Advanced Computer Architecture

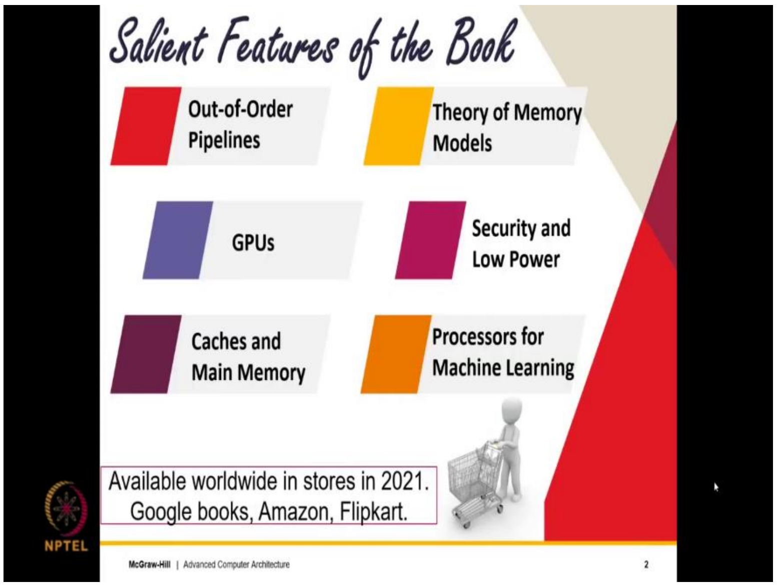
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Chapter - 09
Lecture - 27
Multicore Systems Part - III

(Refer Slide Time: 00:17)

The slides are meant to be used along with the book, Advanced Computer Architecture, Smruti R. Sarangi. https://www.cse.iitd.ac.in/~srsarangi/advbook/

(Refer Slide Time: 00:23)



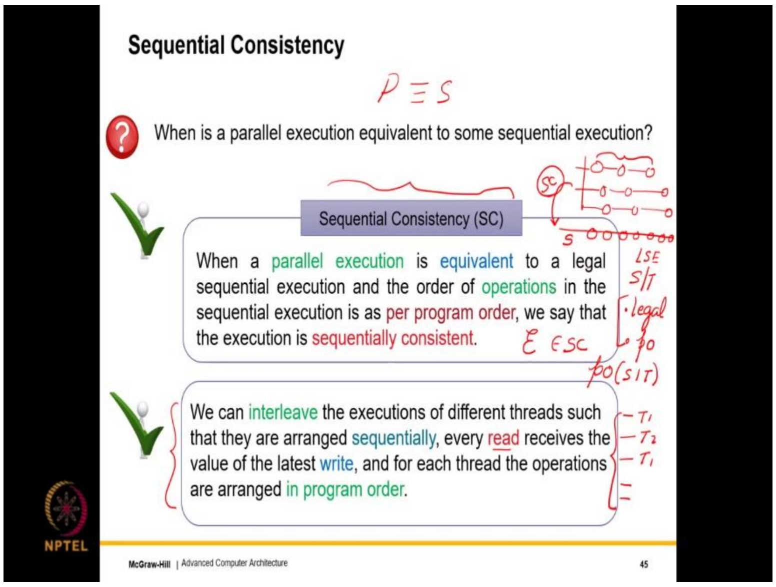
Salient Features of the Book

- Out-of-Order Pipelines
- Theory of Memory Models
- GPUs
- Security and Low Power
- Caches and Main Memory
- Processors for Machine Learning

Available worldwide in stores in 2021. Google books, Amazon, Flipkart.

McGraw-Hill | Advanced Computer Architecture 2

(Refer Slide Time: 00:38)



Sequential Consistency

When is a parallel execution equivalent to some sequential execution?

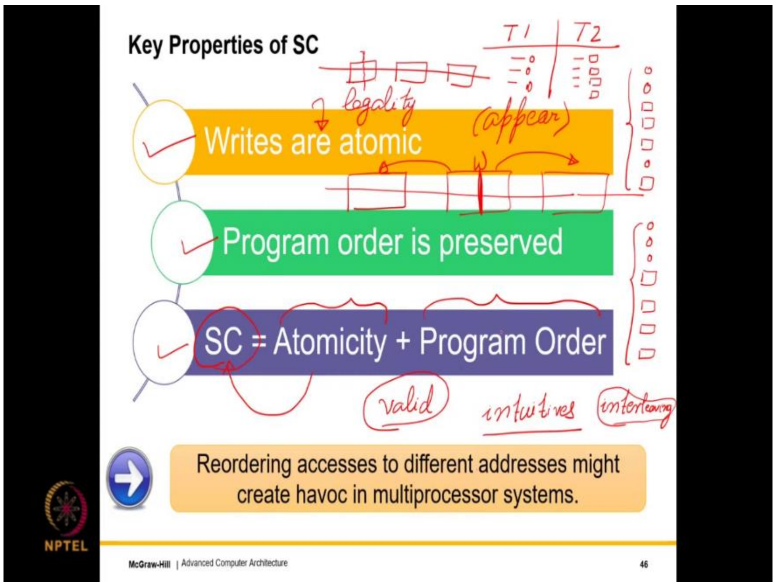
Sequential Consistency (SC)

When a parallel execution is equivalent to a legal sequential execution and the order of operations in the sequential execution is as per program order, we say that the execution is sequentially consistent.

We can interleave the executions of different threads such that they are arranged sequentially, every read receives the value of the latest write, and for each thread the operations are arranged in program order.

McGraw-Hill | Advanced Computer Architecture 45

(Refer Slide Time: 00:43)



Key Properties of SC

Writes are atomic

Program order is preserved

SC = Atomicity + Program Order

Reordering accesses to different addresses might create havoc in multiprocessor systems.

McGraw-Hill | Advanced Computer Architecture 46

T1
T2
legality
(appear)
W
valid
intuitive
interleaving
NPTEL

Let us now start out by extending what we have discussed in the previous slide set. So, there we had established an equivalence between a parallel execution and a sequential execution. So, we had given this criteria, but we are not extended the logic further to actually create a correctness criteria or to say that when a given outcome is valid or not.

In a certain sense this was not extended to a memory model. So, let us create our first memory model called sequential consistency or SC. So, this is fairly old and this was pretty much the first widely accepted memory model to be proposed in the late 70’s by Leslie

Lamport, it is the most intuitive yet the most impractical. So, let me read it out it’s an easy definition. So, when a parallel execution is equivalent to a legal sequential execution and the order of operations in the sequential execution is as per program order we say that the execution is sequentially consistent.

So, what are we doing we have a parallel execution. So, we say that for each thread let us order them in program order. So, this is the typical diagrams that we have been drawing for each circle represents an operation, then we define an equivalent sequential execution where the criteria for equivalence was specified where essentially, we take all of these operations and we arrange them in a linear sequence with a one to one correspondence subject to the fact that the orders remain the same.

So, a further caveat is that yes this needs to be legal sequential execution and second this order needs to be as per the program order of the thread. So, because program order across threads makes no sense, but within a same thread of course, the program order is the order in which we fetch instructions or the order in which we retire instructions. So, since that is in order in program order. We say that the order in the final sequential execution which is also this order in a parallel execution this has to be as per program order and the final sequential execution has to be legal.

If it is possible to convert such a parallel execution into such kind of a sequential execution with these two properties, one it is legal and for every thread the operations appear in program order then we say that the parallel execution is sequentially consistent or it is an SC. So, we can say the formally the execution is an element of SC which means that it is sequentially consistent or in other words it means that it is possible to take a parallel execution and sequentialize it in this manner.

So, we will get into the deeper meaning of this, but we can visualize this differently. So, if you would recall the previous slide set, what we were actually doing is that we were taking operations issued by different threads and we are essentially we were trying to interleave the execution. See we are seeing we take an operation from thread 1 another operation from thread 2 another from thread 1. So, we are try to create a sequential interleaving the same way we interleave our fingers.

