Computer Architecture

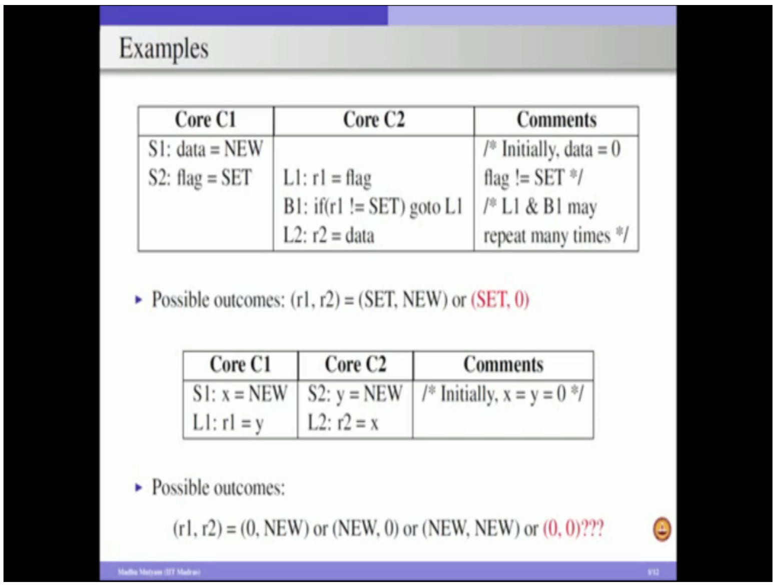
Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module – 09
Lecture – 32
Memory Consistency (Part 1)

So, when you are dealing with the multicore systems, multiple applications can run on multiple cores. And these applications can be a multithreaded applications or multi program applications. When applications are running on multiple cores they share the memory. So, once we have a shared memory system associated with our multicore systems we need to provide the correctness of the shared memory. And the correctness of shared memory is provided by using coherence as well as the consistency.

So, we already discussed the cache coherence mechanisms and protocol associated with cache coherence maintenance. And now in this module we are going to discuss the memory consistency. And before formally defining this memory consistency, now we are going to see a set of examples and that will motivate us to go for memory consistency models.

(Refer Slide Time: 01:06)



Examples

Core C1	Core C2	Comments
S1: data = NEW S2: flag = SET	L1: r1 = flag B1: if(r1 != SET) goto L1 L2: r2 = data	/* Initially, data = 0 flag != SET */ /* L1 & B1 may repeat many times */

- Possible outcomes: (r1, r2) = (SET, NEW) or (SET, 0)

Core C1	Core C2	Comments
S1: x = NEW L1: r1 = y	S2: y = NEW L2: r2 = x	/* Initially, x = y = 0 */

- Possible outcomes:

  (r1, r2) = (0, NEW) or (NEW, 0) or (NEW, NEW) or (0, 0)???

So, consider a 2 core system where in core 1 we are executing this piece of code where there is a store instruction which is storing value NEW into variable data. And another store instruction which is going to set the flag bit. And in core 2 we are executing this piece of code where we are actually loading the value of this flag into a register r1 and then there is a branch instruction which is actually checking whether r1 is not equal to set or not. And based on this condition outcome we either go back to this L1 executing this load instruction once again or otherwise we are going to execute second load instruction that is r2 equal to data.

So, given this piece of code, now what are the possible outcomes we can expect? So, as a programmer when we write this piece of code our expectations will be, if the load 2 is executed that indicates that this branch condition is false. So, the branch condition false indicates that r1 is equal to set, when r1 is equal to set indicates that the flag is equal to set. So, as a result when flag is equal to set here then automatically this first store instruction should have been completed.

So, that means the data should get a value NEW. So, in that scenario now r2 should get value NEW. Now, we will see what are the possible outcomes we will get? So, if the output is set and NEW for r1 and r2 then this is according to our expectations or this is according to programmers expectations, but some scenarios what is going to happen is we can get value set, 0 where r1 is going to have value set and r 2 is going to have a value 0, but this is actually against the programmers intuition.

When a programmer writes this piece of code he assumes that the store one is going to execute first then store 2 and once store 2 is executed so flag is equal to set. So, as a result this branch condition will be false and then it will execute this second load instruction and this second load instruction is going to get the value NEW, but so we are actually getting set, 0. So, that is nothing but, so this second store instruction is completed before the first store instruction and because this store is completed. So, the condition is false.

So, as result we can execute this load instruction and when we are executing this load instruction before this store instruction. So, we are going to get the value of 0 in r2 register because initially data is equal to 0. So, this is against to the programmer’s intuition. Now, again we will see another example, here again we have 2 cores, core 1 and core 2. And core 1 is executing a store instruction followed by a load instruction here we are storing the value NEW and in the load instruction we are reading the value from memory location y.

Similarly, core 2 is executing store instruction followed by a load instruction, but here store 2 is actually storing a value NEW into a memory location y and load instruction is going to read a value from memory location x to register r2. So, now what will be the possible outcomes from this piece of code. And initially our x and y values are 0. So, now as a programmer our expectation will be, so it can execute in this order either S1 L1 S2 L2 or S1 S2 L1 L2 or S1 S2 L2 L1 or S2 L2 S1 L1. So, we can have multiple combinations here. So, but our expected values will be either r1 r2 will have 0 NEW or we can have value NEW 0 or we can have value NEW NEW. So, this 0 NEW will get in scenario where we execute this r1 is equal to 0. So, that means. So, we have to execute this S1 L1 first before S2 and after that we can execute L2. So, effectively we can get this outcome when we execute the instructions in the order S1 followed by L1 followed by S2 followed by L2.

Similarly, if we execute the instructions in the order S2 followed by L2 followed by S1 followed by L1 then we are going to get the value NEW,0 because we are executing S2 first. So, we are storing NEW into memory location y and after that there is a load instruction, which is actually loading the value from memory location y to the register r1. So, as a result register r1 is now having the value NEW. And also in previous sequence because we are executing load 2 before store 1.

