

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

Module – 09
Lecture – 33
Memory Consistency (Part-2)

So, in the last module we discussed sequential consistency model. And in this module we are going to discuss the total store order and relax consistency models. So, we know that from the sequential consistency a model that we need to consider program order, we need to consider the write completion and write atomicity. And once we put these conditions then the overall performance will be degraded in the system. And also we may not utilise hardware performance optimization techniques efficiently. So, for example, consider scenario where we have the write buffers in our process from the hardware point of view.

(Refer Slide Time: 01:01)

Write Buffers Cause Problem to SC

- ▶ Processors use write buffers to hold committed stores until the stores are written back
- ▶ Write buffers are not a problem for a single-core processor
- ▶ Multi-core processors can produce the result $(r1, r2) = (0, 0)$, which is forbidden by SC

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

The diagram illustrates the execution flow for two cores, C1 and C2. On the left, 'Program order of Core C1' shows a vertical arrow pointing down, with a dashed box containing 'S1: x = NEW;' and 'L1: r1 = y; /* 0 */'. On the right, 'Program order of Core C2' shows a vertical arrow pointing down, with a dashed box containing 'S2: y = NEW;' and 'L2: r2 = x; /* NEW */'. In the center, 'Memory order' shows a vertical arrow pointing down. Dashed arrows indicate that Core C1's store S1 is written to memory before Core C2's store S2. However, Core C2's load L2 is executed before Core C1's load L1, reading the value from its local cache (NEW) before Core C1's store is written back to memory. This results in an 'Outcome: (r1, r2) = (0, 0)'. A small orange icon is visible in the bottom right corner of the slide.

And these write buffers are mainly helpful for performance improvement. The reason is whenever there is a store instruction then processor can write this store value on to write buffer and from the processor point of view this store operation said to be completed once we write that to the write buffer. And so after that we can take the instructions from this store buffer or write buffer and we can send it to caches and so on. So, but if we consider sequential consistency model in our system then we cannot use this write buffers. So, this write buffer concept is going to create a problem with respect to the sequential consistency

model. So, we will see an example here. So, consider a 2 core system where in core 1 we are going to execute a store instruction followed by a load instruction.

And similarly, in core 2 also we are going to execute a store instruction followed by a load instruction. Now, for example, if we have write buffers in our multicore system what is going to often is when core 1 is executing this store instruction $x = \text{new}$ then it will store this instruction in our store buffer and so whatever the value written on to the store buffer may not be immediately reflected or communicated to the other cores so as a result when core 2 executing its load instruction $r2 = x$ it will read the old value. Here the assumption is both the caches associated with the core 1 and core 2 have initial values of this x and y as 0. Now because the store operation from core 1 is not communicated to core 2.

So, as a result core 2 is going to read value 0. Similarly, store operation of core 2 is not communicated to core 1 so as a result load operation in core 1 is going to read the old value. So, when we have write buffers in our multicore system and if we assume that we have an invalidation based cache coherence protocol. So, let us see if core 1 is performing a write operation. So, according to cache coherency protocol so it cannot immediately write to a particular block, it has to send an invalidation signal to all the cores in a multicore system.

And once it gets the acknowledgement from all the other cores stating that they have invalidated their copies, then this core 1 is going to perform the actual write operation, but waiting for the acknowledgement to receive will take longer time. So, and it will waste the processor cycles for core 1. So, as a result one of the optimization what we can do here is we can consider write buffer so that this core 1 is going to write his value to the store buffer and it proceeds with the subsequent instructions. And as soon as it gets the acknowledgement for the invalidation signals from all the other cores in the multicore system then we take this store instruction from this write buffer and we write it to actual cache.

So, this is the whole the operations we are going to perform when we are dealing with the store instructions. But now if we consider sequential consistency model in our system Sequential consistency says that we have to strictly follow the program order between all the memory operations. So store to load, load to load, load to store, store to load and store to store. We have to consider strictly this program order as a result when core 1 is performing a store operation it cannot proceed with any other subsequent instructions until this store is said to be completed.

And that is going to degrade the overall performance. So, as a result in order to utilise these write buffers for performance optimisation then we have to relax our sequential consistency model. So, in this particular example we actually consider 2 cores and each core is performing a store operation followed by load operation. And assume that the hardware is supporting the write buffers. And now when each of these cores having the write buffers and whenever they perform a store operation.

So, this store operation will be written to the write buffer. And it is not actually communicated to the other cores. So, as a result when core 1 is performing store operation this x equal to NEW is not communicated to core 2 and core 2 is performing a load operation it can load the old value. Similarly, when core 2 is performing a store operation so core 1 can read the old value for that corresponding variable and that is going to violate our sequential consistency expectations. According to sequential consistency in this particular case we are supposed to get $r1$ and $r2$ as NEW and NEW but here because of this write buffers we may get values of 0 and 0 where 0 and 0 are initial values of x and y .

