Parallel Computer Architecture

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

Week - 10
Lecture - 55

Lec 55: Implications of Sequential Consistency

Hello everyone. We are doing module 7 on memory consistency. This is lecture number 2, where we are going to look at implications of sequential consistency. So, before we begin we need to understand ,why is it important to maintain the program order. We said for sequential consistency we should maintain the program order and the write atomicity. These were the two requirements.

The first requirement was program order. And a program order means the order in which the instructions are written by the programmer. So there could be reads and writes apart from arithmetic instructions. So every read and write has an order in the program and we need to maintain that order.

So what are the different orders? We have a write followed by a read, a write followed by a write, a read followed by a read and a read followed by a write. So it is important to maintain these orders. So let us look at the importance of maintaining the write followed by the read order. We will do them one by one. So when I want to maintain sequential consistency and I need to maintain the write followed by the read order, this might not be always easy in every system.

If I take an example here where I do not have a cache, for example, okay. So we have a multiprocessor system and it does not have a cache. We need to maintain the program order. So I need to maintain the write followed by the read program order. The architecture is no cache, it is a bus based system, but I have write buffers, okay.

So if I have no cache, what would happen? The reads and writes would all go to the main memory through a shared bus. So P1, P2 go onto the bus to the memory, but whenever a write happens, it is using a write buffer. So this is the write buffer in which it will temporarily store the write because when a write takes place, its value is not needed by the processor and hence it can put it into the write buffer and then go past this write. So this write, it puts the value in the write buffer and then continues to the read instruction. So this program order will get violated if I have write buffers.

So the example we are considering is here. So this is the Dekker's algorithm for critical section, where you set the flag 1 equal to 1 and then check if flag 2 is 1 and then enter the critical section. Similarly, P2 is using the opposite flags. So this is the write followed by the read. This is the write followed by the read, okay, alright.

Now in the presence of write buffer and the program which we are discussing, it is there on the right side of the figure. Let us see the different actions which can take place. Suppose the write of flag takes place. So this happens first, but then this value goes and sits here, in the write buffer. Similarly, this value also goes and sits in the write buffer.

So although they executed in the program order, but the flag values have not reached the memory, they are sitting in the write buffers. Subsequently, it is going to issue this read instruction. This read instruction of flag equal to 2 goes on to the bus, goes to the memory and reads the old value of flag equal to 0, this one. Okay, so it goes to the memory, reads old value of flag 0. Next is this one comes, it goes to the main memory, right.

It goes to the main memory, reads the old value of flag equal to 0 and when the flag is 0, both of them enter the critical section. So both of them will see the value of flag 0 and enter the critical section. Because after some time only these two write buffers will get flushed and update the memory. So in the presence of write buffer, even if P1 and P2 executed the program in program order, they were not able to satisfy sequential consistency. Okay. They violated our mutual exclusion problem because both processes went into the critical section. We can say that this write buffer optimization was safe in a uniprocessor system because there you will have a single write buffer and when process P1. Now imagine both write buffers merged into a single one, we have a uniprocessor system and whenever a read for that variable happens, it definitely checks the write buffer.

If the write buffer has a value, picks up the value from there and in case I had a merged write buffer, both of them together, so when P1 went for reading the flag, it would get or when P2 goes for reading the flag 1, it will get the latest value from the write buffer. Okay. Not like it went to the main memory and got the old value. So this optimization is safe in a uniprocessor. But we want to make sure that it should also be okay in a sequential consistent system, but here it is not. Right. So such reorderings is violating the sequential consistency requirement in a multiprocessor environment. So even if you wrote the program correctly, this particular architecture is not executing it in the desired manner.

Okay, let us see one more example which will show the importance of maintaining a write after write ordering. So this program P1 does write and the write. So this is the write followed by a write. So two writes are happening. P1 writes data 2000, head writes data 1 and P2, P2 is doing simple reads.

It is doing read to the two values. But if you see the programmer's intuition here, you would say that well, I think there is a connection between these two and when head becomes 1 only then data will be read or printed and the programmer says, I want the value to be 2000 because head becomes 1 only after data becomes 2000. Okay, so that is the intuition of the programmer and meaning of this program. Okay. Now coming back to our current setup, if I would say that I have a multiprocessor system with multiple memory modules, data items could be scattered across them and suppose I assume that data and head, these are my two variables, they both sit in different memory modules.

Okay, they get allocated to different memory modules. Now we will see how sequential consistency can get violated in this scenario. The scenario is about overlapped writes. So both writes happen in the program order, right. So they happen in the program order, but the data equal to 2000 was sitting in a memory module which was little far, right.

So this was on a slow path. So data equal to 2000 has not yet reached its memory. In the meanwhile, head equal to 1 reaches its memory. P2 starts reading and because head has already reached the memory and become 1, this while loop is going to break and it will go and read the data. So when P2 goes and reads the data, it sees the value of data equal to 0 and it prints the value of data 0.