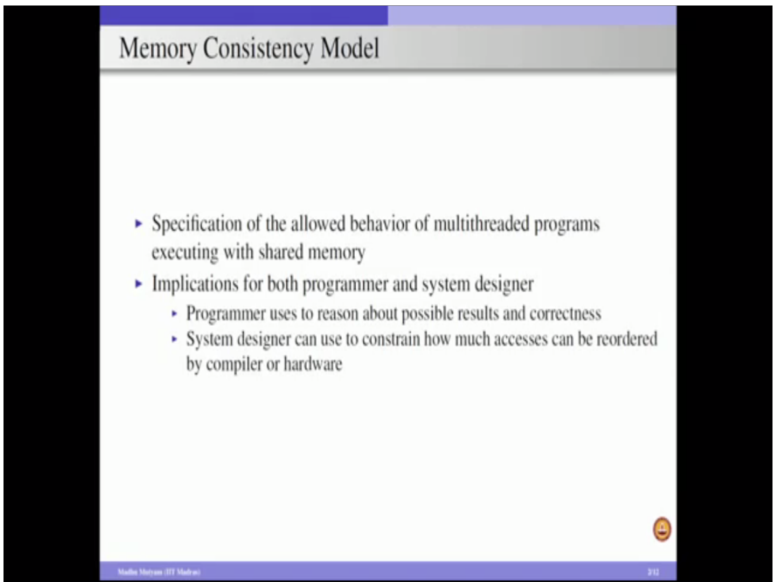
So, as a result we are going to read the old value from memory location x. So, as a result r2 is going to have the value 0. This is also valid sequence according to the programmer's expectations and now we will see the third outcome which is NEW and NEW. So, this can happen when we execute the instructions in this order - S1 S2 and after that L1 L2 or otherwise S2 S1 after that L1 L2 or L2 L1. So, we perform the store operations first and then we perform the load operations then we can get this value NEW and NEW.

So, as a result when we write this piece of code as a programmer our expectations will be we either get 0 NEW, NEW 0 or NEW NEW, but this piece of code can also give an unexpected outcome which is 0 0. So, that is. So, the load instructions may be executed before the store instructions and if you do so then automatically you are going to get the r1 equal to 0 and r2 equal to 0. And after that you are actually writing to the memory locations x and y. So, because our underlying hardware may execute instructions in a different way than what the programmer is expected.

As a programmer when we write this code we expect that our first instruction is going to execute first, then the second instruction and third instruction and so on, but our hardware and as well as the compiler can rearrange our instructions in the program to maximize the performance. We already discussed as part of superscalar processor design instructions can be reordered as long as they are not having any true dependence. So, once we are executing the instructions in an out of order fashion at the hardware level.

So, the programmer has no clue how the instructions are executed and as a result we will get some unexpected outcomes than the set of expected outcomes. So, as a result the entire correctness of the system will be at stake. So, as a result we need to have a memory consistency models which will give us. As a programmer it will give us some set of rules saying that you can expect these possible outcomes. And also at the same time it is going to provide set of rules to the hardware designers or the compilers saying that you cannot go beyond this particular limit for rearranging the instructions or applying any of the optimizations.

(Refer Slide Time: 09:25)

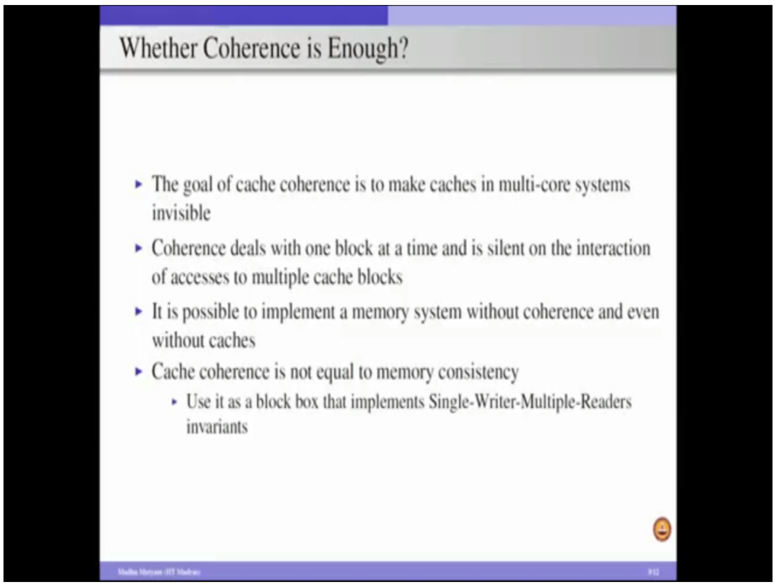


Memory Consistency Model

- Specification of the allowed behavior of multithreaded programs executing with shared memory
- Implications for both programmer and system designer
  - Programmer uses to reason about possible results and correctness
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware

In other words so, memory consistency model typically gives specifications of a load behavior of multithread programs executed on shared memory. So, it provides the information to both the programmers as well as the system designers. For programmers it will give set of indications so that the programmer can use these indications to reason about possible results and the correctness of the program when it is executed. Similarly, the system designer can use these indications to constraints how much accesses can be reordered by the compiler or the hardware so that we can achieve maximum performance improvement without losing the correctness issues, but any way we have in our systems the coherence support. When we have cache coherence support do we really require memory consistency on top of that? Yes indeed we require consistency even when we have the coherence support in our system.

(Refer Slide Time: 10:35)



Whether Coherence is Enough?

- The goal of cache coherence is to make caches in multi-core systems invisible
- Coherence deals with one block at a time and is silent on the interaction of accesses to multiple cache blocks
- It is possible to implement a memory system without coherence and even without caches
- Cache coherence is not equal to memory consistency
    - Use it as a block box that implements Single-Writer-Multiple-Readers invariants

So, the goal of cache coherence is to make the caches in multicore systems invisible because generally when we write a program, for the programmer it does not care whether there are 2 levels of caches, 3 levels of caches, private cache shared cache and so on. So, his idea is there is a processor and the memory and he writes a programs thinking that the entire code and the data will be there in the memory. And whenever the program requires then the data will be supplied to the processor from the memory. And, but once we have this private caches and shared caches or the multi levels of cache hierarchy to improve the overall performance, then we will have a cache coherence support.

So, that the cache coherence ensures that the data, the replicated data that is available with multiple private caches is in consistent state or in coherent state. So, as a result this cache coherence set up ensures that the caches in the multicore system are invisible, but at the same time it provides the correctness from the coherence point of view. And also we know that the cache coherence deals with only one block at a time and it is silent with the interaction of access of multiple cache blocks.

When there is a store instruction from one processor then if any other processor is going to read from the same memory location, then the cache coherence ensures that they are going to get the correct data. Either by using a invalidation based protocol or the update based protocol, but if processor p1 and processor p2 both are performing store and load operations on different cache blocks or to different memory locations, then cache coherence is not going to communicate this write to the processor which is going to read because effectively this write is to a different memory location and read is from a different memory location.