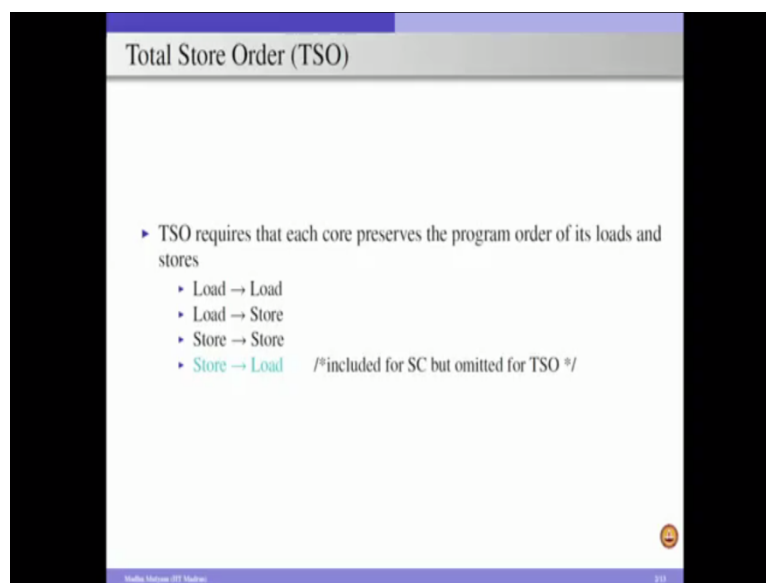
So, that is mainly because so, this store is performed or written to the store buffer and this store buffer cannot be read from the other core. Similarly, this store is going to perform to a its corresponding write buffer and the contents of this write buffer cannot be read from this core. Because of this, so our expected outcome will be different with respect to the sequential consistency model. But now the question is whether we want to go to the sequential consistency or whether we want to utilise the hardware performance optimisation techniques or not.

Generally the processor vendors will go for this hardware based performance optimisation techniques and they want to consider a relaxed consistency models. So, that programmer can expect more than whatever the possible outcomes that will be available with respect to sequential consistency model. So that the programmer will expect extra outcomes in addition to our sequential consistency model behaviour. Now, in order to utilise write buffers in our processor, now we have to see what is that we have to relax in our sequential consistency model requirement. So, that we can come up with a new memory consistency model.

So, here actually once we have write buffers our whole idea is each of these processors can proceed with the subsequent load operations which are following the previous store operations because these store operations are written to a store buffer and it is not actually

communicated or written to the actual cache. And before communicating to the actual cache or before communicating to the other cores we want to proceed with the load instructions. So, that means here we want to go ahead with subsequent load instructions after any store instruction. In other words, we want to relax one particular order among the sequential consistency that is store followed by a load. If we relaxed that then we can expect this 0 0 as one of the valid out comes for this piece of code

(Refer Slide Time: 08:50)

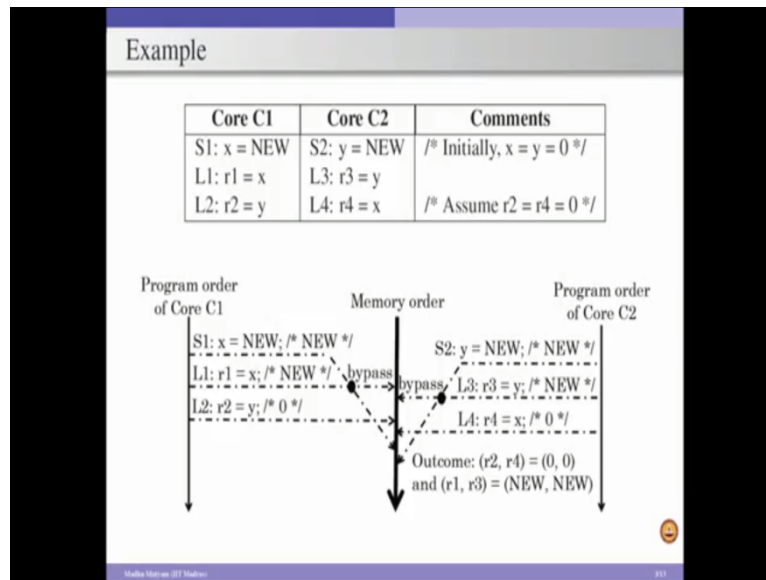


So, that is what total store order memory consistency model says. So, here the TSO requires that each core preserves the program of order of it loads and stores for these 3 conditions, but not for this. If you have all this 4 then this is exactly same as sequence consistency model, but in total store order because we are going to exploit write buffer concept for performance improvement so we relaxed this particular condition. In place of this we actually consider write buffers and we also consider these write buffers are performing in FIFO order, first in first out order when we writing to the write buffer. So, we still enforce load to load order, load to store order, store to store order but whenever there is a store followed by a load, then we can perform subsequent load operation before actually completing the previous store operation.

So, that is the relaxation we consider and that is a reason why this is also called as a one of the relaxed consistency model compared to the sequential consistency and the sequential consistency is a rigid consistency model and that is going to degrade the overall performance.