So, this was creating a sequential execution. So, this will be considering a sequentially consistent execution if number 1 it is legal which means every read receives the value of the latest write. So, latest and non-latest in a parallel 1 means nothing but in a sequential one the latest write is easy to find we just walk towards the left and we find the latest write to the same address.

And in each thread the operation. So, for each thread. So, let us see if I take a projection with respect to a certain thread, for each thread the operations appear to be in program order. So, we can say that this predicate holds true for every thread that the sequential execution projected on every thread they appear in program order and this is the same this will be true for the parallel execution as well with legality.

So, if we think about it in a deeper way. So, this in a sense captures the intuitiveness of what we have been trying to say. So, what are the key properties of SC? Well the key properties are that we take 2 threads and then derive the two properties. So, let us consider a 2 threaded system we have operations we have operations we just create an interleaving of them.

So, let us say these operations are let us represent them by circles and let us represent these operations by squares sorry by rectangles. So, any of these inter leavings is going to be represent a sequentially consistent outcome as long as we interleave them in a certain manner and the execution is legal and of course, the interleaving is as per program order. So, as you can see a sequentially consistent outcome can still have a sequentially consistent system can still have multiple outcomes that is fine.

But all of them have to be produced via an interleaving where two things are preserved one is legality and other is program order. So, I have written this in a slightly different manner. So, when I say that I can arrange all the operations on a sequential time line essentially, it appears as if the operation appears to execute at some instance between some instant of time between the start and the end.

So, I can say that all the writes in particular are atomic. So, reads of course, do execute at one point in time because they read the value of they read a value and essentially a read is the private operation for a code, but a write is meant to be visible to everybody and all the observers and all the cores agree on a precise time of the write, which in any case is a direct corollary of the fact that I can arrange these operations sequentially.

See if I can arrange these operations sequentially like this, then what I say is that look if there is a write over here all the cores in any case agree on this order of operations. So, I can say that there is a unique instant of time, a precise specific instant of time at which the write appears to complete which means that any point in time or any operation before this point in time does not see the write and all operations after this point of time see the write.

So, whenever an operation completes instantaneously, we say that the operation is atomic. So, of course, as I said a read by definition is a private operation in the sense I read something I keep it nobody else sees it and in a sense whenever I read it at 1 point of time when I get the value, it is it appears to execute at that point in time. So, that is not a big deal, but writes being atomic is important because everybody should agree at a given write was visible to the world at a precise point in time.

So, this does happen in the sequentially consistent memory model because when we arrange everything on a linear time line, a write has to appear to happen at a given point in time. So, the key word that I would like to use is that is the word appear and you may often see this word in the concurrent systems literature where the way to actually understand this is that all the threads agree that look there is one instant of time it does not matter whatever their instant is, but at least the instant of time exists.

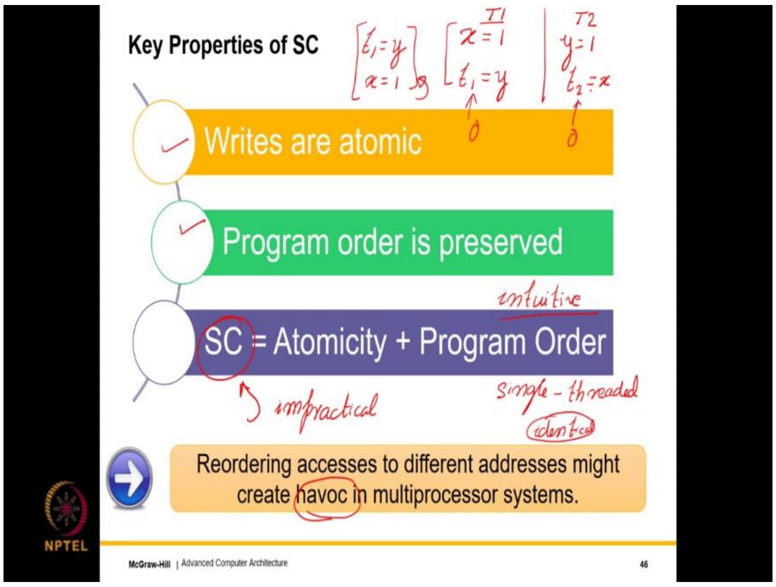
So, it is important we do not care what instant of time it was as long as all of us agree that look there was a point in time where the write executed instantaneously and this and all the reads after that will get the value of this write after that to the same address. So, this is legality, a legality in this case is transferring to the fact that writes are atomic. The second is program order is preserved in the sense in the final sequential execution which is equivalent to the parallel execution.

For each thread the operations appear in program order which is anyway the order in which we were interleaving in our discussion in a previous slide set. So, this is intuitive this is intuitive. So, all that sequential consistency does is that it takes two of our intuitive notions of legality and program order. Legality gets converted to atomicity and it combines atomicity and program order and lo and behold we have our first consistency model called sequential consistency, which given a parallel program will give us a set of valid outcomes which to our mind are intuitive.

In the sense, it’s all a question of taking the instructions are different threads and just interleaving them. So, the key word here is interleaving I would request all of you to thoroughly understand the meaning of this pick up a dictionary and the meaning of this is that I maintain the order of each thread and I just interleave them the way I have done over here.

So, this is a very very important correctness criterion and so, let us say that this is a goal standard of all consistency models where this is clearly the most intuitive.

(Refer Slide Time: 11:26)



Key Properties of SC

$\begin{bmatrix} t_1 = y \\ x = 1 \end{bmatrix}$ $\begin{bmatrix} T1 \\ x = 1 \\ t_1 = y \end{bmatrix}$ $\begin{matrix} T2 \\ y = 1 \\ t_2 = x \end{matrix}$

0 0

- Writes are atomic ✓
- Program order is preserved ✓
- SC = Atomicity + Program Order

intuitive

impractical

single-threaded

identical

Reordering accesses to different addresses might create havoc in multiprocessor systems.

McGraw-Hill | Advanced Computer Architecture 46

So, in terms of intuition it is very very high and in terms of common sense sequential consistency appears to be the way to go because it preserves it gives us a very simple property that take a parallel execution. As long as you can convert it to the sequential execution with these two properties guaranteed which appear to our innate sense of reason.

We can say the execution is correct we can reason about it anything outside that we will say is an non SC. Now here is a sad part this is why I say it is impractical because almost all the optimizations that we have proposed break SC. So, let them be or let they be out of order processors MSHR’s you name it all the cache NoC core optimizations that we have proposed the either break program order because loads are issued early.

Some of them particularly the ones that we will discuss in this chapter as well as methods that use the NoC will break atomicity. So, that is the reason there is a problem. So, here again I would like to see that if a single threaded program is executing on a multicore processor the results will be identical to that executing on a unicore processor for single threaded there is no issue.

The issue arises for multithreaded programs that share variables because if we reorder accesses to different addresses. So recall this example that we had given in the previous slide set where this is what we had x = 1, y = 1, we read in y and we read in x.