So, in other words cache coherence always deals with whatever the operations we are performing for a single memory block not for multiple memory blocks or multiple cache blocks, but whereas in the case of memory consistency we actually deal with the operation that are performed on the memory irrespective of whether it is for a single memory block or for a multiple memory blocks. And also another thing is we can design our multicore system without caches. We can have multiple cores and then the memory. So, there no cache once there is no cache there is no need of cache coherence protocols or cache coherence design, but even then we need the support of memory consistency.

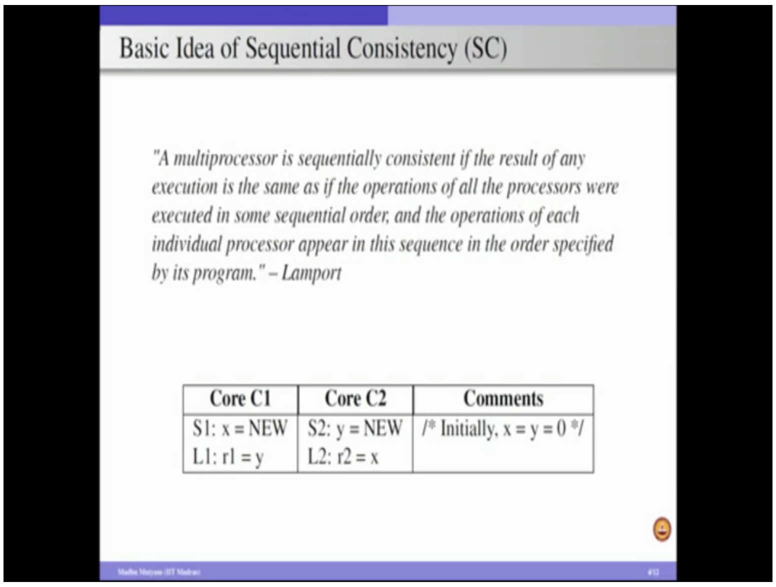
So, this actually says that memory consistency is required irrespective of whether we have a support of cache coherence or not in our systems, but because all the current multicore systems have multiple levels of caches with the private caches and the shared caches. So, as a result we have support of cache coherence. So, we can use this cache coherence mechanism for helping our memory consistency models. So, we can use this cache coherency as a block box that implements a single write and multiple reader invariants.

When I say single write and multiple reader invariant, so when we have a cache coherence system in our multicore processor, then when we perform multiple reads we are not going to change the content of the cache block or the memory block, but whereas, when we want to perform multiple writes at any point of time only one write will be applicable on a memory location or cache block. So, as a result all the writes to the same block will be serialized. So, as a result so the block content will be changed with respect to the latest write.

So, in other words when implement a cache coherence system. So, automatically when we perform one write. So, we will get the corresponding data updation in our memory block or the cache block and when we perform simultaneously multiple reads we are not going to change the state or the contents of the block. So, that is what the cached contents will not be changed when we perform multiple read operations, but the cache contents will be changed with respect to a single write at any point of time.

Also when we have multiple writes we want to perform on a cache block in a multicore system with the cache coherence support, the cache coherence system ensures that the writes will be serialized. At any time only one write will be happened to that particular cache block and it will update the contents of cache block. So, in summary we are going to use this cache coherence as a tool to aid our memory consistency models whatever we are going to design. So, in this module we are going to discuss sequential consistency model. And in the next module we are going to discuss total store order model as well as the relaxed consistency model.

(Refer Slide Time: 15:47)



Basic Idea of Sequential Consistency (SC)

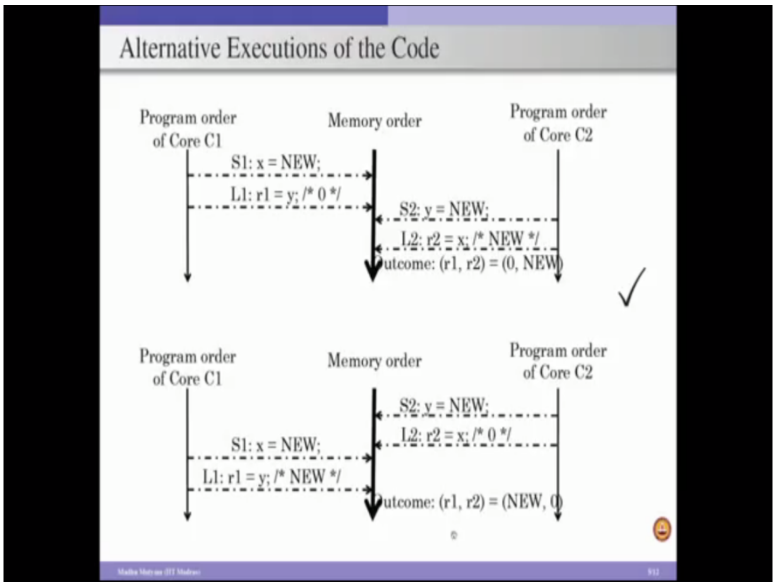
"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." – Lamport

Core C1	Core C2	Comments
S1: x = NEW L1: r1 = y	S2: y = NEW L2: r2 = x	/* Initially, x = y = 0 */

Lamport has defined the sequential consistency as follows. A multicore system is sequentially consistent if the result of any execution is same as if the operations of all the processors were executed in some sequential order and the operations of each individual processor appear in this sequence in the order specified by its program. So, it says that when we have a multicore system where multiple applications are executed on each of these cores. And each of this program is going to execute instructions in the program order and also because we have multiple programs executed simultaneously on this multicore system, we can interleave the instruction execution from all these applications in any order.

So, as long as it produces a value which is consistent with any of the interleaving orders of these instruction executions then we can say that the entire system is said to be sequentially consistent. So, we now explain the sequential consistency concept with an example. So, consider a 2 core system where core 1 is executing a program that consists of a store instruction and a load instruction. And core 2 is executing another program which is having again another store instruction followed by a load instruction. Now, once we have this 2 programs running simultaneously on these 2 cores. Now, we will see how what will be the expected outcome and whether the expected outcome is inconsistent with the sequential consistency definition.