So, that is the reason why we are considering this total store order and most of the x86 based processors are actually using this total stored order memory consistency model in their systems.

(Refer Slide Time: 10:20)



So, consider an example. So, here core 1 is performing a store operation, storing a value NEW into memory location x and after that it is performing 2 load operations loading the value stored in memory location x to r1 register and similarly loading value stored in memory address y into register r2. Similarly, core 2 is going to execute this piece of this core where it is writing value NEW to a memory location y and after that it is going to read value stored in memory location y to register r3. After that it is going to read a value which is loaded memory address x to register r4. We also assume that initial x and y are 0 and also r2, r4 are 0. Now, if we expect let us say r2 r4 are 0 and 0. Then our expectation of r1 and r3 also 0 and 0, but that not be the case because we have our cores equipped with the write buffers and because of this write buffers. So, our outcomes will be slightly different.

So, here because when we are performing this store operation, this store is returned to the write buffer and after that there is a load instruction from core 1. And this load is to the same location where the previous store is performed. So, when we are executing this load instruction what we do is we actually go to the write buffer and see if there is any matching request or not. If there is a matching request then we can supply that value to the corresponding register and then that register will be now loaded with the latest written value

from that particular core, but this value is there in the write buffer and this write buffer cannot be read from the other core.

So, as a result when this core is actually trying to read the value from the memory location x it is going to read the old value only and the same logic. So, here when is performing as store operation this store will be written to a store buffer and store or a write buffer both are used interchangeably. And when it is performing a load operation to load the value from the same memory address location then it is going to get the value of NEW from the write buffer associated with this core.

And this write buffer context cannot be read from this core. So, as a result when the core 1 is going to perform the second load operation it is going to read value 0 that is old value. So, as a result with this a piece of code executed on a multicore system where each core is having write buffer then our outcomes will be r2 and r4 are 0 and 0, but r1 and r3 are NEW and NEW. Again so, here we assume that each of the cores has write buffer and also we are considering a total stored order mechanism.

So, when we consider total store order memory consistency model. So, before this store is set to be completed we can proceed with subsequent instructions and when we perform the first load is instruction it can get the value from the write buffer that is also a called as a bypassing logic. So, it is effectively this load is going to get the value from the write buffer and this is the load bypassing, so we just get the value and the subsequent load which is L2 to which is going to read its value from the cache. So, that has the stale value. So, as a result we are getting this. So, note that this write buffer is not going to create any problem when we consider a single core system because an a single core system whenever we have a load operation, this load is going to get the latest written value to that particular location.

The reason is whenever we have a previously stored instruction, the store can be written to the write buffer or store buffer. And after that there is a load instruction this load will first consult the write buffer if there is a match, then the value will be supplied if not then it will go to the next level cache and so on. As a result in a single core system write buffer is not going to create any problem even when you have sequential consistency model, but whereas in the case of multicore system when we have write buffers write buffers contents can be available only to the associated core, but not to the cores.

So, that means we cannot access the write buffer of core 1 from core 2 because these write buffers are not visible to the other cores. If you want to make these write buffers of each core to be visible to the other cores, then the design will be very complex and we cannot go with that particular type of design. So, in summary, we can use the hardware optimisation technique for performance improvement in terms of write buffers. So that processor will not be stalled for completing the store operation before proceeding with the subsequent operations.

And also we can relax our store to load ordering so that we can come up with new memory consistency model and that is called as a total store order memory consistency model. And once we have a total store order memory consistency model as a programmer we can expect the other outcomes also and here any subsequent load to store operation, if both are to the same location, then we can get value from write buffers. So, as a result programmer will add the corresponding outcomes also as parts of his expected list of outcomes for the piece of code he is dealing with.

(Refer Slide Time: 16:29)

The slide, titled "Defining TSO Formally", lists the following rules for Total Store Order (TSO):

- ▶ All cores insert their loads and stores into the global order ($<_m$) respecting their program order ($<_p$)
 - If $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$
 - If $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$
 - If $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$ /* Enable FIFO write buffer */
- ▶ Every load gets its value from the last store before it to the same address
 - Value of $L(a) = \text{Value of } \text{Max}_{<_m} \{S(a) \mid S(a) <_m L(a) \text{ or } S(a) <_p L(a)\}$ /* Need bypassing */
- ▶ Use FENCE to enforce Store \rightarrow Load order
 - If $S(a) <_p \text{FENCE} \Rightarrow S(a) <_m \text{FENCE}$ /* Store \rightarrow FENCE */
 - If $\text{FENCE} <_p L(a) \Rightarrow \text{FENCE} <_m L(a)$ /* FENCE \rightarrow Load */

So, now we will formally define that TSP, total store order memory consistency model. So, here the TSO says all the cores insert their loads and stores into the global order respecting their program order. So, each of the cores will insert their load and store instruction in the program order especially, when we are dealing with the load to load, load to store, store to store. So, that is what we given. For example, if we have a load operation followed by

another load operation, but these 2 load operations are 2 different locations in the programme order.

