Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 10
Lecture - 56

Lec 56: Relaxed Consistency Models (1)

Hello everyone. In this lecture, we are doing module 7 on memory consistency. This is lecture number 3, where we are going to discuss relaxed consistency models. Okay. So, what are these relaxed models? In the previous lecture, we have looked at sequential consistency and we ended the lecture saying that sequential consistency is too restrictive, it has a lot of limitations. Why? Because we want a reasoning model for the programmer to derive the possible outcomes. And while doing that, because of directory protocols or the network, there is lot of latency in servicing the memory accesses. Right.

So, if you have a cache miss, you have to go to a far off memory bank to access and while this long latency is being serviced, the processor remains idle. So, we want to optimize on this and say that let the processor execute future instructions which are possible. So, essentially overlap the memory accesses, but sequential consistency says that you could not cannot do that, you cannot do certain reordering. So, we are putting restrictions on the compiler, on the hardware which is affecting the performance. Okay.

So, we are restricting performance optimizations to preserve sequential consistency, but the high cost that is the latency of memory access is motivating us that, we permit the overlapping of reads and writes. So, we want to allow reordering to some extent while still preserving the semantics. So, when we allow reordering, we come up with newer consistency models which are called relaxed consistency models. So, when we say relaxed consistency, what are we going to relax? At the end of the day sequential consistency must be met. So, sequential consistency said, you need to preserve program order and atomicity.

These two requirements were there. Now when I am relaxing, I want to relax one of them or both of these. Okay. So, let us now divide these two conditions into smaller parts. If I say program order, program order maintaining means reads and writes ordering. So, we have four orderings, write followed by read, read followed by write and all the four possible combinations.

So, which of these four combinations I am going to relax will decide my newer

consistency model. Similarly, write atomicity said that, a write should be visible across the complete system only then we can say that the write is finished. When I do a read, the value which I am reading should be globally visible by everybody before I can say I have, before I have permission to read that value. So, for write atomicity I can divide it into two parts saying that, am I allowed to read my own value or am I allowed to read others value. Okay. So, using this I have five combinations to relax. Okay.

So, to derive the relax consistency models, the options I am exploring is relaxing the program order by either relaxing the write after the read, the write after the write. So, I am going to discuss models which relax these one by one. So, let us first relax this order. And we, in the previous lectures we saw that we need to maintain the write followed by read order for certain programs to work correctly. Now we are discussing how can we permit to relax WR order. The second is relaxing WW, third one is read followed by read and write.

Now all these are program orders to different locations, remember they are to different locations. The second one is relaxing the write atomicity requirement. Here, when I say write atomicity requirement, we had a write A followed by a read A. Okay. So, in the same process. So can I do this read A because I have done the same write. This is called reading our own write, okay, read our own write.

Am I permitted to do this read A? Well sequential consistency says no you cannot because until this effect is seen by everybody, you cannot do your read. The second condition is, I do a read of B and this B was modified by some other process. Okay. So can I do this read B? Well, I do not know, a sequential consistency says you can do this read B provided the effect of this write B has percolated everywhere in the system. It has finished only then you can do read B. So this read B is reading others write early because there was somebody else who wrote B and I am reading it. So read others write and read own write, these are the two concepts for write atomicity and for this we have two options.

Can I read my own write early? That is, can I read this read A early before others see it? Can I do the read B early because there could be a process far off here which is trying to do read B and it has not seen this B yet, but because that one has not seen B yet, can I read the new value of B? Okay. So these are the two options I am going to discuss. So overall we have five things to consider. So when we say read own write early, it means that read our own write that is the same process reads the value it has changed, before the invalidations or updates have percolated in the system and when I would do this because when I did the write, that write went into the write buffer or it went through the write through caches because I have done the write, it is close to me, can't I access it?

So that is my demand that let me read my own write earlier than others. Okay. So read from the cache before the write completes across the system. So in a cache based system, this will permit me to read the newest values before, even before the write gets serialized.

Remember coherence when a write happens you have to go onto the bus and you get serialized onto the bus or you go through the network to the directory and your write gets serialized across the system. So even before I get onto the bus because I changed that write in my cache, before I go onto the bus, before the write gets serialized, am I permitted to read my own value? Okay. So if I am permitted it would be faster because otherwise it is an unnecessary stall for the processor. So that is the meaning of read your own write early. So while we do these relaxations we need to definitely guarantee or have some assumptions and what are those? One is that one, a write will eventually be made visible. Okay. So we are not saying here, that this write B, so suppose I got the value I do not care about whether the write B went.