(Refer Slide Time: 17:33)

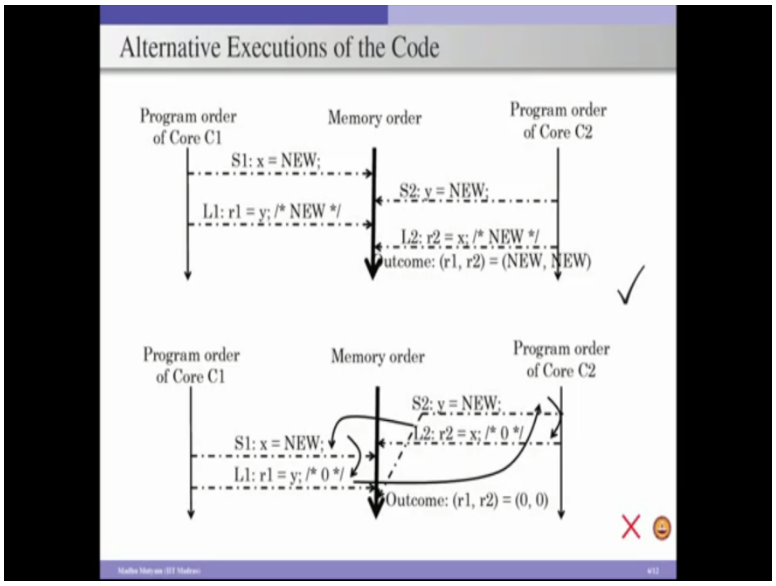


So, here the core 1 can execute its 2 instructions first that is the store 1 and load 1 and after that core 2 can execute its instruction that is store 2 and load 2. So, if we consider at this order. So, store 1 load 1 is coming first and store 2 load 2 is coming next. And here we can clearly see within a program the instructions are in the program order because our first program is having store 1 first followed by load 1. Similarly, the second program is having store 2 first followed by load 2. So, in both the programs we are not violating the program order.

So, this is according to the sequential consistency definition by Lamport. And also, we are executing instructions in some interleaved fashion, here in this particular example we are executing the instructions of program one first followed by instruction from program two. So, as a result we are getting an outcome 0 and NEW and this is consistent with the Lamport's sequential consistency definition. So, as a result we can consider this as valid outcome for this piece of code.

Now, consider another way of executing these instructions. So, here we are actually executing the instructions of second program first then we are executing the instructions from the first program. So, here, so we are executing the store 2 followed by load 2 which is in the program order as specified by second program, then we are executing store 1 followed by load 1 which is again in the program order as specified by the first program and again this is also, one of the interleaving of the execution. And this is again consistent with sequential consistency definition. So, as a result whatever the outcome we are getting here is also valid. So, effectively the outcome NEW, 0 is also valid outcome. So, these 2 are according to the programmer's expectation.

(Refer Slide Time: 19:33)



Now, consider other alternative. So, here we are executing store 1 first and then we are executing one instruction from the second program that is store 2. And then we will come back to program one and core 1 and we execute the next instruction, that is load 1. And finally, we go to core 2 and execute the left over instruction from program that is L2. So, as a result this is also not violating the program orders in individual programs and this is also another interleaving of instruction execution. So, the outcome of NEW, NEW is also valid according to sequential consistency definition.

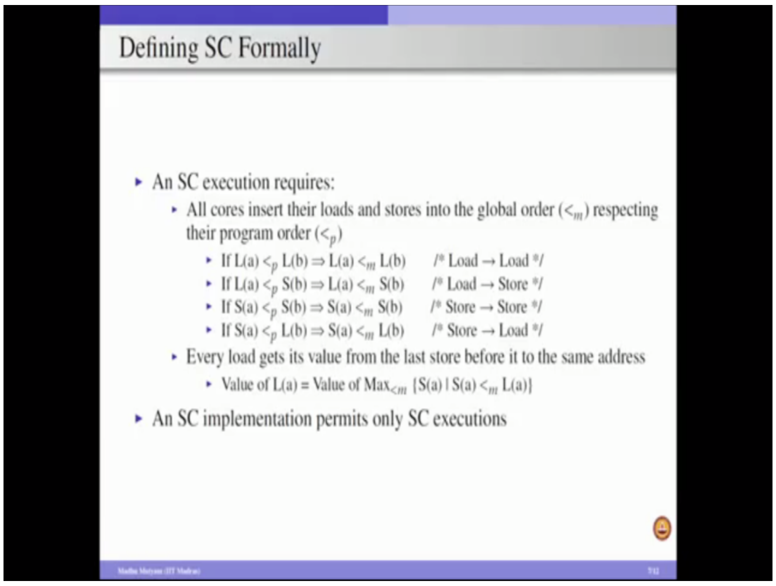
So, once we have a system supporting sequential consistency, then programmer can expect either 0 NEW, NEW 0 or NEW NEW as outcomes for this multithreaded program execution, but so here we may get an outcome of 0 0 also. As long as our hardware is not respecting the memory consistency model definition or the specifications, then it can rearrange its instructions as a result we can get some outcome which is not consistent with our sequential consistency definition.

So, see here in this particular piece of code we are executing first, the load 2 instruction and then we are going to the program one that is executed on core 1 and execute store 1 and then load 1 this is in the program order. And finally, we are executing store 2 from program 2. We can clearly see here we are executing load 2 first before the store 2. And which is actually violating the program order, but according to the sequential consistency definition each of the individual programs running on our multicore system should respect the program order, but here because this load 2 and store 2 these are independent and if our underlying hardware is looking at this independent instructions and then rearrange instructions to execute for performance improvement.

Then it can execute load 2 first before the store 2 and as a result we will get completely unexpected outcome here. And as a result this particular outcome is non consistent with respect to the sequential consistency definition. Now, if we have this memory consistency model defined in our multicore system. And once this is defined then automatically underlying hardware also has to respect this memory consistency model and accordingly it has to look at when it is rearranging the instructions for performance optimizations.

Even when there is set of independent instruction that can be executed in out of order fashion, if it is going to violate our underlying memory model our memory consistency model, then the hardware should not go for this rearranging of the instructions because we are not worried too much about the performance, but we are worried too much about the correctness. As long as the correctness is maintained then we can go for any rearrangement of instructions to extract more performance improvement, but if we get significant performance improvement, but without respecting the correctness then that is of no use. So, as a result we have to give more importance for the correctness then we can think of going for the performance improvement techniques.