Then in the global order also it will be coming in the same order. So, if load from a location A is coming before load from location B in the programme order, then in the global order also load from A will come before load from B. Similarly, if load from A is coming before store from B in a programme order then in the global order also A load from a will come before a store from store to a location B. Similarly, if there is a store operation to a memory location A comes before store to memory location B, in the programme order then in the global order also store to a location A comes before a store to a location B because in the TSO we are actually considering the write buffers.

So, we consider FIFO first in first out based write buffers here so that any subsequent load wants to read then it is going to read the latest written value. And every load gets its value from the last store before it to the same address. So, that is nothing but the value of load from a memory location A will be the latest store to that particular location in the global memory order or previous store to the same location in the program order. So, actually this second component is talking about bypassing where as this talks about the global order. So, this store can be from any other core. So, this part is same as sequential consistency model but because in our TSO we are using write buffer. So, as a result we are adding this component also. That means our load can get the latest written value either from the other cores or from the same core. So, that is what is the overall meaning of this particular sentence and if we want to enforce strict order between store and then load and then we have to use fence instructions.

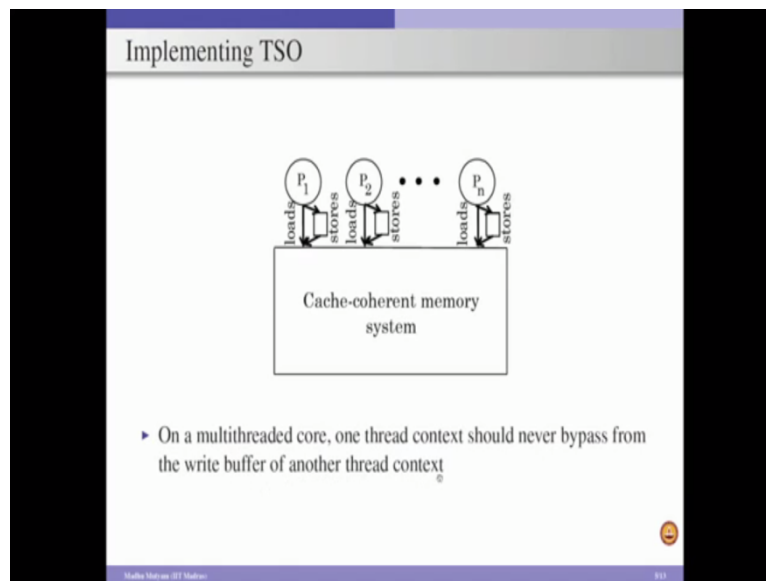
This fence instruction will be an address less instructions. And once we have this fence all the instructions before this fence from that particular core needs to be completed in the programme order. And it cannot proceed with any other instruction subsequent to this fence until all the instructions before this fence are completed in the programme order. So, that is what the meaning of this. For example, if there is a store instruction to memory location A which is coming before this fence instruction in the programme order in the global order also this store instruction will come in before the fence.

So, that is effectively we are following this store to fence order. And similarly, if there is a fence instruction coming before a load instruction from a memory location A in the programme order, then the same thing will be reflected in the global order also. So fence will

come before the load instruction in the global order. So, that is going to say fence to the load ordering. So, whenever we want to strictly enforce that the previous store needs to be visible to every other core then we have to consider a fence instruction immediately after that store. So that when we are going to execute the subsequent load this store is already available to everyone. And then as a result we can strictly enforce load to store ordering.

In other words if you want to come up with the sequential consistency model on top of TSO all way have to do is we can insert a fence instruction between a store and load. If store and load are coming one after another in the programme. And note that when we issue a fence instruction, it is going to affect only within that particular core where we are actually executing this fence instruction. For example, this fence is issued in core 1 this fence instruction is not going to affect the ordering of the instructions one core 2, core 3, core 4 and whatever the other cores but whereas it is going to the affect the ordering of instructions in core 1 only.

(Refer Slide Time: 21:40)



So, now when we want to implement the TSO then what we can do is we actually add extra buffer that is called as write buffer or store buffer so that all stores will come to this store buffer and processor need not wait for these store instructions to be completed. And it can proceed with the subsequent load instructions. And if any subsequent load is actually matching with the previous stores and if the previous store is there in the store buffer then we

can bypass the value and the load can be getting the value from the store buffer so that the processor can proceed with subsequent instructions.

And whenever the cache coherence system is taking care of the invalidation signals and so on and then we can start writing these values from the store buffer to the actual cache. And if you want to strictly enforce this store to load ordering then we have to insert a fence. So, that when we insert a fence for example, here in core, in the core that executed on the processor p1 then immediately all the instruction that all there in the store buffer needs to be returned to the cache. Then only we can proceed further with the remaining instructions following the fence instruction.