Okay, so this happened because I had different memory modules, each accessible at different latencies or hop distances. Okay. So sequential consistency said that I want the value of data to be 2000, but the network delays cause the head to finish before the data gets written and hence P2 reads the new head but the old data. Now how do you solve this problem? So for this you would say that P1 should have actually waited for data equal to 2000 to finish. How would it know that data equal to 2000 has completed? Either that it waits for the memory to send an acknowledgement.

So when data equal to 2000 finally reaches, an acknowledgement goes back to P1 and only then P1 issues the next head equal to 1 instruction. Okay. So if we do this, then our sequential consistency will guarantee. But in normal cases, if we do not have this extra acknowledgement and we do not wait between the two writes, then we cannot guarantee SC. The other two orders are read-write and read-after-read.

Okay. So here we have the read-after-read. Now you would say that reading two different variables, we can definitely reorder them because reads to different variables can be easily overlapped. But P1 does them in the program order. So P1 is doing program order, but P2 says I have just two reads, can I overlap them? Because this is very much possible in a system which are using lockup-free caches, so they use speculative execution or dynamic scheduling. So if I have, you have these processor optimizations, then the reads will get overlapped.

So here when the reads get overlapped, the read of data, because they overlap, it so happens that the read of data reaches the memory first and it goes and reads the value equal to 0, it comes with this value and then the head goes. But unlucky for us that the head goes little later only after this head is updated. Okay. So we have already cached or read the value of data equal to 0 because that read has completed. The read of head finishes only after the write of this head, hence P2 will get head equal to 1 and data equal to 0. Okay. So at time instant T1, the read of data finishes at P2, at time T2 and T3, P1 finishes the two writes and the last action is reading the head.

So when head is read, it gets the newest value of head equal to 1. Okay. So the same example here, in the absence of cache and in the presence of cache, in both the cases, we are going to violate the sequential consistency. So in the absence of cache, the reads are overlapped, P2 reads the data value of 0, initially. So P2 finishes the read, then P1 does the two writes and then P2 reads head equal to 1, hence it gets the new head and the old data in the absence of cache. Well, in case we have a cache available and if I assume that data is in the cache of P2. So because data is in the cache of P2, when P1 does a write of this, it is going to send an invalidation or an update in the system which will eventually reach here. But we do not know how fast because coherence does not say about this. It will eventually reach there, but before it reached P2 had reordered the reads and it reads the old value of data from its cache. Okay.

So it reads the cache data equal to 0. After this, head and data get written to the memory, data in P2's cache is yet to receive the update or the invalidation, then P2 reads the new head and definitely it got the old data. Okay, so what is the solution for this, that P1 should wait for, so P1 has to wait for all these invalidations and updates to finish before it can write to the head. So that is the write followed by the write order has to be maintained. When we say maintain this order means, you have to finish the impact of this write across the complete system before you start doing the next write.

Okay, so what are the implications of my sequential consistency model? Well, first thing is it defines the constraints on the order in which the memory operation should appear to get executed. The benefit of that is the programmer can reason about the outcome of the program, software has to know what is the memory consistency model and when we discuss memory consistency, we are discussing it from the programmer's perspective because the interface between the programmer and the compiler, compiler and the OS, so all these interfaces here, here, programmer and the compiler. So all these have to follow the same consistency model. So between every pair the programmer understands the consistency model, the compiler understands the same one. So there is a contract between every layer, every pair of layers that, this is the consistency model we are going to maintain.

Because, suppose the hardware which is the processor, it maintains all the orders, it does not reorder. Okay, we have a in-order processor. So processor may maintain all the order. But if the compiler has already reordered then the programmer cannot reason about the outcome because if programmer thinks that well, hardware is in order and in the middle the compiler has already reordered. So the final outcome cannot be concluded by the programmer. So therefore any consistency model has implications on the programming language.

When I say programmer it is the programming language you are writing the programming, the compiler as well as the hardware which is the processor and the memory subsystem. So sequential consistency will give the programmer an intuitive semantics of the program order and the interleavings, so that all the conditions are nicely satisfied and we can derive the outcome of our programs. So when I want to do this, I would put restrictions on what ordering to maintain. We will say that, we normally say that maintain the write followed by read order, maintain the write-write order, maintain the read-write order and so on. So if we put so many restrictions, on top of it we put the restrictions on write atomicity.

So what are the implications of this? The implications is that, we have lots of restrictions on the compiler optimizations. If the compiler cannot optimize, the performance will be hampered. Okay. Then the fewer reorderings we allow. The easier it is for the programmer to reason because this programmer knows that this is the order it is going to follow, so programmer can derive the outcome. So it is easier for the programmer but the performance is hampered. So the consistency models job is that it wants to strike a balance between trying to reorder as much as possible but at the same time still maintain the simplicity of the program. And it also has to be portable.

So when I shift a program from one system to another, am I preserving the semantics across the two different systems? SC is going to put restrictions on what it can reorder so that it does not violate the sequential consistency requirements. The fewer the reordering is better for the programmer but worse for the performance. So therefore we want to strike a balance between the programming complexity and the performance. And it has to be portable, that is implementable on many platforms while still preserving the semantics. So what are the challenges to maintain sequential consistency? We will say that well, if you want SC, running the program order defined by the source code. So program order is the source code not the compiled code.