(Refer Slide Time: 23:23)



Defining SC Formally

- An SC execution requires:
  - All cores insert their loads and stores into the global order ($<_m$) respecting their program order ($<_p$)
    - If $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$ /* Load → Load */
    - If $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$ /* Load → Store */
    - If $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$ /* Store → Store */
    - If $S(a) <_p L(b) \Rightarrow S(a) <_m L(b)$ /* Store → Load */
  - Every load gets its value from the last store before it to the same address
    - Value of $L(a)$ = Value of $\text{Max}_{<m} \{S(a) \mid S(a) <_m L(a)\}$
- An SC implementation permits only SC executions

Madhu Mutyam (IIT Madras) 7/12

So, after discussing that example, now we will formally define our sequential consistency. So, an execution that is said to be sequentially consistent requires that all the cores insert their loads and stores into the global order respecting their program order. Here this less than p indicates the program order and less than m is indicating the global memory order. So, here when we have multiple applications running on multiple cores of our multicore system and each of the core is actually issuing their load and store instructions from the corresponding application in the program order, then we can construct a global memory order like this.

So, if there is a load instruction from a memory location A and there is a load instruction from memory location B and these two are from the same application and they are following the program order, then in the global order also we can just have this order. So, effectively this particular statement says that, if a load from a memory location A is coming before load from a memory location B from a particular program then in the global order also this load A will come first before load B.

So, this is actually saying that we have to maintain the load to load order. So, that is what it says because this is from a single application, where there actually sending a load instruction for a memory location A and then we are issuing a load instruction from memory location B. And if they are in the program order then in the global order also they will come in the same order.

So, first load A will come and then load B will come. So, now consider another scenario. So, we have a load instruction from a memory location A followed by a store instruction from, to a memory location B in the program order. Now, in the global order also global memory order also this load has to come before this store. So, that is what is given here. So, this is called as load to store ordering. So, if there is a leading load followed by a trialing store in the program order, then in the global order also the load should come first then the store.

Now the third condition says, that if there is a store to a memory location A, followed by a store to a memory location B in the program order. Then in the global order also this store to location A comes first before store to a location B, that is what the store to store ordering. And finally so, if there is store to a memory location A coming first in the program order before load to a memory location B, then in a global order also this store should come before this load. So, that is what the store to load ordering. Effectively when we have sequential consistent model defined in our multicore system, then we have to respect the order between every pair of load and store instructions.

Load to load order should be maintained, load to store order should be maintained, store to store order should be maintained and finally, store to load order should be maintained. And also every load gets its value from the latest store before it to the same address. So, what is this? For example, in our program we have a store instruction to a memory location A and then there is a load instruction from the memory location A, that means this store and load are from the same application and to the same location.

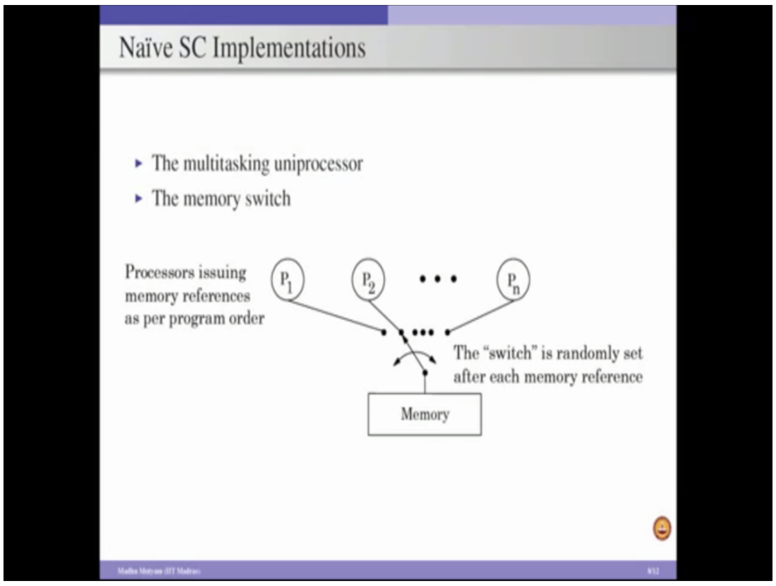
So, as a result the subsequent load should get whatever the value produced by the previous store. So, this will be perfect because our instruction execution from a single program follows the program order, but what about if the store is from one application and the load is from some other application. So, how do we do that? So, effectively let us say program 1 which is executed on core 1 is issuing a store instruction to a memory location A and a program running on core 2 which is actually trying to read the value from the same memory location A.

So, now if in the memory order the global memory order if the store is coming first, then this load from the second core should get the value supplied by this store instruction only. And also there may be multiple writes to the same location, but when we issue to this load instruction then load instruction should get the latest return value to that particular location.

So, that is what we are giving here. So, once we execute the instructions in a sequential consistent manner, then that entire implementation is said to be sequentially consistent.

So, in other words an SC implementation permits only SC executions. When we say our system is supporting sequential consistency then all the instructions whatever they are executed on our system should be supporting sequential consistent execution only. And this sequential consistent execution will respect this load to load, load to store, store to store, store to load ordering. As well as the load is always going to get the latest write to that particular memory location from where this load is going to read. So, how do we implement this sequential consistency in our system?

(Refer Slide Time: 29:51)



Naïve SC Implementations

- The multitasking uniprocessor
- The memory switch

Processors issuing memory references as per program order
P_1
P_2
P_n
The "switch" is randomly set after each memory reference
Memory

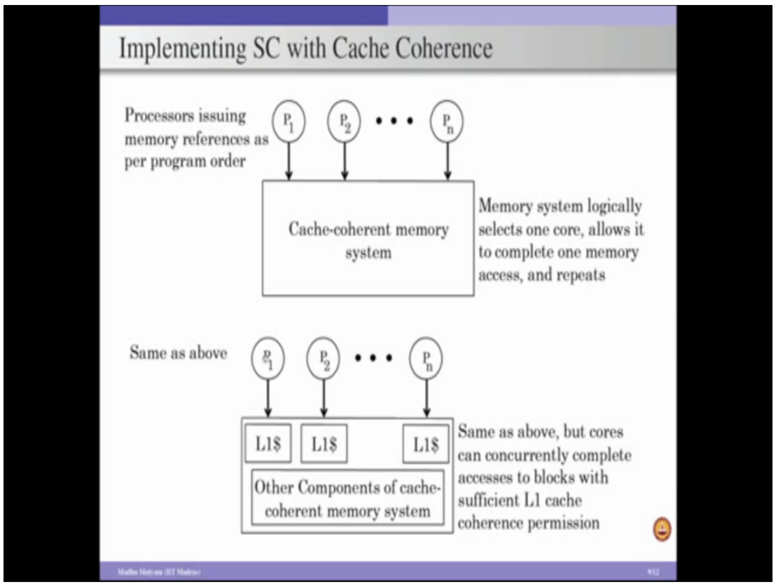
A naive method is, so consider a unicore processor where this unicore processor is actually supporting this multitasking. So, that means, so this unicore processor can execute one task now and then there may be a context switch and then it will go to another task and execute and then again go to third task and execute for some time and then there is a context switch and comeback to the first task and execute.