In other words whenever we insert a fence we have to drain our store buffer, when I say draining the store buffer we have to take each instruction in the FIFO order. And then we have to write it to our subsequent caches in our cache coherent memory system. So, that is the reason why. So, when we are using fences we have to be selectively use these fences instructions. Otherwise this is going to degrade the overall performance. Whenever there is a necessity then only we can go for fence instructions otherwise we can just not insert any of this fence instructions in our program. When we have this, the store buffers and if there is a context switch happen. And at the time of a context switch we have to write the values of the store buffer to the caches.

So, we cannot use this content for subsequent thread which is executed on any of these cores and so on. So, effectively so whenever we issue a fence instruction or whenever there is a context switch thread context switch then we have to ensure that we drain the store buffer. So, this statement clearly says that during the thread context we have to drain otherwise like the subsequent thread may read the values from the store buffer and that may be wrong. So, in another words one thread context should never bypass from the write buffer of the other thread context. So, now we will see the other relaxed consistency model and before that we are going to give an example of why we want further relaxation in our memory consistency model.

(Refer Slide Time: 24:36)

Example: Opportunities To Reorder Memory Operations

Core C1	Core C2	Comments
S1: data1 = NEW S2: data2 = NEW S3: flag = SET	L1: r1 = flag B1: if(r1 != SET) goto L1 L2: r2 = data1 L3: r3 = data2	/* Initially, data = 0 flag != SET */ /* L1 & B1 may repeat many times */

- ▶ Programmers expected orders:
 - ▶ S1 → S3 → L1 loads SET → L2
 - ▶ S2 → S3 → L1 loads SET → L3
- ▶ In addition, SC and TSO also require S1 → S2 and L2 → L3
 - ▶ Not needed by the program for correct operation

So, consider 2 core system where core 1 is executing 3 store instructions. One is writing value NEW to memory location data one, another one is writing NEW value to another memory location data 2 and third one is writing set value to a flag, a memory location or variable does not matter. And similarly, core 2 is executing 3 load instructions but it also executes condition instruction or a branch instruction where this branch instruction checks whether r1 is equal to set or not. If it is not equal to set then it will again go back to this load, first load instruction and it will be in the infinite loop. And when r1 is equal to set then it is going to execute L2 and L3. So, here both the data 1, data 2 are initially 0 and flag is not equal to set.

And now will see in what way we can execute these instructions in our program. If you see core 1 code especially this 2 instructions are independent S1 and S2, we can execute this S1 and S2 in any order. Of course, we cannot execute this S3 before this S1 and S2 though these 3 store operations are independent, but if we execute S3 before S1 and S2 then what is going to happen is our r2 or r3 may get the old value that is 0, but according to the programmer's intuition. So, the idea is, so when we execute this instruction, the last store instruction, his idea is these 2 instructions are already executed. So, that when we come to this L2 and L3 then these are going to get the NEW value.

So, this is what his intension when he writes this piece of code. So, as a result we have to execute S3 after this S1 and S2, but there is no need of executing S1 first and then only S2.

We can execute S2 first and after that S1. And finally, we can execute S3 and so on. Similarly, in this core 2, so here it does not matter whether we execute L3 first or L2 first among these 2 instructions. Of course, when we come here already the condition is false.

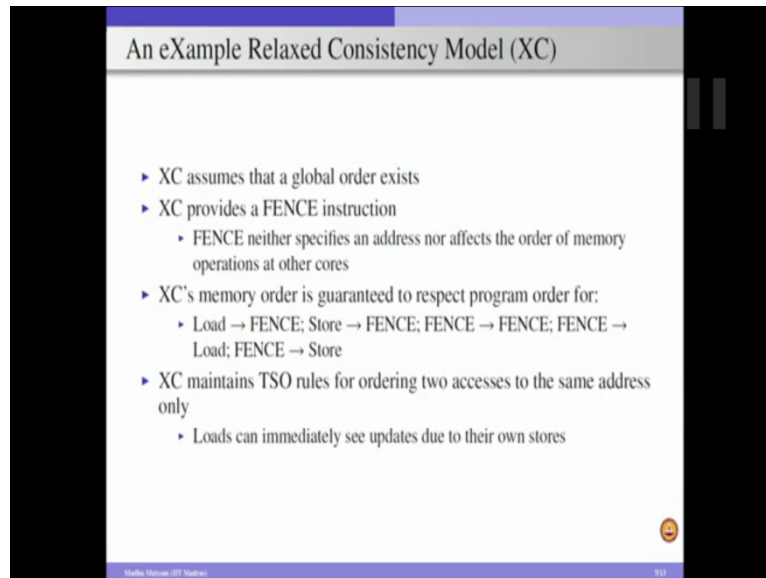
So, as a result we come here, but when we come here we can execute this L2 and L3 in any order it does not matter. So, programmer's expectation is either S1 executed first followed by S3 followed by L1 followed by L2 or he expects S2 should be executed first followed by S3 followed by L1 followed by L3. So, as long as the programme is executed like this he is going to get expected outcome so everything is fine. But if you are considering sequential consistency model or total store order model then it clearly says that, if there is a pair of store instructions, then we have to strictly follow the store to store ordering.