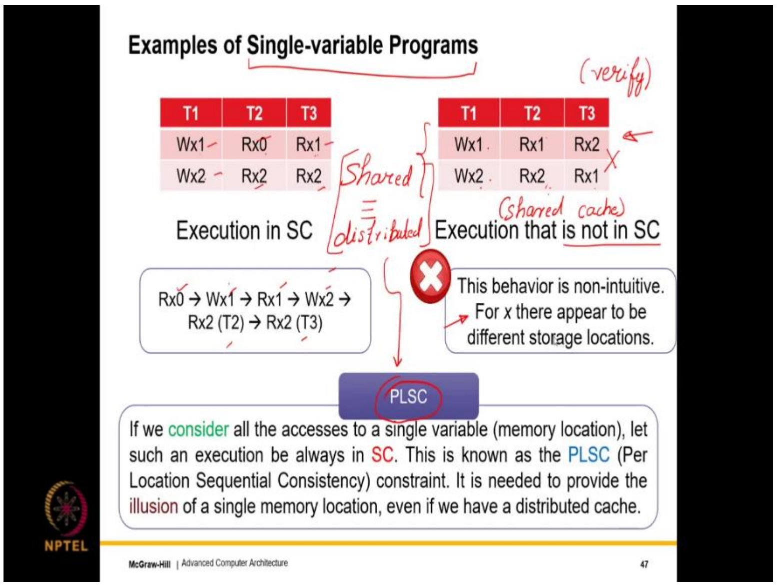
So, from the point of view of this program, we can reorder the accesses in the sense we can send y first and send x later. As far as we are concerned this program is correct compilers can also reorder the processor definitely will reorder and it will not change the final state or the outcome of the program because these are all after all accesses to different variables, but as we see this will create a create havoc in multiprocessor systems because this harmless reordering that we have been doing up till now and that too very very carelessly without any concern for what would happen in a multiprocessor system.

This will actually create a lot of issues in the sense it will directly breaks sequential consistency because if we reorder these two instructions send the loads early we can read both T 1 and T 2 = 0 and 0 and as you can see this execution is clearly not sequentially consistent.

So, as we can see this execution over here is clearly not sequentially consistent. If we allow reordering because program order is not being preserved, but sometimes on performance we break it and this is where it appear to be a perfectly great thing to do when we were discussing an out of order pipeline, but when we consider multiple cores we see that the intuitive correctness notion of sequential consistency cannot be guaranteed. So, the main idea is you do something good somewhere something bad happens somewhere else.

So, we did great things in the memory system and pipeline something bad happened when we start writing multithreaded code.

(Refer Slide Time: 15:42)



Examples of Single-variable Programs

T1	T2	T3
Wx1	Rx0	Rx1
Wx2	Rx2	Rx2

Execution in SC

Rx0 → Wx1 → Rx1 → Wx2 → Rx2 (T2) → Rx2 (T3)

T1	T2	T3
Wx1	Rx1	Rx2
Wx2	Rx2	Rx1

Execution that is not in SC

This behavior is non-intuitive. For *x* there appear to be different storage locations.

PLSC

If we consider all the accesses to a single variable (memory location), let such an execution be always in SC. This is known as the PLSC (Per Location Sequential Consistency) constraint. It is needed to provide the illusion of a single memory location, even if we have a distributed cache.

McGraw-Hill | Advanced Computer Architecture 47

So, let us look at a few examples to understand and also to further to have a slightly deeper discussion. So, let us first take a look at single variable programs. So, in single variable programs we will not have this reordering because they were two different variables. So, a single variable programs let us look at this example where I write 1 then write 2.

So, thread 1 read 0 first, then 3 reads 1 first and then 2. This execution is in SC why because as you can see we can order them a sequential ordering exist Rx0 ok Wx1 Rx1 Wx2. So, at no point of time I have broken program order yet Rx2 a thread 2 Rx2 a thread 3. So, this is a legal ordering it's an equivalent ordering and we have not compromised on program order.

But let us take a look at this Wx1, Wx2. So, thread 1 writes 1 and 2 to x, but thread 2 reads 1 first then 2, thread 3 reads that in the reverse order 2 first and then 1. So, little bit of a homework kindly verify this execution is not in SC it is not possible to come up with an equivalent sequential execution for this parallel program which is legal and preserves program order.

And regardless of whatever you do you will not be able to come up with an equivalent sequential execution. So, this execution is not in SC. Furthermore, if you look at it this behavior is non intuitive because for the variable x they are appear to be different storage locations. For the same variable x, it appears that there are different storage locations in the sense. So, here again the observer is at the level of the thread. So, you are essentially looking at it from the point of your software.

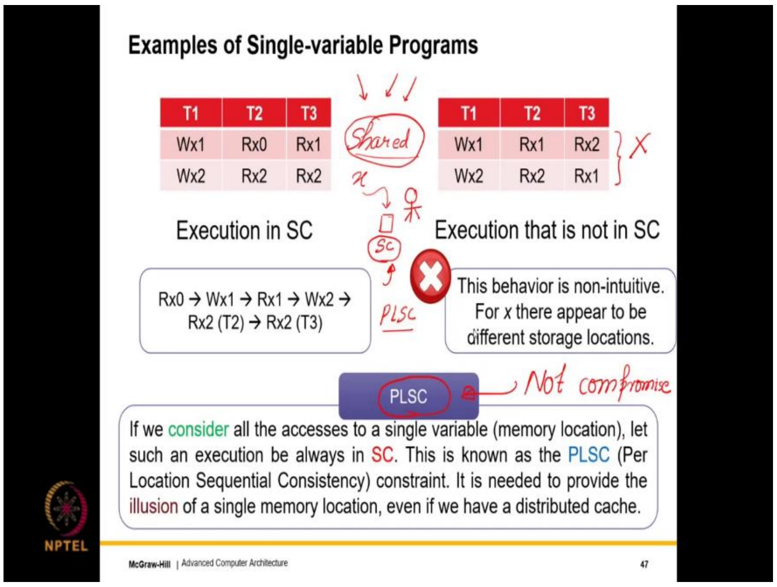
So, it appears that for one location for the first thread 2 it sees the correct order of writes, but for thread 3 it sees the writes in the reverse order. So, somehow the update though x is getting garbled up and that is not really what you would expect to see in a shared cache that has a single location for x because any observer who is sitting on x would essentially see one return first and then 2, thread 2 would agree with this, thread 3 will not agree hence thread threes execution is not valid.

So, what we see is that now we are kind of looking at the finer points that for a single variable program or let us see if I consider all the accesses to a single variable in a multi variable program if SC is not guaranteed, it appears as if we do not have a single storage location for x and this is something that we explicitly wanted to ensure when we set that for an outsider a shared cache should appear identical to a distributed cache write. So, this is something that we wanted to ensure, but you can clearly see that if we have an execution of this type this is not being ensured.

Now, is there a connection between a shared is equal to distributed and SC? Yes, there is and this connection is very interesting this is called PLSC which I will define now. So, if we consider all the accesses to a single variable or a memory location which is x in this case, let us such an execution be always in SC. So this correctness criterion is known as the PLSC constraint PLSC stands for Per Location Sequential Consistency.

It is needed to provide the illusion of a single memory location even if we have a distributed cache. So, what is it that I am doing? What I am essentially doing over here is I am saying that look, consider a shared cache which is slow and inefficient, but for the variable x it has a single location place our friend who is an observer over there.

(Refer Slide Time: 20:25)



Examples of Single-variable Programs

T1	T2	T3
Wx1	Rx0	Rx1
Wx2	Rx2	Rx2

Execution in SC

Rx0 → Wx1 → Rx1 → Wx2 → Rx2 (T2) → Rx2 (T3)

T1	T2	T3
Wx1	Rx1	Rx2
Wx2	Rx2	Rx1

Execution that is not in SC

This behavior is non-intuitive. For *x* there appear to be different storage locations.

PLSC

If we consider all the accesses to a single variable (memory location), let such an execution be always in SC. This is known as the PLSC (Per Location Sequential Consistency) constraint. It is needed to provide the illusion of a single memory location, even if we have a distributed cache.

McGraw-Hill | Advanced Computer Architecture 47

So, our friend will observe at this location a sequentially consistent execution primarily because your friend will basically see requests coming in and then it will be legal. So, definitely this kind of an execution will not be seen by an observer and even if let us say we are observing this cache from outside and only tracking accesses to a single location. So, we will see that look this is the shared cache we are pumping in accesses by different threads and let all of them be at a same location, let us filter them out that way.