Effectively so, we have a single core and multiple tasks. And at any point of time this core will execute instructions from only one of the task. And when we are executing instruction from one particular task we know that we are executing the instructions in the program order in that particular task. So, effectively this is like a switch model. Also we can extend this for a multicore system or a multi processor system, where we have multiple processors and multiple cores. And all these cores are connected to the memory.

Now when an application running on any of these cores, when it is issuing the request, then if these requests are load or store request then they have to go to the memory. And now this memory is going to be accepting the request at any point of time only from one particular core. And once it processes that particular instruction then it can move on to another core and then accept any load and store instruction from the corresponding application running on that particular core and so on.

So, in other words this is acting like a memory switch. And there may be requests from all these cores and at any point of time this switch is going to take or accept the request from one particular core at any point of time. And after processing that then it will move on to the next one randomly and then again it will take the request from that particular corrected core and then process and then again switch to some other core and process the request and so on. So, this way we can simply implement our sequential consistency in our multicore memory system.

(Refer Slide Time: 31:59)



Implementing SC with Cache Coherence

Processors issuing memory references as per program order
P_1
P_2
P_n
Cache-coherent memory system
Memory system logically selects one core, allows it to complete one memory access, and repeats
Same as above
L1$
L1$
L1$
Other Components of cache-coherent memory system
Same as above, but cores can concurrently complete accesses to blocks with sufficient L1 cache coherence permission

But most of our current systems are almost all the current multicore systems or multiprocessor systems are actually have the cache coherent memory system, where we will have set of private caches for each of the cores. And then there is a shared cache or the memory and we will have cache coherence support so that the entire memory hierarchy will maintain the cache coherence.

And once we have this we can treat this as a black box. And all these cores are connected to this black box and at any point of time when this multiple cores are sending their request only one request from a particular core can be accepted. And it will be processed by this cache coherent memory system and the response will be supplied. And after that again this system is going to accept a request from some other core and so on. So, for example, consider a bus based simple multicore system where we have 4 cores connected to a common bus and each of these cores when they want to get some data from the main memory or the shared cache, then they place their request on the bus, but note that multiple cores cannot place their request simultaneously on the common bus.

So, for that they have to go for bus arbitration logic and they will request for the bus and bus will be given for only one of the requesting cores. And whoever wins the arbitration they are going to place their transaction on the bus. So, that the memory of the shared cache is going to supply the data for that particular request. And after that the bus will be released and again all other cores again compete for the bus and whoever wins then they will send their request on the bus and so on.

So, that way we can implement our sequentially consistency efficiently in this. At the same time we are having cache support to improve the overall performance, but this particular cache coherent memory system can also be viewed like the set of private caches associated with each of the cores and then the remaining component of the cache coherent memory system. It consist of may be a shared cache followed by the main memory and so on.

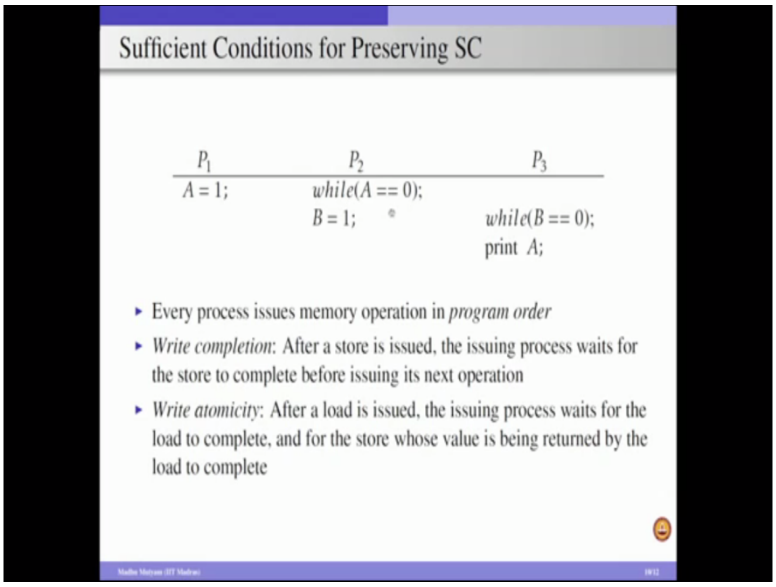
So, once we have this type of design what is going to happen is as long as applications are getting hits for their load and store request in their local caches. Then they do not have to deal with the common bus to send their transactions to the share cache or the memory. So, that way what we are going to do is, once we have this cache coherent system with the private caches then rather than restricting only one core sending the request to the cache coherent system. Now, multiple cores can send their request to their private caches, but if any of these cores miss in their private caches then again they have to go for bus arbitration logic to get the exclusive permission to use the bus, which is connected to the shared cache or the memory.

So that they can send their request on the bus so that they will get the response from the shared cache or the memory. So, at least, as long as, as long as we have our private caches which supply the data for the request from these cores, then we can improve the overall performance. And this also ensures that the parallel execution from multiple cores and that will improve the overall performance.

So, effectively once we have a cache coherent system with private caches and the shared caches, we can implement the sequential consistency efficiently. And at the same time we can ensure the performance improvement by parallelizing the request processing from multiple cores by using the private caches associated with the corresponding cores, but the sequential consistency puts a restriction that we have to respect the ordering between loads and stores.

If there is a load and a store even when these load and stores are 2 different memory locations we have to respect that order in our global memory order. And as a result the sequential consistency is going to degrade our performance.

(Refer Slide Time: 36:24)



Sufficient Conditions for Preserving SC

```
P1          P2                  P3
A = 1;      while(A == 0);
            B = 1;              while(B == 0);
                                print A;
```

- Every process issues memory operation in *program order*
- *Write completion*: After a store is issued, the issuing process waits for the store to complete before issuing its next operation
- *Write atomicity*: After a load is issued, the issuing process waits for the load to complete, and for the store whose value is being returned by the load to complete

So, now we will see what are the sufficient conditions for preserving sequential consistency? So, consider a piece of code, here we consider 3 cores in our system. Where core 1 we are executing a store instruction that is A=1 and core 2 we are executing a branch instruction that is a while (A = 0) condition and then there is a store instruction B = 1. Then in core 3 we are executing this while condition and then we are printing the value of A. So, now when we write this as a programmer our expectation will be.