So, it means if you consult TSO or SC then we have to execute S1 first then only we can execute S2, then we can go for S3 and also sequential consistency and TSO requires that load to load order also should be maintained. So, it means L2 should be executed first then only L3 can be executed, but this is actually not necessary according to this piece of code. As long as if S1 is executed before S3 or S2 is executed before S3. So, he will get the expected outcome. So, everything will be correct. So, there is no need of strictly enforcing order between S1 and S2 similarly L2 and L3. So, this is what is expected but when we have SC or TSO implemented in our systems.

Then automatically we require or we force the instructions to be executed in this order and that is actually not necessary and that is going to degrade, that may degrade the overall performance. For example, if this store is, let us say incurred a miss then subsequent store cannot be executed until the previous store is completed and so on. So, that is actually degrading the performance. So as a result when we want to go for performance improvement mechanisms.

Then we need to further relax some of these conditions. So, when we relax these conditions then we can come up with the NEW memory consistency model. So, that programmer will reason his program according to this relaxed consistency and accordingly he will expect set of outcomes. And then also the hardware side we can use optimisation mechanisms to exploit out of order execution or independent instruction executions so that the overall performance can be improved.

(Refer Slide Time: 29:54)



An eXample Relaxed Consistency Model (XC)

- ▶ XC assumes that a global order exists
- ▶ XC provides a FENCE instruction
 - ▶ FENCE neither specifies an address nor affects the order of memory operations at other cores
- ▶ XC's memory order is guaranteed to respect program order for:
 - ▶ Load → FENCE; Store → FENCE; FENCE → FENCE; FENCE → Load; FENCE → Store
- ▶ XC maintains TSO rules for ordering two accesses to the same address only
 - ▶ Loads can immediately see updates due to their own stores

So, we can come up with an eXample relaxed consistency model, we can name it as XC. And here we assume there is a global order exists among all the memory instruction and so on. And we also use a fence instruction similar to the TSO. And so this fence will not specify any address and also it will not affect the order of memory instructions or memory operations in any other core. Other than the core where we are actually applying these fence instructions. And so whenever we want to enforce the ordering among the instructions we can separate this pair of instructions with a fence instruction.

So, effectively if you have a load and a store and if you want to execute load first and after that store then we will insert a fence instructions between this load and store. So, effectively once we have a fence instruction between any pair of memory instructions all the instructions before the fence should be completed before we executing any instruction following the fence instruction. And this relaxed consistency model maintains total store order rules for ordering 2 accesses to same address only.

This is actually says that if you have a load instruction to a same location where there is store operation performed previously. Then this load has to get the value from the previous store, but if the load is to a different location as compared to previous store, there is no need of enforcing store to load order in that. So, that is what we are actually mentioning here. So, only when there is a necessity then we will order instructions otherwise we are not going to enforce any order any ordering between any of the memory instructions. So, in other words

XC maintains the TSO rules only for the accesses where these access are to the same memory location.

So, because of that loads can immediately see updates due to their own stores in the corresponding store buffer. So, that loads can get the value can store buffers and if the loads are to the different memory location as compared to the previous store operations we do not have to keep our load operation to be waiting until the store operation is executed or whatever.

(Refer Slide Time: 32:22)

Example: Using FENCES Under XC

Core C1	Core C2	Comments
S1: data1 = NEW S2: data2 = NEW F1: FENCE S3: flag = SET	L1: r1 = flag B1: if(r1 != SET) goto L1 F2: FENCE L2: r2 = data1 L3: r3 = data2	/* Initially, data = 0 flag != SET */ /* L1 & B1 may repeat many times */

► FENCES ensure that

- S1, S2 → F1 → S3 → L1 loads SET → F2 → L2, L3

So, now consider this example again. So, according to the programmer's idea. So, he wants either S1 to be executed before S3 or S2 to be executed before S3, but he does not care about the ordering among S1 and S2. Similarly, he wants L1 to be executed before L2 and L1 to be executed before L3, but he does not want the ordering between L2 and L3. So, there is typo here it should be L3 load 3.

Now, because he does not want ordering among this store 1 and store 2 similarly, load 2 and load 3 what we can do is we can put a fence instruction here and here. Once we put a fence instruction here that indicates that. So, we do not care about how the ordering of these operations in execution but when we want to execute this instruction these 2 should be completed. In other words any instruction which is coming after fence can be started executing only when all the instructions which are coming before this fence instruction said to be completed. Similarly, here whenever we want to execute either L2 or L3. So, it is

compulsorily required that L1 should be completed but after that it does not matter whether we execute L3 first or L2 first.

