache. Now, any request which comes for this block on the bus.

So, if I have just the bus snooper sitting here with the L2. So, L2 should somehow know that the particular block is in L1 cache and also in the modified state because L2 snooper has to take appropriate actions of providing the data to the bus from this particular cache block, okay. So, we need to somehow make sure that I also mark that block as M in the L2 cache, okay. So, while doing inclusion, we have to make sure that this guarantee that is L1 is subset of L2 and the two states are consistent, that is L2 knows what has changed in L1. If we can do these two things then the inclusion becomes a good solution for my multilevel cache problem. Okay.

Right, so this is the left hand side was my proposed design. Now how does inclusion help me? So, bus transactions which are relevant to L1. If there is a transaction which is relevant to L1, which will be handled by this bus snooper, is now also relevant to L2. Why is it relevant to L2? Because L1 is a subset of L2, right. So, when we guarantee this, anything which comes on L2 is anything which comes on the bus is relevant both for L1 and L2.

So hence if only L2 snoops the bus it is good enough for us, okay. So bus snooper is relevant to L2 also as well as L1 and L2 both, okay. Then bus transactions that request a block which is in M state in L1 or in L2. Wherever it is in M state L2 snoop is sufficient because L2 somehow knows that L1 has the block, okay. So the task of bus snooper, green one is actually done by the blue one. And if you do this then we do not need the green bus snooper. So what we have now? We have the processor side FSM, the bus snooper attached to the L2 and the processor side attached to the L1 and whatever protocol modifications we do during this topic is going to handle this whole business.

So we got an idea that inclusion is the magic solution to solve this but is inclusion easy? Definitely not, inclusion is not a very trivial thing to maintain, okay. Now one more thing that it is very straightforward because my L2 cache can have 1000 blocks and out of those 1000, 200 blocks are in L1 okay. So where is the difficulty? The difficulty comes in due to the arrangement or the design of the individual caches and how we handle them that is when we load a block we have to evict certain blocks. So during

block eviction the type of victims which we choose will also affect the inclusion property, okay, because to begin with even if we have guaranteed inclusion during runtime it may not be valid, okay. And then when you have inclusion, the processor references which go to L1, now these have to be transferred or informed to L2 because all the references with respect to L1 are going to change the state of the block in L1.

They are not changing the state of the block in L2. So how do I maintain inclusion by also propagating this information? One thing. Second is if the bus transactions come on L2, only L2 knows about them, L2 changes its state so now this state change has to be propagated to L1. So inclusion actually also means that they know each other's information as well as the blocks also are in both the caches, right. So whatever is in L1 is also in L2. So information should be propagated from L1 to L2 as well as from L2 to L1.

We have to establish this requirement to maintain inclusion, right. So L1's state changes need to be informed, L2's state changes should be informed to L1 and then we need to propagate information from L1 to L2. Okay. So will it be always guaranteed as we discussed? May not be because even if I started with an L1 subset of L2 during runtime this may not hold. To understand this we will take an example, okay. So here you can see I have got two caches L1 and L2 and I have defined some terms for that A1 is the associativity of L1, N1 is the number of sets, B1 is the block size and hence this is the total cache capacity of L1.

Similarly for L2 I have A2 associativity, N2 number of sets, B2 block size and product of these three as the capacity, okay. So remember this configuration A1, N1, B1, A2, N2 and B2, okay. So let us understand why inclusion may not hold in normal cases. We have two caches L1 and L2 and both of them probably are set associative and given that L1 is set associative, what happens is whenever L1 wants to evict a block, it evicts a block based on the history of accesses which is the popular LRU based policy, okay. So whenever you are using a history based access to evict a block, the history of L1 is different than the history of L2.