So, if it is not in SC, it will appear as if look maybe we do not have a single physical storage for x and something really strange is happening in there, but if we, but if we genuinely have a single location for x, it has to be sequentially consistent unless accesses are reordered. So, let us assume for a single location is never the case then if they are not reordered then the only thing we need to guarantee is atomicity or legality and it will be legal because it is a single location.

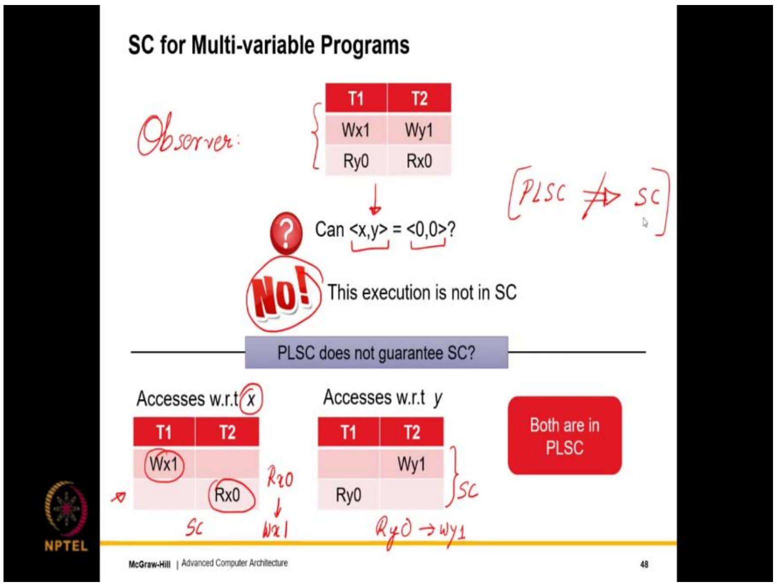
So, whenever I read I am bound to get the value of the latest write. So, for a single location in a shared cache if I am observing write I will see it to be sequentially consistent. Hence, the PLSC criterion is necessary and sufficient of course, this is an informal proof, but there are more formal reasoning in the book, but what you need to take away from this discussion is that the PLSC criterion is both necessary as well as sufficient to ensure that a shared cache appears to an outsider.

The same as a distributed cache or rather putting in the other way the distributed cache appears to an outsider, the same as the shared cache because per location all the accesses are sequentially consistent. So, it appears that we have a single location whatever the observer at that location will see, an observer outside the cache will also see that look requests are going in we have been serviced in program order and second every read is getting the value of the latest write.

So, the PLSC criterion for us is important. So, we will not compromise on this, but the PLSC is not the same as SC, PLSC is only per location and there is a deeper connection between them we will discuss this in sometime, but the key thing is this is something that we will not compromise on. So, SC of course, form SC was being the general thing for multiple locations and there we found that look a lot of our optimizations are not guaranteeing SC fine fair enough.

But per location sequential consistency let us strictly disallow this behavior and let us say that we will always have a such that a shared and distributed cache appear identical to outsiders, how to ensure it in a separate question, but let us at least have it as one of our objectives.

(Refer Slide Time: 24:06)


SC for Multi-variable Programs

Observer:

T1	T2
Wx1	Wy1
Ry0	Rx0

Can <x,y> = <0,0>?

No! This execution is not in SC

[PLSC $\not\Rightarrow$ SC]

PLSC does not guarantee SC?

Accesses w.r.t x

T1	T2
Wx1	
	Rx0

SC

Rx0 → Wx1

Accesses w.r.t y

T1	T2
	Wy1
Ry0	

SC

Ry0 → Wy1

Both are in PLSC

McGraw-Hill | Advanced Computer Architecture 48

So, we have discussed the single variable case let us discuss the multi variable case. So, let us now consider this execution which is our running example that we have been showing. So, there are two threads T1 and T2, T1 writes 1 = x, T2 writes 1 = y and then we read. So, we read y T1 reads y it reads it to be 0 because the load is issued early likewise T2 reads it reads x = 0.

So, can x, y = 0 0? Well, in a modern out of order processor yes as per SC no. So, as per. So, this order is clearly not sequentially consistent this execution, but let us look at this execution from the point of view of each location. So, as I said in such theoretical discussion which is of course, very important the reason it is important is basically because without a little bit of theory.

So, when do you need theory, it’s a broad it is a philosophical thing, but when we actually need a theory. So, look at the night sky. So, previously when people did not understand how really planets and stars move they say that maybe you have Orion or the hunter and it is hunting a scorpion or a lion or a bull. So, they created this notion of constellations right and then they said where they gave a certain connotation to the pole star and so on.

But when they started understanding things better they found that everything could be explained with Newton’s laws or Einstein’s laws and pretty much that could explain most part of most of the movements of the night sky. Of course, this theory is not complete in the sense this do have dark matter and dark energy and so on, but that sudden done the objective of having a theory is to somehow explain experimental results or engineering decisions and choice.

So, in this case, given the fact that we are looking at a very complex engineering system unless we have some specification we will get lost. So, we have already discussed that in our complex engineering system this execution is valid this is not in SC, but we still need a little bit of a theory to understand this better that is why we created the notion of observers and this is very important because every observer has a point of view.

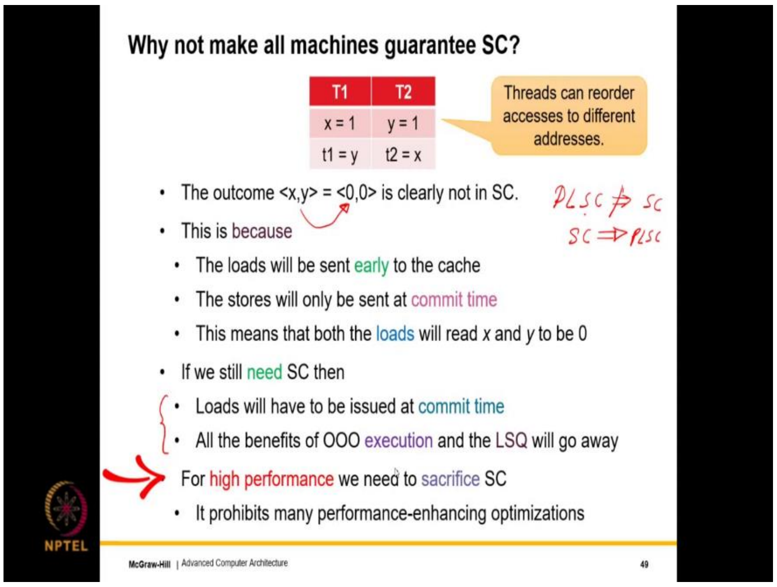
So, we can always attach an observer per location with observer only watches what happens for a specific location write. So, we can have an observer for x. So, that would observe a write to x by thread 1 and a read to x 1 from thread 0. As we have discussed in 1 of the assumptions in the previous slide set the absolute times do not matter. So, in any distributor or concurrent system for such models the absolute time the wall clock time does not matter.

What matters is diagrams like this? This is clearly sequentially consistent because we have a sequential order which is Rx0 and Wx1 which means that x was initialized to 0 and read 0 first and then you wrote to 1. Similarly, if I have another observer with respect to y again this is the execution that the observer sees this is again sequentially consistent well why not Ry0 Wy1.

So, we do have per location sequential consistency because for both x and y they are per location sequentially consistent. So, now, we have an interesting example where overall we do not have SC, but we have PLSC, but we do not have SC. So, this means that even if we have per location sequential consistency, it does not imply that the overall execution is sequentially consistent.

So, what does this mean for us? This means that even if we have a system with a lot of distributed caches where PLSC does hold in the sense the distributed caches appear as a monolithic shared cache for outsiders it still does not guarantee that the overall system provide sequential consistency and this is being captured by this example where you can clearly see that PLSC holds, but SC does not. So, PLSC does not imply SC.