I do not care whether this process got the value. Okay. It does not mean that. It means that the far off process should definitely get the value of B eventually. Because and I will, we can conclude that coherence is definitely taking care of this. So a write will become eventually visible to all the processors and all the writes to the same location are serialized. So this is a very easy condition now for us, why? Because if I have no cache then the memory, which is a single point of synchronization is going to manage that, or if I have a cache, the cache coherence protocol easily guarantees this requirement.

The other condition is that all the models are going to enforce data and control dependencies , which is definitely required. You cannot oversee these because that is the correctness semantics of any program. So while relaxing we need to make sure that these two things happen and these become straightforward. There are several reorderings which we are going to do and any relaxations which we will do if the programmer wants to still maintain the order. For example I said, I design a memory consistency model which says W to R reordering. I am not going to follow, that is I can relax the order but the programmer wants this order to be there.

So what can we do? The programmer cannot say that, oh let me run this program on a sequentially consistent system and not on the relaxed consistency system. Okay. So the same program has to run on the system because that is what he was or he was she was given. Okay. So suppose I am running a program on a relaxed consistency model which is relaxing the WR order but the programmer has a requirement that the WR order should be maintained. So what should they do? Okay. So you would say that well the order is not maintained so anything can happen. The outcome is not predictable but the

programmer wants it to happen. Wants it to happen in a sequentially consistent manner, WR order must be maintained, reordering not allowed.

So if that is the requirement then all these models give you something called a safety net. Okay. So they give you special instructions or I would say a package in which you can fit your secured instructions. Suppose these are the two instructions I want to be maintained in the order, we pack them inside a safety net and say that this is the safety net, inside this you follow the program order, inside this do not do the reordering. Okay. You can do all the reordering elsewhere, but within the small segment do not do the reordering. So these are called safety nets. Okay.

So we are going to discuss few relaxed memory consistency models. This table is a summary table and we will see few of those and you will understand all the meanings as we move on. But as a, to give you a feel of what we are targeting, I have put it here. So these left hand side column tells you the different relaxation models. The top one is definitely sequential consistency and the bottom ones are the relaxed models.

These three columns tell the different relaxations in the program order and these are telling about the write atomicity. Okay. Write atomicity we had two relaxations: read own write early, read others write early. And in program order we had three, write after read, write after write, and read after read write. And I told you the concept of safety net. So these are the safety nets available across all these architectures. Okay. So we will see them slowly. And this slide is a summary slide of all the examples which we are going to consider. Okay. So one point of reference for all the examples, when I say example one, it is this top one, when I say example six, that is the bottom right corner example and I am of course going to take all these examples and put them on the slides as we discussed, but this is one place to see all of them.

Okay, right. So let us start with relaxed consistency models with the first model of relaxing write followed by a read. So why do I want to do this because, when I do a write followed by a read, that writes the value which I am writing as a processor, I do not need to use that value immediately, right. So that value will take some time to get written into the cache, if it is a write through cache, go to the next level and so on. Okay, so it is going to take time. And in case it is a write miss, write miss means, I wanted to write the value but that block is not there in the cache. So in such a situation I can keep the write miss pending because the value it is going to return or be effective is not useful for me immediately. Okay. I am more interested in the future execution of my program and my next instruction is a read, so read is more important because I need that value. Write is not very important, so I want to reorder it.

So while the write miss is still pending, the processor can issue a next read. This read may or may not hit into the cache. If it hits, good. If it does not hit, and if I have a lock up free cache, I can still keep this read pending and go on for future reads. Okay. So I want to do this for my performance. So I want to keep the write miss pending and let the processor issue the future reads, so that I get improved performance. Now this reordering which I am talking about is with respect to the previous write from the same processor. So the reordering W followed by R, this is on the same processor but the variables could be different and they have to be different, if they are same we cannot definitely reorder them. If it is this, I cannot reorder. If it is write A followed by read B, I can reorder this, but this is on the same processor. Okay.

