Parallel Computer Architecture

Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

Week - 11
Lecture - 58

Lec 58: Relaxing all Orders

Hello everyone. We are doing module 7 on memory consistency. This is lecture number 5 in which we are going to discuss relaxing all orders. So relaxing all program orders. So program orders were reads followed by writes and we have seen two, three small models which were relaxing one type of an ordering at a time. In this lecture we are going to see how can we relax all possible orders.

But before that, what is the benefit of relaxing all orders? Okay. So all the orders if I want to maintain, it is going to put lot of restrictions on various optimizations at the compiler level and at the hardware level. Because if you have multiple reads to be done one after the other, and there are cache misses, memory latencies, network latency in a multi-core system. So one read is taking more time. So in the meanwhile, can I issue the next read and finish the next read and maybe some more reads after that.

So issuing multiple reads, that is one read bypassing the second read, is a good optimization and I want to exploit this. The second is, I have some writes which miss into the cache and there are subsequent reads and writes. So as long as the data dependencies and the control dependencies in the program are maintained, I should be okay to reorder certain instructions. Okay. So how do I establish this? And all of this has to happen in a transparent manner, because the, all the compiler optimizations are important and they are transparent to the programmer. So still preserving the program semantics, I wish to allow as many reorderings as possible.

And this is definitely going to give me a high performing system. Okay. So when I am relaxing all possible orders, the benefit I gain is, I can have multiple reads outstanding, the reads can complete in out of order manner. These are very good for dynamically scheduled processors which execute out of order execution, use lot of predictions and so on. And if I allow all reorderings, it is going to be very beneficial for all the compiler optimizations. So these optimizations are important for performance and they are transparent from the programmer, alright.

So for a high performance memory system, I need a model which allows me to relax all the program orders. Okay. So under this, we have five models. Okay. The list is given here. The first is the weak ordering. The second is RC which is release consistency.

Third one is Sparc V9 relaxed memory ordering which is the RMO. Fourth is IBM PowerPC model. And the fifth one is Digital-Alpha. Now among these, five WO is the seminal model that is the basic model. This model was extended a little and changed to an RC model which is supported by the Stanford Dash system.

So these two are more theoretical or academic models and the commercial architectures are the other three models, that is your RMO, Power PC and Digital-Alpha are commercial architectures. All these models relax all four program orders. From write atomicity perspective, all read their own write early. Regarding reading others write early only the RC and the Power PC permit reading others' write early. And if I do so much relaxations, you will definitely say they will violate sequential consistency in all our examples. Okay. So we have these five models, two basic ones and three commercial implementations.

Coming back to this table, you can see here, all orders are relaxed. So I have a tick in every column. I have a tick in all these columns because all of them allow to read your own write early and then these two models permit to read others' write early. Similarly they come with this set of safety nets. Okay. So what is the concept of a weak ordering? Now what are we thinking of? We want to relax all orders and when I relax all orders, how do we guarantee sequential consistency? So we see the program made up of several instructions, data accesses, definitely maintaining the data dependence and the control dependencies.

But only there are few places in the program where the programmer intends to maintain sequential consistency. Other places it is okay, as long as the data and control dependencies are managed. Okay. So to enforce the program order, what should we do? As programmer we will insert synchronization operations within the program. So when I write a program, I permit the compiler and the hardware to reorder all the instructions up to some correctness. Of course data and control dependence correctness. But I will insert certain instructions in that, which are called synchronization instructions and within these synchronization instructions, we have to maintain the order.

So when I say, this is my sync instruction, up to that sync instruction, you can do all reorderings but you have to finish the complete code execution up to that instruction and only then start the remaining part of the code. Okay. So that is the simple concept of weak reordering. Okay. So here, if this is my program, right. I have just made one blue box where you can assume all instructions are written and I want to enforce program order. So what should I do? I have to identify at least one memory operation which can act as a synchronization point. So I will say okay, this is the blue part here but after this, so you can reorder.

Okay, I give a permission to the system that you can reorder in this blue portion but you have to finish all the work there before you can start the next chunk of the instructions. So I will put a sync instruction somewhere in the middle. Okay. So I am going to divide this program into small parts and say that these are the locations where I want to do synchronization. So I will do a sync instruction then again some part of the code, another sync instruction, remaining part of the code. Okay, so what does this say? Before the sync instruction we are permitted to reorder.

After the sync instruction, we are permitted to reorder and between this also we can reorder. But when you come here you have to finish this, and only then you can start the pink portion. Here you have to finish the pink portion and only then you can start the blue portion. So this way I have now modified the program to a synchronized program and if you do this, the small portions between the sync instructions are available for reordering, but across them, they cannot reorder. So the pink and the blue cannot reorder with each other. Okay, so memory operations are of two types, data and synchronization.