(Refer Slide Time: 29:25)



Why not make all machines guarantee SC?

T1	T2
x = 1	y = 1
t1 = y	t2 = x

Threads can reorder accesses to different addresses.

- The outcome <x,y> = <0,0> is clearly not in SC.
- This is because
  - The loads will be sent early to the cache
  - The stores will only be sent at commit time
  - This means that both the loads will read *x* and *y* to be 0
- If we still need SC then
  - Loads will have to be issued at commit time
  - All the benefits of OOO execution and the LSQ will go away
- For high performance we need to sacrifice SC
  - It prohibits many performance-enhancing optimizations

$PLSC \not\Rightarrow SC$

$SC \Rightarrow PLSC$

McGraw-Hill | Advanced Computer Architecture 49

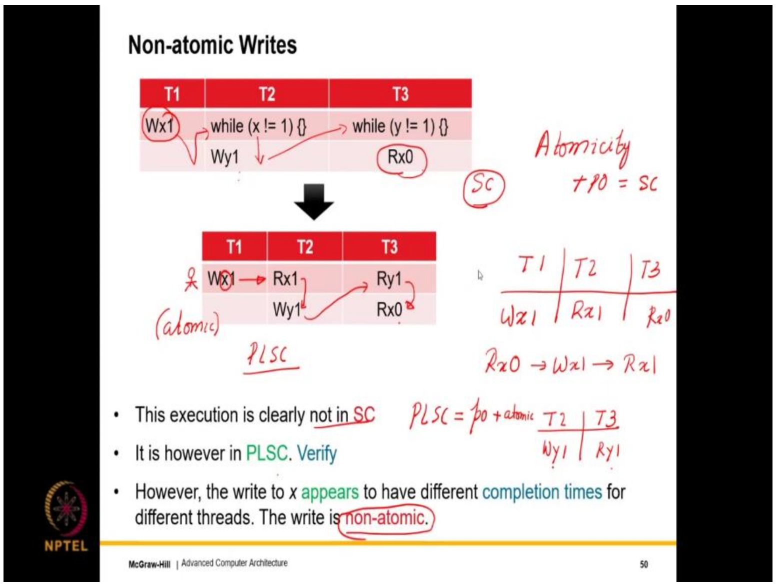
So, then let us take a look at this particular execution once again. Say in this case we are assigning them to the temporary T1 and T2 which are registers. So, this outcome as we said is clearly not in SC and so, as we have seen this is not in SC, but its there in PLSC. So, important point needs to be said here that look PLSC does not guarantee SC, but definitely SC guarantees PLSC because if it is overall sequentially consistent, it is going to be per location sequentially consistent as well because program order is guaranteed legality is guaranteed that will be there per location.

Say SC guarantees PLSC. So, PLSC does not guarantee SC. So, we can say that PLSC is weaker and SC is stronger. Now let us come back to the discussion, the reason why PLSC guarantees do not translate to SC guarantees is basically because the loads will be sent early to the cache the stores will happen at commit time. So, the load will in the sense bypass the stores, if we still need sequential consistency loads will have to be issued at commit time which means we will not have out of order execution.

All the benefits of out of order execution will go away. All the benefits of having an LSQ will go away. So, pretty much everything that we have learnt in a previous chapter on performance enhancement will be rendered to a null. So, for high performance we have to let go of SC we will have to sacrifice SC we may not sacrifice PLSC.

So, we will not actually sacrifice PLSC, but we have to sacrifice SC in interest of high performance otherwise many of these performance enhancing optimizations will not be possible.

(Refer Slide Time: 31:39)



Non-atomic Writes

T1	T2	T3
Wx1	while (x != 1) {}	while (y != 1) {}
	Wy1	Rx0

T1	T2	T3
Wx1	Rx1	Ry1
	Wy1	Rx0

- This execution is clearly not in SC
- It is however in PLSC. Verify
- However, the write to *x* appears to have different completion times for different threads. The write is non-atomic.

Atomicity + PO = SC
SC
(atomic)
PLSC
T1 | T2 | T3
Wx1 | Rx1 | Rx0
Rx0 → Wx1 → Rx1
PLSC = po + atomic
T2 | T3
Wy1 | Ry1

So, now, let us look at some of the ways in which we can break SC because SC anyway as we have said is meant to be broken. So, SC is an essentially atomicity of writes plus program order = SC. So, program order as we have seen is clearly being violated over here because our out of order pipeline let us look at atomicity.

So, in atomicity what is happening? Let us take a look at this execution where we write 1 = x as long as the x ≠ 1 we keep looping in a while loop. So, since we write 1 = x ultimately this while loop exits, then we write 1 = y. Again another while loops on y until it is 1 finally, it exits at this point what would you expect you would see a clear path of causality and then you would see a clear path like this you would expect to read x = 1, but let us say you need x = 0.

So, then what I can do is I can create an execution like this well let me assume the first time I access the while loop I read x or y = 1. So, I never iterate. So, then they should look like this. I write, I read, then the next is a program order relation which has to hold because unless I exit this I cannot execute this and then again I write, I read in this case you expect to read 1, but you read 0.

So, what is happening? This execution is not in SC that should be obvious. So, if it’s not obvious you are try to put it into a sequential order and you will find that it is not put able because there is a clear path and legality is being violated this should read one which is actually reading 0.

So, it is a violation of legality it’s not in SC. It is; however, in PLSC kindly verifying it is very easy. So, what you do is for PLSC you just create an execution for a single variable. So, its Wx1, Rx1 and Rx 0. Of course, it’s in PLSC because I can order them sequentially Rx0, Wx1, Rx1 fine what about for y for y? I can do the same.

So, for y what I can do is that I can have Wy1 Ry1 and this is also in PLSC because first this instruction executes and then this. So, this is in PLSC its not in SC one more of those executions which is in PLSC, but not in SC, but there is a fun point the fun point is that we are violating SC in a different way in the previous slide we are violating the program order constraint in this case you are violating atomicity constraint.

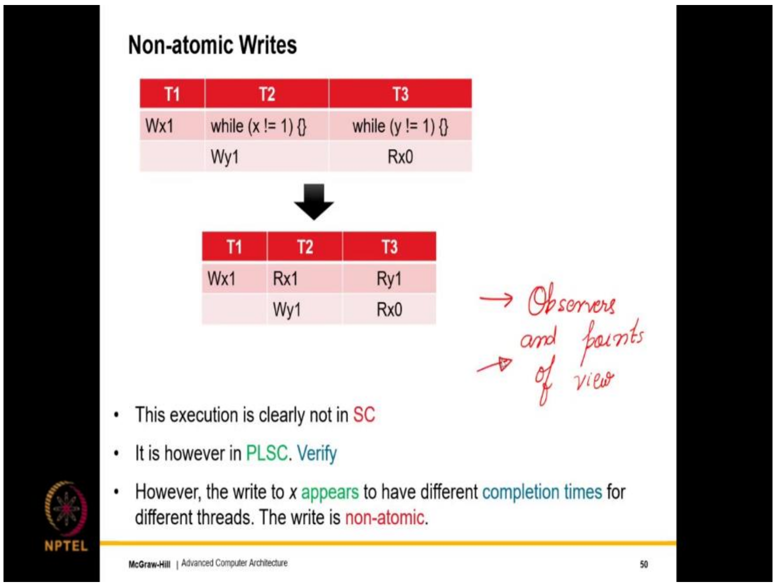
So, it appears that the write to x, it appears the write to x is happening at two different times. So, thread 2 is reading the value of x the updated value of x, thread 3 is reading the old value. So, since a write appears to be happening at two different points of time it is non atomic. So, then I mean this complicates or life significantly.