So under the write followed by read relaxed order, I have these three models. Okay. So a subset of the table is shown here. The first one is the IBM 370 model. It relaxes this order. Definitely all these models are going to relax the WR order. They do not relax write- write, they do not relax read-write, okay. So that is why the columns are blank. Read own write early. So this one does not permit anything. It says write atomicity has to be strictly maintained. The TSO model, the second one, it says you can read your own write early but do not read others write early. And the third one says you can read others write early, you can read your own write early. Okay. So we have three models: IBM 370, Spark V8 TSO, TSO is a short form for total stored order and PC for processor consistency. So IBM 370 is the strictest order. So this slide is describing the same table which we discussed in the previous slide. What it does? It prohibits the read from returning the value until the write is made visible to all the processors. This is the write atomicity requirement, okay.

See write atomicity, it does not permit. It says, even if you write the same variable, you want to read your own variable, not permitted, if you are reading a variable from somebody else, until everybody in the system has seen that variable, you cannot read. So it is the strictest model. So even if the same processor, right. So if the write is still pending and others have not seen it, then the same processor is made to wait. So the same processor making the change has to wait for others to see the value before it can use the value. So IBM 370 is the strictest in terms of write atomicity. The Spark TSO, that is total store order, it is slightly more relaxed it says you can read your own write early, read own write before it is seen by others. It also says that even before it gets serialized, so even if you before you get the bus you can read your own value, but it says don't read others value early, okay. So the read is not allowed to return another processor's value, until all in the system have seen it. So until everybody in the system has seen it, you cannot read that value. But you can read your own modified variables earlier.

PC is the last one, third which is the most relaxed constraint. So it says you can read, can return a value of any write before the write is serialized or made visible to all. So you can read your own writes as well as others writes early, even before it has percolated across the system, okay. So with these definitions, so when we discuss a relaxed model, we will first look at the definitions, we will look at the table which discusses the different relaxing orders and then we will look at examples to further understand how that relaxation works, where all we can say that this relaxation was not sufficient for my program semantics, or understanding from the programmer's perspective, okay. So we will do small examples which we discussed earlier, a chart of examples was there. We will take them one by one as we move on to understand the relaxed models, okay. So the relaxed first model was relaxed WR. Now this model allows the write and read to be reordered. It has also permission to reorder atomicity. So in this case, if I take the first example, all the models in the relaxed consistency are going to violate sequential consistency in my current example, okay.

So every time you relax or you generate a relaxed consistency model, you have to cross check that, is the sequential consistency guaranteed or not guaranteed? if the programmer wants it? and the model does not support it. The programmer has to use a safety net. If the model supports it, well and good, okay. So let us do the first example. So the first example was the same dekker's algorithm for critical section. So this is a mutual exclusion problem where P1 and P2 want to enter the critical section and they set their respective flags and wait for each other's permission, alright. So in this example, all the models violate sequential consistency, why? See this is the write and this is the read, and we are saying write read relax, right.

So I am going to allow this reordering. So if I allow this reordering, the flag equal to 2 this read will happen before the write, same case with P2, I will finish the read of flag equal to 0 and then I will do the change. So initially flag 1 and flag 2 are 0, both P1 and P2 will read them to be 0 and both of them will enter the critical section, okay. So this way the sequential consistent outcome has not happened and hence we say that, all the models, the three ones IBM 360, TSO and PC violate sequential consistency in example number 1. Now we look at example 3 and 4. So this was example 3, so you can pause the video and reason for yourself.

So P1 has got a write, a write. P2 has a read followed by a read, okay. So am I relaxing write after write? Am I relaxing read after read, in my current model? No. What are we relaxing? We are relaxing write and reads, okay. We are allowing this reordering. We are not allowing reordering of writes and writes. Hence TSO, PC, IBM 370, all of them satisfy sequential consistency in example number 3.

Same with example 4, where even here, you have a write followed by a write and a read followed by a read, okay. So we do not reorder them. The model says you cannot reorder these things. Hence SC will be guaranteed. Next was example 5. Now this is interesting. Here, you have a write which goes here. Now this is read, this is write, we are not reordering them.

So from the program order perspective, we have no problem. This is a read followed by write, we are not going to reorder them. So they will execute in the program order. So we are safe till there. But now the problem comes related to write atomicity. So write atomicity requirement, if you recollect the IBM 370, said that IBM 370 said you cannot read your write early, you cannot read others write early.