So data are normal instructions, synchronization are these special safety net type of an instruction. To enforce the program order between the two operations, the programmer has to identify at least one of them as synchronization and the model's intuition is that, if you reorder the memory operations of the data region between the sync ops, it does not affect the correctness. So our intuition is that, if I permit reordering here, between these boxes. So this reordering is permitted and that is not going to affect the program correctness. Only thing is before the sync we have to finish all the instructions and the pink and the blue cannot reorder with each other.

So operations declared as sync are provided by the safety net for enforcing the program order. So we have to finish all the reads and writes before the sync operation and that is we cannot issue next operation until the sync is done. So the read write block has several instructions which can reorder. So within that block, we have a lot of freedom. So when the sync operations are not so frequent, that is my pink and blue boxes are big enough, I have a lot of scope of doing the reorderings and exploit the performance benefits of hardware as well as compiler optimizations. Okay.

So if I take one example, suppose I have a linked list, here. So this is a distributed linked list being managed by several processors and there is a head pointer to this. So these are three processors which are dealing with this linked list data structure and now there is a new node to be added to this. New node will get added maybe at the head pointer. So the head pointer has now to be changed to a new node.

If all three processes decide to change the head pointer, now we have a problem because if we permit reordering we will have issues. So this is kind of a mutual exclusion type of a problem where we need to handle this carefully, that is the shared data item to be exclusively modified only by one process. So what will we do? We will have our normal program but the code segment which changes this head pointer. Okay. So the code segment which is going to change this head pointer, I am going to put it in a safety net. So this is the code segment which updates the head pointer.

In a normal program we will simply write it as it is, but to stop reorderings, what we will do, we are going to encase this within a safety net. So we will acquire a lock on another variable, the synchronization variable. We will do this task and then we will unlock the variable. Okay. So this way we can insert a safety net and the weak ordering says, that it will permit reorderings before this, it will permit reorderings after this and within this segment it can reorder as much as possible but otherwise it does not. Another example, this is a producer consumer type of a problem where P1 produces values.

So P1 produces the values of A and D and they are used by P2. So when A is produced, P2 has to use it. If we leave it as it is, we do not know in which order they will get executed and what values they will get, but as a programmer what you want to enforce is, all these values A and D are modified, when the new values are available only then P2 should start executing. Therefore we are going to encase this again between two points of synchronization.

So we will acquire a lock here. So we will wait until P2 has finished consuming the value. Once P2 finishes consuming the value, it will give us the release on flag 2. Once we get the release on flag 2, we will execute this portion then we will release the flag 2 and we will also release flag 1. Once flag 1 is released, this process can start executing. Okay. Within the two, acquire and release constructs, you can reorder.

So we are permitted to reorder here, we are permitted to reorder above this, we are permitted to reorder after this and so on. So there are small pieces where reordering is permitted. Only thing is, we have to finish all these reorderings, finish that execution only then start things after the top label. While reordering you have to make sure that the data dependence and control dependence are maintained. So keep looking at this one, I think we can reorder all of them because there is no dependency between any, whereas if you look at P2, what all things can be reordered.

If I call this instruction 1, 2, 3 and 4, can I reorder 3 and 4? We cannot, because 4 is using B so that is the data dependence. Can we reorder 1 and 2? Yes, you can. Can you reorder 1, 2, 3, in any order? Yes, we can because they do not use each other's data. Okay. So you have to take care of these data dependences while the reordering takes place. So weak ordering is very simple, you simply have to insert sync instructions in the program.

So here flag is used as a synchronization variable. So program semantics are not violated if you reorder sync instruction, if you reorder instructions between the two sync instructions. Okay, we. So this is the segment where reordering is permitted. And when I declare this flag as a synchronization variable, the compiler or the programmer will map it to an appropriate instruction in the hardware, which will be equivalent to the safety net provided by that particular memory consistency model. Okay. So looking at the left hand side figure for the weak ordering, this is the overall model.

You do several read writes here, then you do the sync, then you do read write, sync, and read write. So you have three blocks here every block can do the reordering but amongst them they cannot do the reordering. Okay. The second model is release consistency. Now this one extends the weak ordering but it distinguishes between the acquire and release.

Weak ordering simply says, sync. It does not say whether I am acquiring a lock or whether I am releasing the lock. Whereas release consistency says that, I will distinguish when I acquire a lock and when I release a lock. So it divides the sync operations into two operations, an acquire and a release. And what it says is, what is the first thing say, this says you finish everything before this to start the things after this. The acquire says that yes, you have to finish things before it but you can only start this after acquiring the lock.