So, you will say that look you're from the point of view of the observer who is only observing the accesses to the variable x, we have PLSC what does PLSC mean? It means per location sequential consistency which is again it means that all the accesses to that location by a thread are in program order fine. I have no issues with that, but it also means that writes are atomic yes, but this is where the point of view matters.

So, the writes are atomic with respect to an observer who is only observing the accesses to variable x, but the writes are not atomic with respect to an observer who is observing the accesses to all the variables. So, the point of view of both the observers is different. The point of view with respect to the observer who's only observing updates to a given variable accesses to a given variable from her point it is atomic, but when we consider accesses to other variables like y in this case, then the writes to x do not seem to be atomic.

That is why we have PLSC which is per location sequential consistency, but we do not have sequential consistency when we consider the accesses to y as well. So, this is why memory consistency models are slightly tricky because what really matters are observers and their point of view.

(Refer Slide Time: 37:20)



Non-atomic Writes

T1	T2	T3
Wx1	while (x != 1) {}	while (y != 1) {}
	Wy1	Rx0

T1	T2	T3
Wx1	Rx1	Ry1
	Wy1	Rx0

→ Observers and points → of view

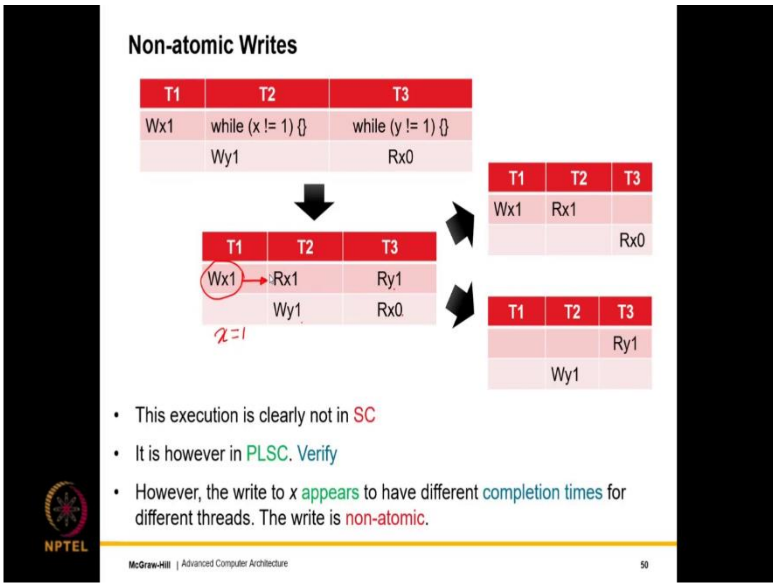
- This execution is clearly not in SC
- It is however in PLSC. Verify
- However, the write to *x* appears to have different completion times for different threads. The write is non-atomic.

McGraw-Hill | Advanced Computer Architecture 50

So, what did we learn? We learnt that observers and their points of view are very very important and crucial. So, in this case PLSC does hold, but SC does not hold because writes are not atomic, you can come back to me and tell me that look how did PLSC hold if writes are not atomic I will tell you that look from the point of view of an observer who is only looking at accesses to x writes are atomic, but when you consider multiple variables writes to x are not atomic.

So, it is important that you understand this part because this is very very crucial in analysis of such systems which is the notion of observers and their points of view.

(Refer Slide Time: 38:11)



Non-atomic Writes

T1	T2	T3
Wx1	while (x != 1) {}	while (y != 1) {}
	Wy1	Rx0

T1	T2	T3
Wx1 $\rightarrow$	Rx1	Ry1
	Wy1	Rx0

$x=1$

T1	T2	T3
Wx1	Rx1	
		Rx0

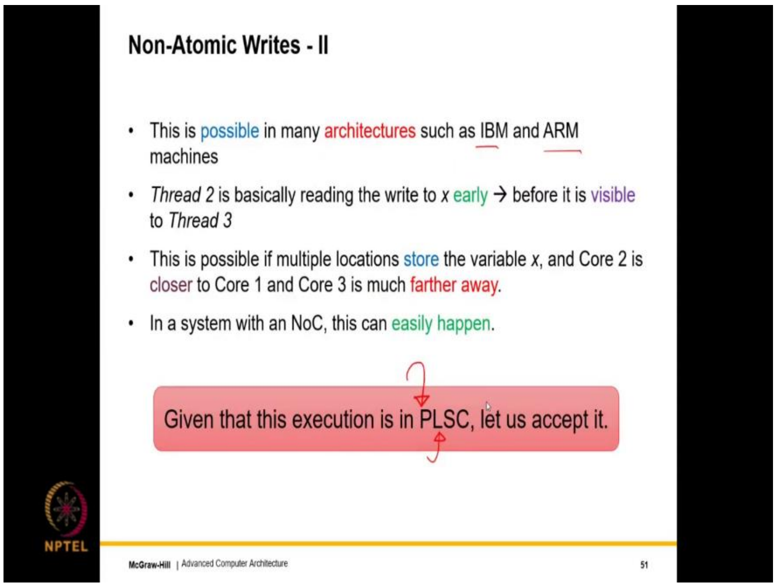
T1	T2	T3
		Ry1
	Wy1	

- This execution is clearly not in SC
- It is however in PLSC. Verify
- However, the write to *x* appears to have different completion times for different threads. The write is non-atomic.

McGraw-Hill | Advanced Computer Architecture 50

So, we anyway we did a little bit of analysis. So, I have that in the slides where essentially what I do is, I look at the points of view of observers only from only for specific variables so you can see that these are clearly in SC.

(Refer Slide Time: 38:32)



Non-Atomic Writes - II

- This is possible in many architectures such as IBM and ARM machines
- *Thread 2* is basically reading the write to *x* early → before it is visible to *Thread 3*
- This is possible if multiple locations store the variable *x*, and Core 2 is closer to Core 1 and Core 3 is much farther away.
- In a system with an NoC, this can easily happen.

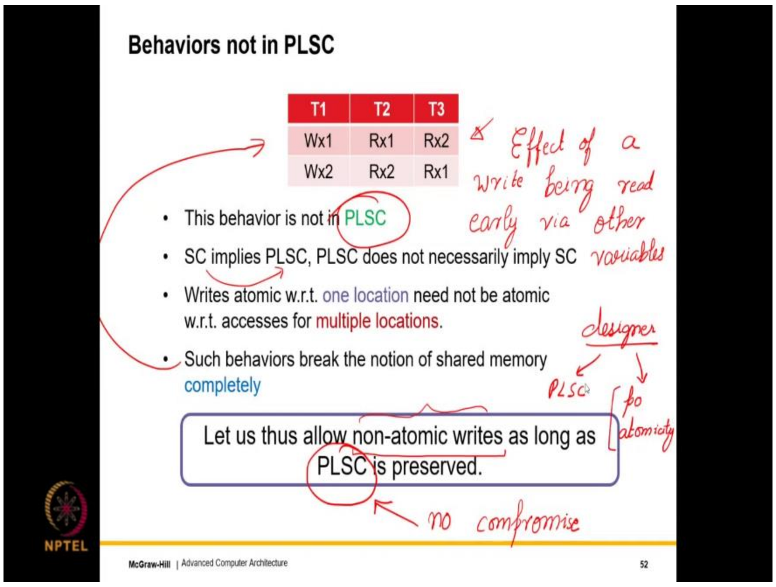
Given that this execution is in PLSC, let us accept it.

McGraw-Hill | Advanced Computer Architecture 51

So, PLSC holds. Non atomic writes well there are many popular architectures such as IBM machines and arm machines where writes are not atomic again for performance reasons. What is happening here? Thread 2 is basically reading the write to x early before it is visible to thread 3. This is possible if multiple locations store the variable x and core 2 is somehow closer to core 1 and core 3 is far away. So, core 2 sees it first sees it first and core 3 sees it later.