So it was the strictest and hence we can say that, it satisfies sequential consistency in this example, because it will guarantee write atomicity. Okay. In the context of TSO, so TSO says you can read your own writes early but you cannot read others writes early. In all these three processes they are not reading their own writes, they are changing a variable and reading somebody else's write. So TSO says, you cannot read others write early. So if you cannot read others write early, this problem of P2 reading A before P3 has seen.

So P2 doing the read before P3 has seen the impact, it is, this one is not allowed, not allowed in TSO and hence I can say TSO satisfies SC. Okay. So IBM 370 satisfies SC because it is very strict on write atomicity. The problem with this example is related to write atomicity. TSO also satisfies SC because these are writes done by others and it does not relax them. Only the processor consistency, the third variant will violate sequential consistency because it says, you can read others write early.

When I can read others write early, P2 is going to read the value of P1 earlier, even before P3 has seen the new value of A. So before P3 sees A, P2 reads A and proceeds and therefore P2 reads new A, changes B and P3 sees the new B, and reads the old A value. And why did this happen, because of the relaxed order of write atomicity. Okay. One more example, so this is the same one in a way variant. I would say that same dekker's algorithm but instead of printing there we had an if loop. But overall it is a write followed by a read order. So similar to example 1, write followed by read. Now write followed by read program order. We are relaxing write and read and you would say well, they will violate sequential consistency because if I allow them to reorder my outcome would not be sequentially consistent.

So let us see this a little more. So here no interleaving can print SC. If I am following sequential consistency I am not going to print both A and B as 0. Okay. You can check it

for yourself ,that A and B both will never become 0 in this particular parallel program. Okay. So let us see program orders what do I have. The program orders I have is instruction 1, then instruction 2 and then instruction 3, instruction 4,right , that is the program order. So this is for P1 and this is the program order for P2. Now if I say that both A and B are 0, so let me start with B equal to 0.

So if B is equal to 0 what does this imply. B equal to 0 says, I printed B first and then B got assigned. Okay. So it will say instruction 2 happened first and then 3 took place. Fine. So print B first happened, that is why B got printed at 0 and then 3 took place, that is B became 1. Okay. What does this imply? I will merge the green with the red, that is I will merge the program orders with this. So this way you can say that because 1 happens before 2, I can say that because 2 happened 2 then 3, so I can say 1 then 4. Okay.

So I am simply following the program orders here. If I follow program orders I can derive this dependency. And from this what do I infer? 1 happened then 4 happened. If 1 happens and then 4 happens what will be printed by 4. So it will print A equal to 1 because instruction 1 has already finished. Okay. So what did we derive, if I start with B equal to 0 I print A equal to 1.

Let us do the same exercise starting with A equal to 0. I would encourage that you pause the video, try it out yourself. If A is equal to 0, looking at the program I can infer that, first A got printed, that is first 4 happened and then 1 happened, correct? Using the program orders, the red program orders, I will say that because it is 4 then 1, 3 happens before 4, so 3 happens then 4 happens, after 4, 1 happens and after 1, 2 happens. So I can derive this from the first one. So this using the program order, using the program orders, I could derive this.

What does this imply? The 2, 3 then 2. So look at 3. 3 is B equal to 1 and 2 is print B. So with this you will say that now B gets printed equal to 1. Okay. So if you start with A equal to 0, you end up making B equal to 1 or you end up printing B equal to 1 and if you start with B equal to 0, you will end up printing A equal to 1. Okay. So no interleaving in any sequentially consistent manner because we need to follow the program orders in an SC behavior, will print both A and B as 0. But if I relax them, I use any of these all 3 models are going to swap 1 and B or possibly swap this. Right.

So it will do this and then this. It will just interchange the location because write followed by read is what I am relaxing, so I will simply swap them. So if I do it this way you are very likely to print A and B as 0 0, which is not permitted in sequential consistency. Right. So with these few examples we got an idea that what are the cases in

which sequential consistency is met, where it is not met. In this particular example sequential consistency is not met, but if the programmer wants it and he definitely wants it because it is a mutual exclusion, critical section related problem and it has to work in that order. The reordering of the read and the write is not permitted in this. Okay.

So how do I solve this? I want to run my program on this relaxed model because it is efficient. It is allowing me reordering but it is not solving my problem. So for this as we discussed we have safety nets. So these safety nets are given to the programmer and it is the programmers responsibility to establish these strict orderings between the safety nets. Okay. The safety nets will be discussed in the next lecture. Thank you so much.