That is the meaning of an acquire, that is you acquire the lock and then execute this critical section. It does not say anything that you cannot execute this critical section parallel to something before this. So the block number one and block number two, they both can happen in parallel because the acquire says, you only can do block two after acquiring the lock. So you acquire the lock and you can do one and two in parallel. And what does the release say? Release says that once you release, the things after that can start executing but the things before it should have finished.

So we have to finish block two only then we can do the release but along with that release set of statements, we can parallelly start the block number three. Okay. So one and acquire can happen in parallel, release and block three can happen in parallel, and in middle we have the block number two. So we have established little more parallelism, so the green one is more sequential, the orange part has slightly more parallelism and it is giving more performance benefit. So the acquire is the read to gain access to the sync variable, we can use locks or while flag equal to zero.

So these examples we have seen. Release is to grant permission to another process like unlock or making the flag one to one and so on. So the acquire essentially delays access to set of statements after the acquire, okay, and it is not related to the accesses above the acquire. So I can say acquire and block one can happen in parallel, but only after acquire finishes, I will start the block after the acquire. That is, my block two can start only after this parallel composition is completed. The purpose of release, release grants access to others and it is not related to accesses that follow.

So release says that, you first finish the code segment then I release that lock. After that we do block three but the release and block three can happen together. Because I am going to finish my block two, only then start the release. So I can do block one, so if you look at this, we can do block one and acquire in parallel . Later we can only do the block two and once block two has finished and block one also has finished ,you can see these arrows, block two and block one, because block one is happening in parallel to the acquire and assuming that block one has finished, block two has finished , we can start the release and also the block number three. So release consistency gives us little more freedom. So what are the sufficient conditions here? So before issuing an operation labeled release, the process has to wait until all the previous operations have finished in the program order, right. So finish everything before the release and only then do the release.

Operations that follow and acquire are not issued until the acquire is completed. So you first acquire, then you do operations after the acquire, for release you finish everything before the release and only then start operations after the release. Okay. So we are not going to discuss the implementations of these models. These are given as examples. Okay. So that was the weak ordering and the release consistency. The other three models are commercial implementations of the same idea, so they are going to use the extensions of WO or RC to implement the real systems.

So the alpha, RMO and the power PC are the other three models. Okay. So for these three commercial models which are going to relax all orders, we need to discuss what are the different types of sync operations. So the overall idea is the same, the programmer has to insert the sync operations and between them we can have the reorderings, but across the sync operations, we cannot reorder. Okay. So each of these implementations come with their own implementations of the safety nets. So we will discuss the different safety nets across all these three models. Okay.

So for the Alpha, there are two kinds of safety nets in the kind of a fence instruction, the MB and the WMB. MB is a short form for memory barrier and memory barrier means ,it waits for all previous memory accesses, reads as well as writes. So MB waits for all accesses. WMB is a write memory barrier, so it says that all the previous writes should finish before you can do newer writes, but it does not put any restriction on the read. So if I have a read which is issued after the WMB, it can still be bypassed. You can still reorder the reads. But the writes before the WMB have to finish in that particular order. So that is the Alpha model so these are the two fence instructions available.

In the RMO model, the fence instruction or the synchronization instruction is the MEMBAR. It comes in four flavors because memory barrier is very generic. You would need memory barrier for read-read, read-write, write-read and write-write. Okay. So we can use any of these four flavors depending on the program. Okay. And Alpha and RMO do not require safety net for write atomicity but PowerPC requires, because it relaxes that order. So it has a single sync instruction equivalent to the memory barrier. The writes are not atomic here because it allows a write to be seen early by another processor's read.

So how do I solve this problem? For the program order maintenanc,e I have the sync instruction. For the write atomicity, as we did earlier, I am going to use a RMW instruction, so that the writes will be guaranteed to be seen atomically before I can move forward. So if you want to guarantee write atomicity for certain data variables, you have to replace them with RMW. If you want to stop reordering, you have to use sync. If you want to maintain write atomicity, you have to use the RMW. So that was an overview of these five models. We are not going to discuss detailed implementations of these five. But overall inserting correct sync instruction manages the task. Now what about portability? So are these models portable across each other? Can I say that I have five models, earlier I had two, three more models and I write a program on one system and I want to now run that program on another system.