In any system with an NoC this can easily happen, but nevertheless as far as we are concerned the execution is in PLSC. So, let us accept it if we are accepting non atomic writes and IBM and arms memory models do accept it. So, you run something on your phone you might very well see this execution. So, this means that PLSC will hold as I said PLSC always holds if we want distributed cache to act like a shared cache, but atomicity might or might not hold. It depends on the memory model. So, in Intel machines atomicity holds, but in IBM and arm machines it does not hold.

(Refer Slide Time: 39:52)



Behaviors not in PLSC

T1	T2	T3
Wx1	Rx1	Rx2
Wx2	Rx2	Rx1

- This behavior is not in PLSC
- SC implies PLSC, PLSC does not necessarily imply SC
- Writes atomic w.r.t. one location need not be atomic w.r.t. accesses for multiple locations.
- Such behaviors break the notion of shared memory completely

Let us thus allow non-atomic writes as long as PLSC is preserved.

Effect of a write being read early via other variables
designer
PLSC
to atomicity
no compromise
NPTEL
McGraw-Hill | Advanced Computer Architecture
52

So, let us now look at a behavior which is not in PLSC. So, this behavior would be that I write 1 and 2 to x and I read them in different orders. So, we have seen this execution before also and this execution is not in SC since there is single location it's not in PLSC. You have also seen SC implies PLSC, PLSC does not necessarily imply SC and we have also seen that writes that are atomic with respect to one location may not be atomic with respect to multiple locations when you are considering multi variable programs.

So, such kind of behaviors break the notion of shared memory completely which behaviors these is this kind of behavior. So, the behavior of this because PLSC does not hold this behaviors kind of breaks the notion of shared memory because for any external what will it observe? It will observe the look I wrote 1 first 2 x and then 2, then better everybody better observe that if T 3 is not agreeing then it appears that x is not stored in a single location and then essentially this does not make any sense.

So, we will allow non atomic writes let us thus allow non atomic writes of this kind let us allow non atomic writes as long as PLSC holds which means that if I am only concerned about the accesses to a single variable they should be sequentially consistent and that will give me a lot of correctness properties of distributed caches make them similar to shared cache.

But of course, when I am considering multiple locations then what will happen is that writes may appear to be non atomic in the sense what is happening here I am writing 1 to x. I am writing 1 to x this write is quickly visible to thread 2, but it is taking much longer to propagate to thread 3. Fair enough, but how do, but how do we actually discover this fact? We discover this fact via accesses to other variables right which in this case is y.

But let us say that we do not actually consider this and then let us say that at some point of time we actually read 1 in thread 3 all subsequent accesses will also give us 1 because the write has ultimately reach thread 3. So, the main reason for this behavior is that look messages and accesses do take time in a realistic system, but if I were only to consider accesses to one variable I will still see that writes are atomic because we will never have an execution where the writes are actually not atomic.

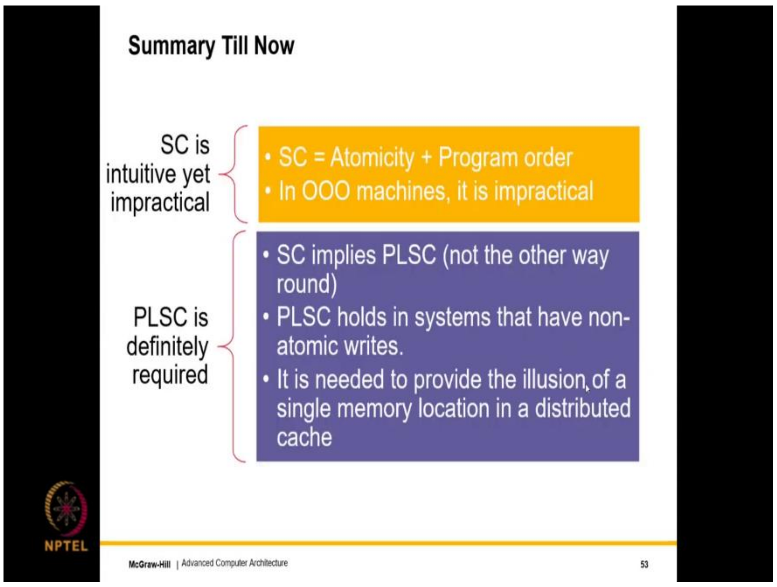
So, we will never see executions like this where writes are not atomic, but what can happen is the moment I consider accesses to other variables, then I can see the effect of a write or an effective write being read early via other variables. So, what is the crux here that I am seeing the effect of a write being read early via other variables.

So, basically if I do not consider other variables I will not see the effect of a write being read early and that is why in this case PLSC holds. So, the because in PLSC I do not consider other variables, but if PLSC is generally not holding as in this case, then clearly all observers will say that look clearly we are not implementing a correct memory system. So, let us thus agree to allow non atomic writes as long as PLSC is preserved.

I mean subject to the memory model some memory models might allow might not allow, but let us say that we will not have objection to it or we will not say that a memory model is invalid, if it allows non atomic writes, but PLSC we will never compromise. So, we have decided to draw a red line. So, PLSC there is no compromise program order and atomicity well we will leave it to the processor designers.

So, what we are saying is that look you are a processor designer, here are certain red lines thou shall not cross one red line is PLSC you will have to obey and observe other things like program order and atomicity that you can break. So, that it all depends on what optimizations you are implementing that you can possibly break, but PLSC do not touch.

(Refer Slide Time: 45:41)



Summary Till Now

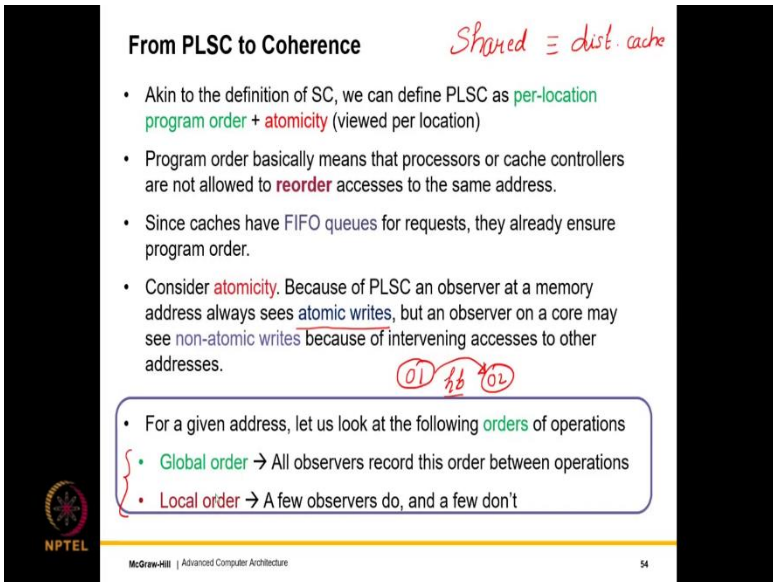
SC is intuitive yet impractical
- SC = Atomicity + Program order
- In OOO machines, it is impractical

PLSC is definitely required
- SC implies PLSC (not the other way round)
- PLSC holds in systems that have non-atomic writes.
- It is needed to provide the illusion of a single memory location in a distributed cache

McGraw-Hill | Advanced Computer Architecture 53

So, what is the summary till now? SC is atomicity plus program order. SC implies PLSC, PLSC does not necessarily employ SC. PLSC does hold in systems that have non atomic writes that have atomic writes PLSC is needed to provide the illusion of a single memory location in a distributed cache. So, PLSC no compromise the rest possible compromise.