So, this print statement should print the value of 1 because it is printing A. And this can be possible only when we come out of this while loop. We can come out of this while loop only when B !=0, but B != 0 only when we perform this store operation, but when we perform this store operation that indicates that we already executed the this while loop and we exited from this while loop, but we exit from this while loop only when we have A equal to 1. So, that is what we have given here. So, effectively as a programmer our expectation will be.

So, as long as this instruction is not executed so we will be in this while loop continuously and whenever we execute this instruction, this store instruction then this condition will be false. So, that we will go for subsequent instruction to this while and which is another store instruction that is B=1. And when we perform this then automatically this core will exit this while loop and it is going to print the value A and A will be 1 because we know that in this particular scenario, if B=1 happens then automatically A=1 should have been completed already.

That is what the programmers expectation, but in our system we may not have, we may not have printed A = A we might have printed A = 0. If our system is not supporting the sequential consistency, when the processor one is going to write to a variable A, it is going to send an invalidation signal to all the processors because we are assuming that here our system is having the cache coherence support. When we have a cache coherence support and whenever any processor is going to write to a particular location, the first thing it is going to do is it is going to send an invalidation signal.

Also here we are assuming the invalidation based cache coherence protocol is designed in our system. Also assume that the local caches in each of these processors have initially some value A = 0. And now when we perform this store operation we send an invalidation signal, but unfortunately this invalidation signal is reached only to processor 2, but not to processor 3 or core 3. So, whenever invalidation signal is reached at core 2 this core 2 is now going to perform this, the load operation because A = 0 is effectively it has to load the value.

And it finds that this variable is or the block which is holding this variable A is invalidated. So, as a result it will go to the corresponding memory or it goes to the cache associated with this core 1 and it gets the value A = 1. And once A = 1 is read then automatically this while condition is false in core 2 and then it immediately executes B = 1. When core 2 is going to write a value 1 to the variable B it is going to send an invalidation signal. And assume that this invalidation signal is immediately reach core 3, but the previous invalidation signal whatever is sent by core 1 is not yet reached core 3. That may be because the path connecting between core 1 and core 3 is highly congested and request is stalled somewhere.

So, next time when core 3 is executing this while loop effectively it has to load the value of B because previously B is invalidated. So, effectively now it goes to cache associated with core 2 to get the latest value of this B. And it is going to get the value as 1 for variable B. So, this while loop is exited and then immediately it is going to execute it is printf statement because when it is going to print the value to A because it is local cache of this core 3 still having the old value.

The reason is whatever the invalidation signals sent by core 1 has not yet reached core 3. So, as a result it is not invalidated its previous A value. So, when we are going to print the value A, then it is going to print value as 0 and that is actually inconsistent. And so, as a result when we want to implement sequential consistency in our model, in addition to respecting the program order, in addition to respecting the global memory order, we also need to have set of conditions, that are sufficient conditions to ensure the sequential consistencies.

Every process issues a memory operation in the program order. So, that is we already mentioned in the formal definition of sequential consistency. And we have to consider write completion. So, write completion says that after a store is issued, the issuing process waits for this store to be complete before issuing any other operation in the program order from that particular application. So, consider simple example, let us say core 1 is now issuing a store instruction and after this A equal to 1 there may be another instruction, but this core 1 cannot execute this subsequent instruction, unless this store is said to be completed.

When I say store is said to be completed, it has to send an invalidation signal, if we are considering an invalidation based protocol it has to get the acknowledgement from all the cores, then only it can write the value to the corresponding variable. Or in another words we can perform the store operation and then only we can proceed with a subsequent instructions in the program order. So, first one says all the instructions should be executed in the program order, but the second one says when there is a store instruction we can not proceed with the subsequent instruction, unless this store is said to be completed. And the third condition says that the right atomicity.

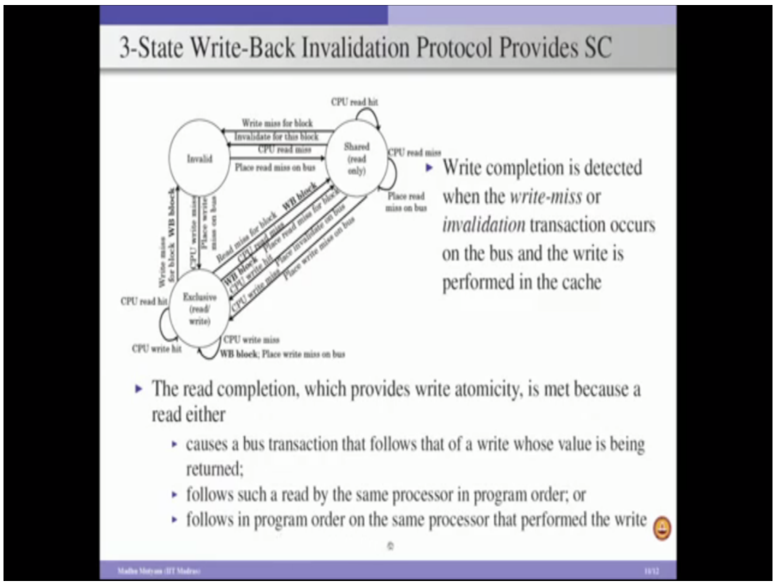
So, here after a load is issued by a process, the issuing process waits for this load to complete as well as it has to wait for the store which is actually supplying the value to this particular load. So, it has to wait for the store to complete. In other words let us say here we consider. So, processor p2 issuing a load instruction that is A = 0, when this load is issued then this process which is running on processor p2, cannot start executing the subsequent instruction unless this load is completed as well as any store which is actually supplying the value to this particular store.

So, here we can clearly see this load is actually loading value from memory location A. Similarly, this process which is running on processor p1 is actually performing a store operation to the memory location A. So, now we cannot start executing this B = 1 unless this load operation is completed as well as this store operation is completed. If we ensure that then this piece of code is going to print value A = 1 in core 3 because when we are executing this printf statement.

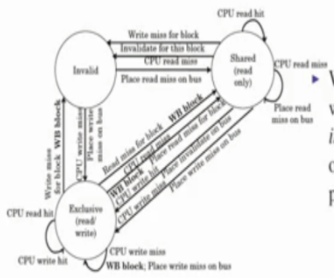
So, that indicates that we exited from this while loop and when we exited from the while loop. So, we have B = 1, but when B = 1 is performed that indicates that the previous operation is already completed, but the previous operation consists of a load. And this load is said to be completed only when any store which is actually supplying the value to this load is also said to be completed. So, that means this A = 1 is said to be completed. When A = 1 is completed then this printf statement is going to print a value 1. So, that is what is write atomicity.