So, that is what the meaning of this and now once we put these fence instructions then it is according to programmer's expectations and we will get the expected outputs. So, once fences are there it ensures that S1 S2 will be executed before this fence and after this fence only S3 will be executed. And also after that we are executing this L1 which is loading the NEW set value. And after loading NEW set value automatically this condition is false then we will execute the fence instruction and after that we can execute L2 and L3 in any order. Because here S1, S2 are separated by comma, this indicates that it does not matter how S1 S2 are executed. Whether S1 is executed first or S2 is executed first but where as if we have this S2 -> F1 that indicates that S2 should be completed before the fence instruction. And similarly, F1 and S3 that means this fence instruction should be executed before executing this S3 instruction. So, this is the overall idea of this the fences in XC.

(Refer Slide Time: 35:05)

The slide, titled "Defining XC Formally", lists the following conditions for global order and consistency:

- ▶ All cores insert their loads, stores, and FENCES into the global order ($<_m$) respecting:
 - ▶ If $X(a) <_p \text{ FENCE} \Rightarrow X(a) <_m \text{ FENCE}$ /* $X = \{L, S\}$ */
 - ▶ If $\text{FENCE} <_p Y(a) \Rightarrow \text{FENCE} <_m Y$ /* $Y = \{L, S\}$ */
 - ▶ If $\text{FENCE} <_p \text{ FENCE} \Rightarrow \text{FENCE} <_m \text{ FENCE}$
- ▶ All cores insert their loads and stores to the same address into the order $<_m$ respecting:
 - ▶ $X(a) <_p Y(a) \Rightarrow X(a) <_m Y(a)$ /* $X = \{L, S\}$ & $Y = \{L', S, S'\}$ */
- ▶ Every load gets its value from the last store before it to the same address
 - ▶ Value of $L(a) = \text{Value of } \text{Max}_{<_m} \{S(a) \mid S(a) <_m L(a) \text{ or } S(a) <_p L(a)\}$ /* Like TSO */

So, once we have this XC then we can define this relax consistency model formally. So, here this says - all the cores insert their loads and stores and fences into the global order respecting these conditions. If there is a load or a store from or to a location A which is coming before a fence instruction the program order then the same load and store will come before this fence instruction in the global memory order. Similarly, if fence is coming before a load or a store

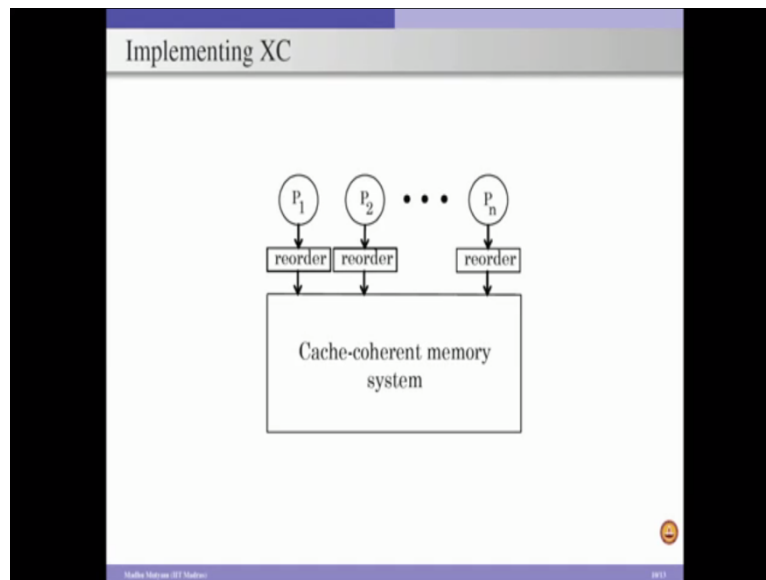
from or to memory location in the program order then this fence should be coming before the corresponding the load and the store instruction in the global order also.

And also if there is a fence instruction which is coming before another fence instruction program order then this fence instruction should be coming first as compared to the other fence instruction in the global memory order also. And also all the cores insert their loads and stores to the same address into the global memory order respecting their program order. For example, if there is a load instruction coming before a store instruction to the same memory location then this should be followed in the same global order also.

And similarly, if there is a store instruction to the same memory location and which is coming before load instruction or a store instruction to the same memory location in the program order that also again we have respect in the global memory order. Here x and y are representing load or store instruction. So, here every load gets its value from last store before into the same memory address. This is the again same as our TSO requirement.

So, here we can get the latest store from any of the other cores or if the same core is actually written value to the same memory location, then it can, the subsequent load can get the value from the write buffer associated with that particular cores. So, that is what is mentioned here. So, this rule is same as the TSO. Effectively this relax consistency model is same as TSO as long as our memory operations to the same memory location. If these operations are 2 different memory locations then we do not have to enforce strict memory ordering between this load and store operation. So, we can reorder the instructions and whenever there is a necessity to enforce an order then we will use a fence instruction for that particular thing.