So L1 decides to evict for example a block A, it evicts block A and it doesn't have that block A that's fine because L2 may still have the block A but when L2 wants to evict a block, what type of a replacement policy and what type of a decision it takes may affect the inclusion because if L1 removes a block and L2 also removes it then inclusion is guaranteed but if L1 keeps a block whereas L2 removes that block then you will violate inclusion, okay. So this happens when we are using a history based replacement policy. This also happens when I have multiple levels of cache. For example L1 comes in two varieties we have a instruction cache and a data cache. So when you have two caches

even if it is a direct map cache you have two copies so you have set 0 in instruction, you also have a set 0 in data cache. So effectively if you see it becomes a set associative cache, okay. So even here the first condition might hold and the third condition is when you have different block sizes that is L2 block is much bigger than the L1 block and you can easily understand that the, when the block sizes differ the way the addresses get mapped to different locations will also differ and hence it will again become non-trivial to maintain inclusion, okay.

So we are going to see examples of all these three conditions to see that inclusion is not guaranteed by default, okay, right. So set associative L1 is the first one we will see, then we will see the instruction cache and data cache and the block sizes example. Reason one, set associative L1 cache with a history based replacement, history based, our popular LRU or its variants. So access history of L1, the history of L1 and the history of L2 will definitely be different because L1 is going to cater to requests coming from the processor. So whatever processor is accessing that is going to be its memory that is that its history whereas for L2 cache which is much bigger and farther, you are only going to send some of the requests from L1, okay.

So the history of accesses in L2 will be definitely different than that in L1, okay. So let us take an example. So we have L1 LRU replacement. I am using three blocks M1, M2, M3 or addresses you can say which go to the same set of L1. They have the same block size so we are not going to touch the block size in this example. L2 is much larger, okay. So L2 is much bigger and we are saying that both M1 and M2 and M1, M2, M3 all three belong to the same set so they map to the same set.

So M1, M2, M3, they go to the same set in both the caches, okay. Now given this scenario see what happens, okay I will show you the example right. So you have here L1 I am just showing one set it is a two-way set associative cache. L2 is shown to be four-way set associative and a huge cache. So it is only concentrating on set-i. You can see M1 is here, M2 is here, M1, M2, M4, M5. So that is the load of each cache block and now what happens.

A read comes from the processor. So processor is sending a read M3. M3 if you see it is going to be a cache miss because M3 is not there in set 0 and M3 maps to set 0. So what is L1 going to do? It is going to invoke the LRU algorithm and it decides to evict M1, okay. So it decides that I will remove M1 because that is the least recently used block, okay. So it makes space by removing M1 but it still needs M3.

Now who will give M3 to L1? So the request for M3 goes as a load request from L1 to L2. So it is not a read M3 but a load request on M3. So M3 comes here, okay. There is a

cache miss on M3 because M3 is not there.

L2 invokes LRU and decides to evict M2. Fair enough for both the caches because their histories are different, okay. So it evicts M2. Now what has happened? See the contents have become like this.

M2 is removed, right. So we removed, sorry, sSo we removed M1 and wrote M3. Here we removed M2 and replaced M3. Here I have M3 which is there in set 0 both because that was the most recent request but because of this request, what has happened is I have a M2 still sitting in L1 but there is no M2 in L2, okay. So this way we have violated the inclusion property. Why did this happen? Because of the history based replacement policy and L1 was a set associative cache, okay. Now we look at the second example or second reason where even if L2 is a direct map cache, the M1 and M2 may still fall in the same set, okay.

Here M1 and M2 are in the same set because L2 was set associative. Even if L2 is a direct map, you can still have the same problem where M1 sits in the same block whereas M2 is also eligible to sit in the same one and M3 is also eligible to sit in the same location because of direct mapping and the way the addresses are, okay. Then inclusion is difficult to maintain here. The second is, if L2 is a two-way set associative cache instead of a four-way, you can still have M1 and M2 both sitting here but again it may replace the wrong block.

A quick example, L2, the first one is L2 is direct map. So if L2 is a direct map cache, M1 and M2 both can sit here but one at a time, okay. M1 is one block, it sits in set 0 or M2 sits not both. But L1 can house both of them together. L1 can have both M1 and M2 but here only one, only one of them can sit here.

