Parallel Computer Architecture
Prof. Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Week - 04
Lecture - 20

Lec 20: Concept of Serialisation

Hello everyone, we are doing module 3 on Cache coherence. This is lecture 2. In this we are going to understand the concept of serialization. Right. So what is cache coherence? Cache coherence occurs when the data across private caches among the processors are different for the shared data items. And how do I solve this cache coherence problem? One is to disallow caching of shared items, which is actually not practical, but that is a very easy solution. Don't allow processors to cache variables which are shared. So.

For all the shared variables, the processors will have to go to the main memory, only change the main memory location, hence there won't be any inconsistencies. Okay. So this is one solution. Other solution is to invoke the operating system for such shared data items. However we want to handle this inside the hardware. Right.

And if you want to do this in hardware, how do I solve this problem? One is that when a cache changes a data item, this change has to be informed to others. Right. So informed to others meaning either tell others to remove their copies or give the new updated value to the other caches. So cache coherence to be addressed in hardware involves eliminating other copies, that is delete the other copies or update the values of the other copies with the latest value. Okay. So when I say update with a new value, this essentially means that whenever we do a read to a location, I am going to get the latest value of data. This was also seen in the model of the memory that whenever you go to read into the memory you are always going to get the latest item.

Now how do you decide latest value? Right. So two processors might write to the same location at the same instant. Same instant, both of them are doing a write back, so both these values are going to the same location, almost close to each other. And there may be another processor which wants to read this value. So if suppose x equal to 5 and x equal to 7, these are the two values being updated and there is another processor P3 which wants to do a read on x. So it wants to do a read for x.

So what value should you get? Again whenever x became 5, we were supposed to

inform everybody that x was equal to 5, but we were not able to reach P3 and by the time P3 has already issued its read of x. Right. These are many actions which are happening and how do I understand the correct behavior? Suppose two processors are changing the values and one of them made x equal to 5, this change of x equal to 5 was not able to reach P3 and by that time P3 started reading the value of x. So that is there was no time to propagate the information and P3 started reading x. So in this scenario, is P3 getting the most latest value? Is it going to get the last value which was updated? Was it equal to 5? Was it 7 or was it an original value which is already there in the memory? So how do I understand this concept? So essentially when I say last value, it is actually in the program order. So program order is the order or the sequence of instructions which execute in a program essentially related to only loads and stores.

So in a sequential program, it is a single order. So a sequence of instructions is there in a single program running on a single processor. But if I have a parallel program, I have multiple threads and each thread has its own program order. So how do I understand the concept of a latest value when there are multiple program orders running in a parallel system? Okay. So we need to understand the concept of a latest value. Okay. So for this, we will look at some formalisms of understanding the program order and the concept of serialization in sequential and a parallel case.

So sequential is a uniprocessor case. Okay. So in a uniprocessor, we can say memory operations occur when I do a read or a write and these are atomic. So when I go for a reading, the read completes. When we go for writing the data, the write completes. In certain processors, there are complex instructions.

What do these instructions do? They might have an internal read followed by a write. So if there is a read, modify, write type of an instruction, we also assume that these instructions happen atomically. That is, in a complex instruction, multiple read or write co-instructions will be done together. Okay. So that is how memory operations are going to happen. Then we say that the memory operation issues when it leaves the processor.

So it has left the processor and we say that the processor has issued a memory operation. But just issuing does not finish the operation because it has to go through a long path, through the cache, through the write buffer, finally to the memory module and if it is on a global interconnect through the interconnect. So this is going to take long time to complete. Okay. So when it is going to take long time to complete, the only way a processor would know that its memory operation is completed, that is whatever it did has finished by issuing reads. So I will explain this.

So if a processor performs a write operation, how does the processor know that the write

has completed? It will know this only by issuing reads to the same location. Okay, so I will quickly give a small example. There is a person, so this person flies a paper aircraft, right, so small paper airplane. So he throws this and this is supposed to be landing into this box and this box is very far. So this person cannot see this box.

So sending this here is like issue the memory operation. So this person issues the memory operation of writing this airplane inside this. So this airplane should come and sit here. Would the person know that once he throws this small paper plane, has the job finished? We do not know. So how would the person know that the plane landed into the box? By going and reading into the box.

So this person has to come here, issue a read and then check whether this plane is landed into this box. Right? So we will say that the processor knows that the write operation is performed by sending a subsequent read operation to the same location. Because without reading that location, we cannot confirm that the value we wrote has been written. Okay, so this will illustrate whether the write is performed. So here I have processor P1 which is issuing u equal to 5, another processor P2 which issues u equal to 8.

Now this P1 issued u equal to 5, so it goes and sits into the memory. How would P1 know that u equal to 5 has reached the memory? By reading that location. So P1 issues a read to the memory and then when the read returns, it will check oh yes, 5 has come back, so my u equal to 5 had reached the memory. Next, if P2 writes a u equal to 8 inside that and so P2 can also cross check this, but now if P1 reads this, P1 will get the latest value. So we can say a write is performed only by issuing subsequent reads to the same location.