Compiler should not change the order of memory operations that it gives to the hardware otherwise SC from the programmers perspective may be compromised because programmer think this is the order of execution, whereas compiler change the order hardware saw the new order and you don't know what the outcome would be. But unfortunately compilers are going to optimize. We can't say that compiler should not optimize because for performance we need to optimize. It is going to reorder. Even if I use explicit uniprocessor compilers, well, still it is going to do some optimization.

To maintain sequential consistency while allowing the programmer optimization becomes a big challenge. So the drawback of maintaining SC is that compilers, we want to put restrictions on the compiler to optimize and so on. So overall our performance is going to be hampered, if I am going to be very strict in maintaining the sequential consistency. Compilers will be stopped from reordering, code motion, right, common subexpression elimination, right. So it is going to eliminate certain code, it is going to optimize your code, it is going to move your code around, to make it more faster, it is going to use pipelining, some register allocation. All these optimizations will have to disallow, if I want to maintain strict SC behavior. Okay.

So if sequential consistency has to be met then the processor has to wait for an access to complete across the complete system before issuing the next one and this is going to be very slow. It is going to add lots of latency in the system and all this latency will be counted as a processor's stall time implying reduced performance. Okay. So if we stop optimizations and if we follow the strict sufficient conditions, our performance is going to be hampered. So that is the drawback of sequential consistency. At the same time, we want the advantage of sequential consistency to be able to reason correctly about our programs.

So this drawback of stall time, how can I hide that latency? You can say there are a variety of methods to hide latency. We can do prefetching of instructions and data or we can say that let me preserve sequential consistency but not all the conditions. Okay. We can say that certain conditions can be relaxed. For example, we can say a compiler, allow the compiler to reorder as long as it can guarantee SC, but you would feel that if the compiler reorders, how do you guarantee SC will happen. So these algorithms are going to be very complex and expensive. Because in the previous example when we were discussing you understood that to maintain SC, you need to see that certain reorderings are allowed but certain reorderings are not allowed.

If you can recollect we had this one. Right. So we said these two reorderings were allowed. But in another example the swapping of instructions was not allowed. So the compiler has to infer all of this before it can give you an optimized code which is sequentially consistent, yet reordered. Okay. So these are going to be expensive. At the hardware level also we are going to issue instructions out of order because if there is a cache miss or if the memory bank is further apart from the requesting processor, it is going to take a lot of network latency and therefore it would be tempted to reorder instructions which would not be in the program order. So you want to do all these optimizations to hide latency.

Processors which are dynamically scheduled out of order so even they do lots of optimizations. We do speculative executions. We need support for rollback in case of mispredictions and so on. So at the hardware level, memory operations are issued and executed out of program order in the interest of performance. Right. So all these techniques work for processors but they do not help the compilers in doing the reorderings.

So we want the reordering. We want the hardware optimizations. At the same time, we want sequential consistency. Okay. So I hope you got the feel that what we are trying to achieve. So we want a sequentially consistent behavior which dictates the compiler don't reorder, which dictates the hardware don't do out of order execution, don't do dynamic scheduling. Okay. But at the same time, we want the compiler optimizations, we want out of order execution. Okay.

So these are conflicting requirements. So how do I solve this? I want SC, I want performance, I want optimizations. Okay. So what is the solution for my sequential consistency problem? So a completely different way to overcome this dilemma is, change the consistency model itself. Okay. So not to guarantee the ordering constraints, but still retain the semantics, which are intuitive enough to be useful. Relax the orders. Okay. But these relaxed orders will allow the compiler to reorder accesses and then present them to the hardware.

So I would say that we come up with a new consistency model which will allow me relaxing certain orders. At the hardware level, I will also allow multiple memory references to be outstanding. That is, we don't do them one by one, one after the other, but we do them in an overlapped fashion. That is, if there is a cache miss, let that cache miss get satisfied. In the meanwhile, I will do some other instructions, other memory accesses.

So we can permit memory references from the same processor, multiple of those to happen and become visible out of program order. So this will allow me to hide or overlap the memory access latency. Okay. That's my target. To solve the problems of sequential consistency versus performance, I am going to think of newer consistency models. Okay. And one would say well, SC was good enough and if you come up with new models, what's the guarantee they will work. Okay.

But we can say that sequential consistency is overly conservative. We are too restrictive, we are too demanding upon the system. Because most of the time the programs which you write, you are not so sure or so, you don't require so much ordering to be followed. You are okay with some reorderings. Okay. So if we do not want to be so strict about ordering every time, can we relax the orders whenever possible. Okay.

So that's the objective of the newer consistency models. But remember the foundation is sequential consistency. Any model you do, it should be sequentially consistent at the end of execution. Okay. So in the next lecture we are going to look at more relaxed consistency models. With this, we stop this lecture. Thank you so much.