So what is this portability? So a program which I wrote in TSO for example. So what was TSO? TSO was relaxing W followed by R and some write atomicity. Now that was a relaxed model but still less relaxed and this model, sorry a program written for TSO system, if I run that program on an RMO system which is the relaxed model, will it work? Okay, so you have to think what will happen. For the TSO model to maintain certain things, the programmer has inserted appropriate sync instructions RMW instructions. So maybe a few of them. But in the RMO model which is more relaxed than TSO, you would need more such sync and RMW instructions than you wrote in the program for TSO. So when you port a program from TSO to RMO, you will have to insert more such special operations. But reverse is oka,y because if your program was running on an RMO it will easily run on the TSO.

So do I have easier methods of doing this? So we need higher level interfaces for the programmer to manage this. So what is the programmers interface? So to make the task easy, we say that let the programmer be comfortable with managing the weak ordering and the release consistency. So not the five models right now or not the three models of earlier, we will say the programmer only concentrates on the weak ordering and the release consistency, okay. So once I have this, what should the programmer do? Only insert the sync instructions at the appropriate positions and assume that reorderings are allowed between the blocks of the sync instructions, okay. So once I have these using flags and explicitly labeled positions, the programmers task is done.

What should happen next? When this program is compiled, that particular system should now map these sync instructions in the programming language to an appropriate instruction mapping in the hardware. So that is the job of the compiler and the hardware to give support for implementing these sync instructions. Okay. So once the program has a point to point synchronization, it is the job of the compiler and the runtime library to translate all these sync operations to appropriate order preserving instructions, that is MEMBAR or fences. Depending on the system, you would have variety of instructions available. So correct instruction has to be replaced. So at higher level, we will simply say while flag is equal to zero and the compiler identifies that this particular while loop is an acquire, is it a release, is it a sync operation and then appropriately translates it to a correct memory barrier type of an instruction.

And definitely because it is a WO or an RC model, we are going to allow reordering between the sync operations and this is very good because it is very useful for out of order executing processors and all possible compiler optimizations. So that is the programmer's interface. So what is the programmer's contract? The programmer has to include point to point event synchronization using flags or give explicit labels. You say that ,these are the sync instructions, that is the contract with the programmer. What is the system's contract? System guarantees that, if the programmer has inserted synchronization instructions, it will appropriately translate it to hardware instructions, so that they are in the correct manner.

When both these things are done, we will guarantee sequential consistency. So SC model which we started before, so we can still guarantee sequential consistency, if we follow this procedure of inserting appropriate sync instructions and the hardware or the system translating it appropriately. So even if the reordering of operations between the sync is done in any way, so the hardware or the compiler, hardware in the sense the processor which does out of order execution, it can reorder instructions, compiler can reorder instructions, both of them as they desire, they can do this, still guaranteeing sequential consistency. Of course we need to maintain the data and control dependencies within the program. It will allow the processor to perform reorderings, it will allow the compiler to perform reorderings and thus with the support from the system, we can have portability.

So SC executions, that is sequentially consistent executions are guaranteed even with weaker models. So we were debating that sequential consistency is so strict. I want sequential consistency, but it put restrictions on my compiler optimizations and my hardware optimizations. So what can I do? So with this method which we have discussed, we can have sequentially consistent executions even on weaker memory models. Okay. So we allow lots of reorderings which helps me in performance and at the same time my correctness is also guaranteed.

So consistency model which is presented at the programmer's interface, what should it be? It should be as weak as possible. Because for portability if you have a weaker model, if you have a program which runs on a weaker model, it will always run on a more slightly stronger model. So consistency model should be as weak as the hardware interface. So if the hardware is weaker, your programmer's interface has to be a weak model, but they need not be same. Because the system guarantees that the instructions will be correctly translated depending on what the hardware is implementing. So if the program has labeled synchronized events, it is called a synchronized program. This will be correctly translated by the hardware to appropriate safety nets which are available for that particular consistency model. And where should we insert these sync events? Well, that is the job of the programmer and the programmer knows that which operations need to be synchronized, so that they can insert those instructions or give appropriate labels because the programmers are writing the program and they know which parts need to be synchronized. Okay.

So overall we permit a program or we write a program which can run on the weakest consistency model, the system guarantees to appropriately translate the sync instructions to the correct hardware instructions that is memory barrier, fence, STBAR, RMW and so on. And this way, given any memory consistency model, we will be able to run programs which will be still satisfying sequential consistency. Because it is not always required that the complete program should run in that particular order. We can reorder them, only thing is certain positions in the program need to maintain the order and the programmer can do this. Definitely there are also methods to find out these sync operations automatically but that is out of scope of this subject. But overall as long as you have sync instructions implemented, your program can allow reorderings yet establish sequential consistency. Okay.

So with this we finish this topic on memory consistency. Thank you so much.