So that was write performed. How will I know that a read has been performed? Now what does this mean read has been performed? A processor reads a memory value and this value which we read just now should not change by other effects that is other writes to the same location. See, see here P1 had a value of 5, it wrote to the memory and then to check whether the write has finished, it did a read on u. So it got u equal to 5. So according to this, the write was complete and the read of u, is this read of u finished? Let us check.

Now when P2 sends a u equal to 8, this u has changed to 8 in the memory, but did it change with respect to the processor? No it did not. So when the u equal to 5 stays with the processor, no matter what changes happen to u inside the memory, this is when we say that the read is complete. Okay. So a read is complete when the future writes to the memory location will not change my read value. So a processor completes a read

operation meaning that subsequent writes to that location cannot affect the value which we have read. Now we need to understand the term subsequent.

What does subsequent mean? It is well defined in a sequential process because it is a program order. So this is a single threaded process that is the instruction number 1, 2, 3, 4, these are the instructions which are doing some operations, read variable a, write 5 to variable b, write a equal to 8 and so on. So this sequence is called the program order. So in a sequential process, the program order is well defined by the instructions of the program and normally these are machine level instructions rather than high level because if you write a high level language program, the compiler might translate it in a slightly different manner. Hence, we will define program order at the translation achieved after the machine level translation.

So we can replace this small sentence. Okay, right. So because the compiler might change the order of some instructions, I am defining the program order with respect to the machine level translation. So that is about the uniprocessor. Because I had a single process, this is the sequence of instructions we are going through. What happens in a parallel processor? Now here what does subsequent mean? If you see in this one, I can say this comes after this, right? So write of a happens after write of b or write of a equal to 8 happens after read of a because here these instructions are ordered.

First instruction 1 happens, then 2 happens, then 3 happens. So this is the order of execution. But what is the order of execution when I go to a parallel case? How can I define subsequent in this case? So what is subsequent here? So because here we do not know what is the final program order. In a sequential it is easy, we just have a single thread. But here we have multiple search orders.

So what is the final program order? So to understand this concept, let us see an analogy. So this is a queue of people. This is queue number 1 and this is queue number 2. So there are two parallel queues and both these queues want to go through the single gate or through a single window. Right. So we want to access a common resource and come on the other side of that.

So what is the order in which you will form a queue on the outside? Will you first finish queue 1 and then you will start queue 2 or will you start first queue 2 and then queue 1 or some interleaving? Right. So essentially the outcome is some interleaving of these. Right? Because it cannot be queue 1 followed by queue 2, it would be say this person comes first, then this is the second one, probably third to exit, fourth to exit, then fifth to exit and so on. So we have an arbitrary interleaving of these people going through the door. So we have actually two program orders. This is one program order, that is one

queue.

This is the second queue and this one is the output program order. And can you say or predict what the output program order is? No, because it at runtime this could be any behavior. But what we can guarantee here is that if you just observe queue 1, the order in which people were standing in the queue 1, the same order you will see in the output queue. That is the person in green shirt will be before the person in the yellow shirt. It can never be that this person goes ahead of that person. Okay.

So only this much can be guaranteed, but the overall program order is not possible to be derived or rather it could be arbitrary. Okay. So similar to this example that this common door was imposing an output order in which these queues will get merged, the memory is going to impose an order in a parallel program. Right. So here the multi-cores and will generate their own program orders. There is a single memory and let us say that there are no caches to not to complicate matters further. Okay. So we have a single memory and we have multiple cores.

Then the reads rights are directly performed with respect to the memory because there is no cache. So all the read rights will go to the memory and memory is going to impose a serial order. If you can now draw the analogy, this gate was imposing the serial order. So this was going to decide the serialization. Right. So this was going to serialize the two queues into one.

So similarly the memory is going to serialize the accesses for these multiple cores. And among the individual processes, the program order will get maintained. As we discussed here, the people inside the queue will be following the same order they were standing earlier. So similarly the individual processes will follow their own program order in the output serial order. Okay.

So a small illustration. Here the purple table is showing one program. So this is some program P1 that is the program order 1, 2, 3, 4, 5. This is another program P2 both running on different processors. So they are running on two different physical processors.

They do not have a cache. Now these are going to do read write to the common memory. So this is the common memory to which they will send their accesses. So first is this one will go out. So it comes here 2, 3. Right. So I am just showing a small piece of the execution.

So read instruction 1 of the purple program comes, instruction 2, instruction 3. So this

is how the memory operations are issued. So processor has issued the memory operations. Same thing happens with the yellow program.