So again inclusion cannot be guaranteed. Second condition is, when my L2 is a two-way set associative, both M1 and M2 sit here. So currently satisfying but in future for any request if L1 removes M1, it is possible that L2 removes M2, okay. So if it removes this M2, we still have the M2 sitting here, okay. So that comes as an issue to maintain inclusion because of history based replacement policies and a set associative L1 cache, okay. So what is the solution? So the effect of this thing is that inclusion can be violated if L1 is not a direct map cache because it was a set associative cache, it used a replacement policy which selected a different victim than what L2 selected.

So no matter what is the configuration of L2. So L2's configuration has got no impact actually the problem comes because L1 is not a direct map cache, okay. So that was one reason. Our second reason was, if I have multiple caches at a given level that is I have

instruction cache and a data cache at the L1 level. So it, you can say that it becomes the same type of a problem if I have M1 sitting here and M2 sitting here. So if you merge them it becomes a two-way set associative L1 but now I have a direct map L1 with two copies, instruction cache and data cache.

So these two blocks which are sitting in instruction and data cache, they may collide into the same location of L2. So L2 may still have both of them in the same location and there may be a problem again as the previous case, okay. So this is, left side you can see L1 instruction cache, it has got two blocks data cache two blocks and they both can independently house M1 and M2 but because both of them map to the same set here, one of them. So this can only store one of them and hence again inclusion will get violated.

So we can quickly see a sequence of events which leads to this. Suppose L1 does a read of M1 then L2 will bring M1. So this is in the beginning when the caches are empty, you have loaded M1 and M1 in both then M1, sorry L1 wants to read M2. So it has space because it is a data cache miss whereas the red one was an instruction cache miss. So even L2 misses but this time it removes M1 and then brings M2, okay. So M, here L2 is going to remove this to place M2 and hence again inclusion gets violated.

So when I have a split cache, I have a problem. The third reason was different block sizes. So for this you can see an illustration again. Here I have given the configuration using both has direct map. Okay. So the previous problems came because the L1 cache was a set associative cache. Now we will say that let me make a direct map but if the block sizes are different, we will have a same problem.

So here block sizes are different one is housing two blocks and one block. Okay. So L2 can keep two at a time. L2 is bigger, it stores 16 words whereas L1 stores 4. Associativity are same.

So we have both direct map block sizes is double, cache size is double. So the illustration is going to demonstrate that if L1 has some addresses 0 and 17 at the same time, that is it can keep both of them together, L2 may not be able to keep it together, okay. Violating inclusion. So we will quickly see an example. So this is the L1 cache and this is as time progresses. Okay. Actually this is the data portion or every column is showing different addresses which it can store. So set 0 can store address 0 and also address 4 so means it has to remove 0 to store 4. Okay, because these all addresses 0, 4, 16 they all hash to the same set.

They all hash to the same set. Similarly 1, 5, 17 all hash to the set 1. So that is the meaning of this. So I have indicated all the addresses which can go inside the set. Same

thing I have done for sets in the L2. So this is a direct map cache but the block size is 2.

So you can store two addresses in one block. So the set 0 can store (0, 1). If it removes this it can store (16, 17). So this is the pairs of addresses can go at a time. So if you see this example, you have in L1, set 0 has the address 0 and set 1 has the address 17. Okay. So 0 and 17 can be there but if I go to L2, L2 can either have 0 or it can have 17.

It cannot have both of them at the same time. So again this example shows that having different block sizes inclusion cannot be guaranteed. Right. So we have seen that inclusion is a helpful property for me to handle coherence of multilevel caches. But it is not so straightforward to handle. So with a set of examples we have demonstrated that inclusion is difficult to maintain and so we will consider possible solutions for this in the next lecture. But with this we finish this lecture. Thank you so much.