(Refer Slide Time: 38:00)



So, in order to implement this XC we actually consider reorder buffer associated with each of the cores. And actually reorder buffer concept we already discuss as part of our superscalar processors. And according to superscalar processor design so we can execute instructions in out of order fashion by exploiting the instruction independence, instruction level parallelism and also we can exploit the function unit availability. So that independent instructions can be executed in an out of order fashion, but finally, when we are committing the instructions we have to commit the instructions in the program order.

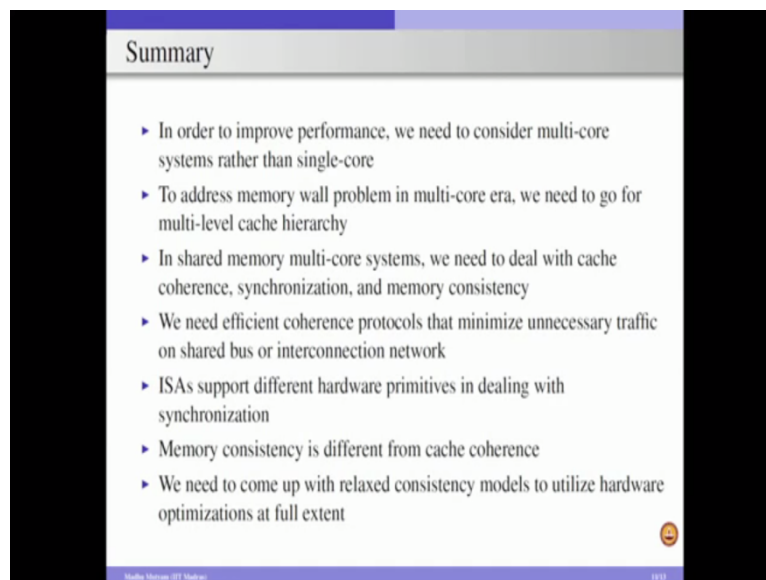
So, that is where we actually make use of this reorder buffer whenever we are issuing an instruction to a function unit. So, we are actually making an entry in the reorder buffer and the instruction can execute in any order with respect to the other instruction of the program, but when we are committing the instructions, we will commit the instructions from the head of this rob. So, that all the instructions will be committed in the program order. So, that is a overall idea of this reorder buffer.

And once we have this reorder buffer mechanism in our hardware, so in order to use this we have to consider a relaxed consistency model, but whereas if we consider sequential consistency are TSO type of thing then as a result we may not exploit this reorder buffers and so as a result performance may not be improved significantly. So, in summary we started with the sequential consistency model, where we cannot use our write buffer or reorder buffer mechanism and we have to strictly enforce ordering between load-stores, stores-load and

store-store and load-load. So, that will degrade the overall performance. So, as a result we will relax some of the constraints, that is ordering between store and load.

So that we can come up with TSO - total store order where we can use write buffers available with our processors. So, that we can improve performance slightly and again in order to further improve the overall performance by exploiting our out of order execution using the reorder buffers then we have relax all these memory orderings wherever there is a possibility. So, that is where we come up with the relax consistency model. Here, in the relax consistency model if there is a requirement to be following the order then we can insert a fence instruction

(Refer Slide Time: 40:44)



Otherwise we can execute instructions in out of order fashion by using this reorder buffer. So with that I am concluding this module and also this concludes this entire course. And in this multicore architecture module so we have discussed the several things and I will just summarise here. So, in order to improve the performance. So, we can go for multicore systems rather than considering single core system so that we can exploit thread level parallelism in our programs.

And once we have multicore systems then all this multicore applications will go to the main memory to get the data and so on but because we already discussed a memory wall problem where the performance of processor is significantly increasing compared to the performance of the memory. So, in order to bridge this memory wall problem we need to consider

multilevel cache hierarchy system. And again when we are dealing with the shared memory systems in multicore systems, then what is going to happen is, there may be an issue with cache coherency, there may be an issue with synchronisation, there may be an issue with the memory consistency.

So, when we have a shared memory multicore system, we have to deal with the cache coherency, we have to deal with the synchronisation, we have to deal with memory consistency and in the last module and this module and the last week and this week we have considered several techniques associated with the cache coherency maintenance, synchronisation and memory consistency models.

As part of cache coherency so we discussed the 3 state cache coherency protocol to maintain the coherency among multiple private caches where they are having the replicated data in their private caches and so on. And we discussed that for bus based systems as well as for the directory based cache coherency protocols and in order to support the synchronisation, so our ISAs instructions set architecture needs to support various hardware primitives.

So that when we write software algorithms to deal with the synchronisation then they can make use of these hardware primitives. And similarly when we deal with the memory consistency, so, we know that the memory consistency is different from cache coherency even the systems may not have the cache coherency but the systems need to have memory consistency and as part of the memory consistency we discussed sequential consistency, we discussed total store order consistency and relaxed memory consistency models.

Among all these 3 memory consistency models, so relaxed consistency model will make use of the hardware components available in the processors to improve the overall performance. And so that is the summary of these multicore architecture modules for this cache coherency synchronisation and memory consistency. So, we covered the material from these 2 books.

Thank you.