Here again instruction 1, 2, 3 are issued. Right. So have they reached the memory, have they finished the operation and in which order will they finish the work. So any guesses? Of course, similar to the queue merging, we are going to have a merge of all these different operations but the order is not known. Okay. So one possible serial order is that we do two of these purples, then we do one orange, then we do one purple, then two orange and so on. So right. This is one possible merge or one possible serialization of these two different program orders. And definitely this is, there is nothing wrong in this.

This is a correct order and we also have to just cross check that the instructions belonging to one particular process are they in the correct program order of the individual ones. That is after merging also, they follow the correct order. We have 1, then the 2 comes and then the 3 comes. So within the output serial order, within the serial order, there is a program order maintained for every individual process.

So that is one possible serial order. I can have another one where the yellow process gets more priority and it quickly pushes few more instructions and then the purple gets moved out. Right. So again this all depends on when this process issues the instruction, how long does it take to travel to the memory. Right. So there are many other factors which will affect this order. Many popular analogies merging of cars in a small road. So you have two queues coming here and the order in which they will merge is not known.

Only thing is one car in a particular queue cannot overtake the same, cannot overtake a car in its own queue. That is the program order among the queues have to be maintained. Right. So what happens in practice? We have seen theoretical understanding of this. So we do not, do we construct the serial order? We need not. Because the serial order is not guaranteed or every process has its own cache and there could be cache hit or miss or network delay and so on. So this serial order which happens each time you run the same set of processes will be always different.

So in practice I cannot construct this serial order and I need not construct this serial order. Why? Because formally what we are going to require is that a multiprocessor system is coherent if the result of any execution of a program is such that each location, for each location it is possible to construct a serial order of the operations to that location and it is consistent with the results of execution. So this is one program order, this is one program order. If I had the program order of P1 completed first then P2 or if I do P2 then I do P1 or I do a mixture of say partial P1 then P2 then P1 then P2. Right.

So this is some hypothetical serial order. So this is one serial order, this is second and third. So we can construct any hypothetical serial order. If I construct such a serial order, then and I do a read write to any location, so it should be consistent with the result. And when will that happen? Provided for every program we follow the program order that is P1 does not reshuffle its instruction, P1 follows everything in its own program order and the value of every read operation is going to give me the latest value written in that particular serial order. Okay.

So we will understand this concept further later. That is what is the last value which I am going to read because the serial mixing of the processes will be different. Hence in one serial order the value could be 5, in another it could be 7. But as long as it is following a hypothetical serial order and as long as every process is following its own program order in this hypothetical serial order we should be okay. Okay, so we will take an example.

Again the same two processes. Here for the first type of serial order which we had constructed. So this was the serial order which we constructed and in this if I ask you what is the value of a at the end of this snapshot, this is not the complete execution, at the end of this snapshot what is the value of a? Okay, so what will we do? We will only try to see the changes in the values of a because I have just removed all the other instructions. The only instructions which change a are kept here. We also have to cross check that individual processes are in their program order that is the purple is following its own instruction sequence, the yellow is following its own sequence and in this if you see the value of a which you will get is equal to 9. Okay.

So this is for this particular serial order. Right. So if I follow this serial order a will be equal to 9 because this is the latest instruction. So this is the last value. If I go back to the definition here, if you see the multiprocessor system is coherent if the results of the program are such that for each location given any hypothetical serial order the result of that value is equal to the last write which has happened to that location and each individual processes following its own program order. So we managed that the purple is following its program order and in this the most recent value to a was equal to 9 and hence when I do a read of a I am going to get a 9. Okay.

I will take the other serial order. We had taken the second serial order in the previous instance. So this was the second serialization order. In this if you notice that the purple process gets scheduled little later and the a equal to 9 has already finished. So now here if I ask you what is value of a, you will say that yes it is equal to 8. Okay. So we will eliminate the others, see that the program order is there and then get the most recent

value of a equal to 8. Okay.

So what you have seen here is that depending on what type of serialization happens at the memory port, the latest value of a will change but as long as you are reading the latest value and as long as every process is following its own program order we are following a consistent system. Okay. So serial order of memory accesses that is memory is the point which decides the order of access. I hope now it is very clear to all of you. Memory is going to decide in which order the reads and writes will be affected. Again there is no fixed sequence among the processes that is they can be interleaved anyway right.

So arbitrary interleaving happens. The only guarantee is that the system is fair that is even if a process gets delayed eventually its access will reach the memory. So in the previous one suppose the yellow process never got a chance or its network was very slow, all the purple instructions finish and then the yellow instructions will start. So overall there is a fairness in the system no matter what is the delay the instructions will eventually get executed. Okay. And inside this the last or the subsequent what is this? It is that value in this hypothetical serial order. So that order could be anything and in this hypothetical serial order whichever is the most recent value will be returned to us. Okay.

So this is the concept of serialization which is very important to understand and solve for cache coherence. Thank you so much.