So, these are the sufficient conditions to ensure the sequential consistency. In summary this when we want to maintain sequential consistency, we have to go for these set of conditions. These are right atomicity which ensures that you cannot proceed with any instruction subsequent to a load instruction unless the load is completed as well as any store which is supplying the value to this load is also said to be completed. And we also have to maintain the write completion, where when we issue a store instruction we cannot proceed with subsequent instruction unless this store is said to be completed. And also we have to issue the instructions in the program order and this program order condition will be for each of the programs.

(Refer Slide Time: 46:02)



3-State Write-Back Invalidation Protocol Provides SC



- Write completion is detected when the *write-miss* or *invalidation* transaction occurs on the bus and the write is performed in the cache

- The read completion, which provides write atomicity, is met because a read either
  - causes a bus transaction that follows that of a write whose value is being returned;
  - follows such a read by the same processor in program order; or
  - follows in program order on the same processor that performed the write

Now, we will see whether our 3 state write back invalidation protocol whatever we discussed as part of cache coherence protocol, will provide the sequential consistency or not. So, for sequential consistency, we have to ensure that our sufficient conditions are met. The first one is issuing the instructions in the program order. So, here we assume that all our programs are issuing the instructions to the program order. So, we do not have to worry about the first condition, but the second condition is the write completion and the third condition is right atomicity.

Now, we will see the write completion. So, write completion is detected whenever a write miss or invalidation transaction is placed on the bus and the write is performed in the cache. According to our 3 state the write back invalidation protocol, whenever we are going to perform any write operation, if it is going to get a hit in the local cache then it is going to place an invalidation signal on the bus. For example, see here you may have a block in the shared state and you are going to perform a write operation.

If the write is a hit then we are going to change the state from shared to exclusive here and we are going to place a transaction that is invalidation transaction on the bus. So, that any other caches which have the data in the shared state for this particular block they are going to invalidate by using this state transition. So, shared to invalid. So, as a result once we place this invalidation signal, everyone in the bus based multicore system will invalidate their copies. So, that the requesting core which is going to perform this write operation will actually perform the write operation on the corresponding block.

Similarly, so if you have a block in the exclusive state and if there is a write request because exclusive state indicates that only one particular cache has that particular block in the modified state. So, if there is a write operation then you do not have to place any transaction on the bus. And because you do not have to communicate this information to all other the caches because they do not have the data. So, as a result you can peacefully perform your write operation there without changing the state.

In this particular case you do not have to place any the transaction on the bus. And also because you have the data in the modified state you can perform multiple writes to the modified block. So, as a result it is not going to create any problem. So, in this case also you can say this write is said to be completed. Whenever you have a block in dirty state or a modified state and you are performing a write operation, you do not have to explicitly communicate with all other caches. And you can perform your write and this write is said to be completed locally, but this is not going to create any incorrectness according the memory consistency.

Now, consider another scenario. For example, you want to perform a write operation to a particular block and you incur a miss. Whenever there is a write miss then for this for example, CPU write miss then we are going to place write miss transaction on the bus. Whether the block in the shared state or in an exclusive state, for example, here also if there is a write miss then we are going to place write miss transaction.

So, that means whenever there is a write miss we are going to place a write miss transaction and all the cache controllers will snoop on the bus. And they respond to this write miss transaction and invalidate if they have the copy in their local caches. So, after they invalidate then the requesting core can perform the write operation. Then we can say this write is said to be completed. So, whether it is a write hit or a write miss. So, using this write miss transaction or invalidation transaction on the bus, we are performing this write completion successfully.

So, this is satisfying the second sufficient condition for our sequential consistency. Now, we will see the right atomicity. So, right atomicity says that when we issue a load instruction we cannot proceed with any subsequent instruction to this load instruction, unless this load is said to be completed. And also if any other store which is producing the value for this load that also needs to be completed. So, this right atomicity is actually provided by using this read completion.

So, when we have a read completion then according to this state transition diagram, this read completion is met because a read is either causing a bus transaction that follows that of a write whose value is being returned. For example, there is a write operation from core 1 and it sends either the write miss transaction or an invalidation transaction on the bus. So, that if core 2 has that particular block of data it invalidates. And after that let us say core 2 is issuing a load instruction. Now, core 2 load instruction see that the block is invalid and then it has to request for that particular block.

Then core 2 is going to place a transaction that is read miss transaction on the bus. So, that is what is specified here. So, if there is a request read request from core 2, this core 2 can send a bus transaction and this bus transaction actually follows a write operation or a transaction corresponding to a write request from core 1. So, that is what is given here because this core 2 is going to get the value supplied by a store instruction produced by our core 1. So that is what is. So, effectively when this read operation is initiating a bus transaction this bus transaction will follow that of a write instruction which is producing the value that is required by this load instruction. Or other scenario, this read instruction is actually following another read instruction by the same processor in the program order. Or consider another scenario where core 2 is actually requesting the block of data. And this block of data is either available in it cache or this block is invalidated by a store from core 1. In that case according to this first condition the load request from core 2 is going to get the data. And after this if core 2 is generating another load request for the same block.

Now, the second load is actually following the first load in the program order. So, that is what is mentioned here. So, this read operation can follow a similar read operation from the same processor in the program order. And the previous read operation can be a hit in the local cache or a miss in the local cache. In both the cases it is going to get the appropriate data supplied from a previous store operation or it is supplied from the shared cache. And the third scenario is this read operation can follow the program order on the same processor that performs a write.

For example, consider a different scenario altogether where core 2 is actually generated a write request first and then there is a read request for the same memory location. So, now when we come to the load request for the same memory location so this load request is actually following the previous store request from the same processor in the program order. So, that also ensures that our read is actually getting the correct data and it said to be completed.

So, effectively we ensure that right atomicity from across the processors or from the same processor. So, that ensures that our read is said to be completed by using this 3 state cache coherence protocol. And also we can ensure write completion by using these corresponding transactions placed on the bus and that also satisfies our 3 conditions whatever we require to ensure sequential consistency. So, with that I am concluding this sequential consistency the discussion. And in the next module I am going to discuss the total store order memory model as well as relaxed consistency model.

Thank you.