(Refer Slide Time: 46:09)



From PLSC to Coherence

Shared ≡ dist. cache

- Akin to the definition of SC, we can define PLSC as per-location program order + atomicity (viewed per location)
- Program order basically means that processors or cache controllers are not allowed to **reorder** accesses to the same address.
- Since caches have FIFO queues for requests, they already ensure program order.
- Consider atomicity. Because of PLSC an observer at a memory address always sees atomic writes, but an observer on a core may see non-atomic writes because of intervening accesses to other addresses.

O1 hb O2

- For a given address, let us look at the following orders of operations
  - Global order → All observers record this order between operations
  - Local order → A few observers do, and a few don't

McGraw-Hill | Advanced Computer Architecture 54

So, how do we move from PLSC to coherence? So, PLSC to a large extent has taken us around 95% in the sense, it has told us that look I have a shared cache where for every variable I have a single physical location and then I have a distributed cache where for every variable I have different physical locations, for an outsider they will look identical if I have PLSC which again can have broken down into per location program order in atomicity.

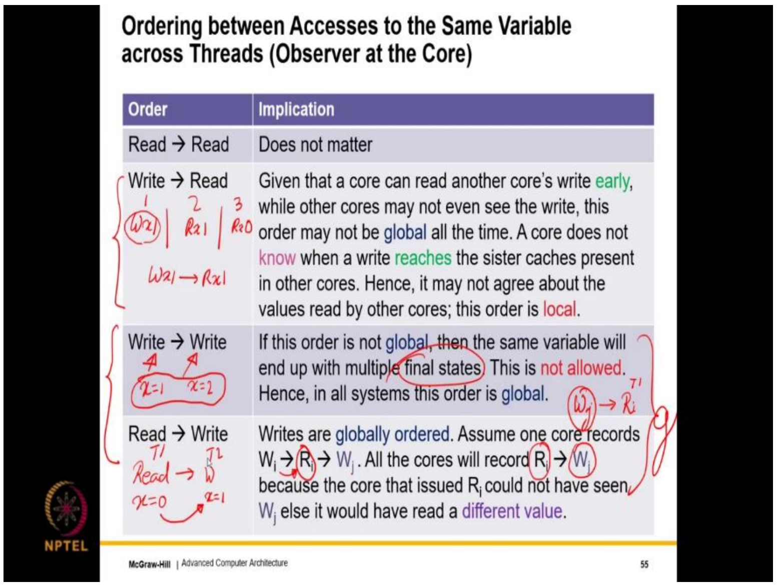
So, program order would basically mean that processors or cache controllers are not allowed to reorder accesses to the same address. So, that they would anyway do because caches are FIFO queues for request. So, they already ensure program order in that sense and also our pipeline needs to be modified such that we will not reorder accesses to the same address but that we anyway do not do our load store queue ensure that we do not do it. So, in the next is atomicity.

So, because of PLSC and observer at a memory address always sees atomic writes only for accesses to that address so, but an observer on a different core as we are seen may see non atomic writes because of intervening accesses to other addresses. So, you take away other addresses it should then become atomic.

So, for a given address we have let us define two orders. A global order is when all observers record this order between operations. So, let us see we have one operation O 1 and one O 2. So, if you have different observers and let us say different cores or in different sister caches of a distributed cache everybody will record that look O 1 happen first O 2 happen later.

So, all of them will record if all of them record and agree. So, it does not matter if O 1 happened before O 2 once again before or 10000 seconds before that does not matter as long as they record such happens before relationship we say that this order is global. If let us say a few do and a few do not, we say that this order is not global it is local.

(Refer Slide Time: 48:36)



Ordering between Accesses to the Same Variable across Threads (Observer at the Core)

Order	Implication
Read → Read	Does not matter
Write → Read	Given that a core can read another core's write early, while other cores may not even see the write, this order may not be global all the time. A core does not know when a write reaches the sister caches present in other cores. Hence, it may not agree about the values read by other cores; this order is local.
Write → Write	If this order is not global, then the same variable will end up with multiple final states. This is not allowed. Hence, in all systems this order is global.
Read → Write	Writes are globally ordered. Assume one core records $W_i \rightarrow R_i \rightarrow W_j$. All the cores will record $R_i \rightarrow W_j$ because the core that issued R_i could not have seen W_j else it would have read a different value.

McGraw-Hill | Advanced Computer Architecture 55

So, let us now look at the ordering between accesses to the same variable across threads where the observer is at a core. So, let us consider two reads to the same address. So, there order does not matter because anyway they will read the same if there are no intervening writes. So, we do not really care consider a write and a read. So, what can happen is that given that a core can read another cores write early while other cores may not even see the write this order may not be global all the time correct.

So, what will happen is there a core does not know when a write reaches the sister caches the rest of the sister caches present in other cores. So, I might say that look I go a write and then I read. So, I might record that order, but the others might not really agree hence it may hence another sister cache may not agree about the values read by other cores write one core may not agree about the values write by other cores.

So, this order is local let me explain with an example. So, let us say I write 1 to x. So, another core reads 1 and I read 0. So, then and further more there could be intervening accesses to other variables. So, there could be a break in atomicity, but pretty much the key idea is that look this order. So, these are separate cores. So, this order of Wx1 to Rx1 I might not agree that the write fully completed which means that all the cores receive the value of 1 and then let me give this numbers.

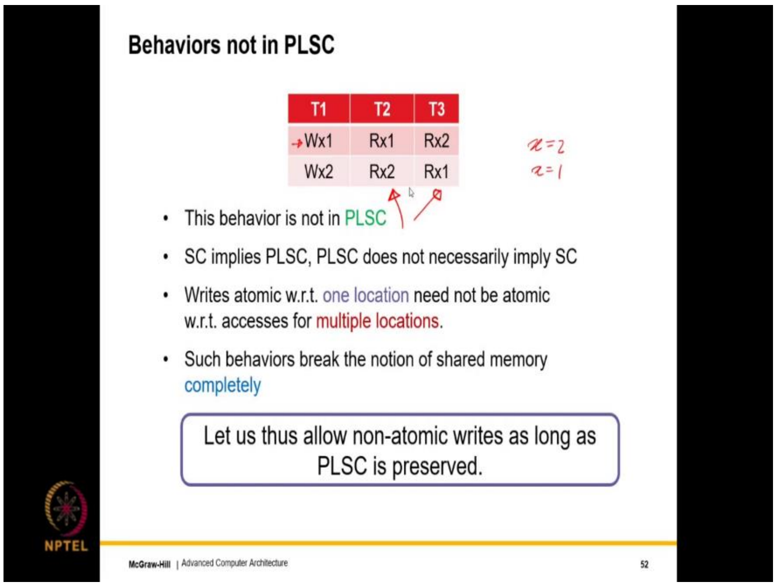
1, 2 and 3 then, core number 2 actually read it. So, I will not agree with that even the core 2 will say that this write was fully completed hence I read x = 1, core 3 will not agree core 3 might say that look this write actually did not. So, I will go back to that slide that talks about this. So, just look at this. So, in this case core 2 might say that look this write fully completed which means that the write was broadcasted all over the place all the cores had x = 1 and I read x = 1.

But I am core 3, I might not agree that the ordering between the write and the read was global because I will say that look that is not correct you read 1 early, but when I try to read x mainly because you read x = 1 you set y = 1 then I read y = 1 and when I try to read x I read it to be 0. So, I do not really agree that the write fully completed and then you read.

The write was in the process of completing a you went and I did not read. So, the write to read order I will say that it is not global it is local. Fair enough since we allow non atomic writes there is nothing much that there is no reason why we should have a problem, but the remaining two conditions are directly following from PLSC.

So, consider write to write. So, consider two observers at two different cores they cannot really record the fact as you can see over here that these writes happened in a different order.

(Refer Slide Time: 52:32)



Behaviors not in PLSC

T1	T2	T3
Wx1	Rx1	Rx2
Wx2	Rx2	Rx1

$x=2$
$x=1$

- This behavior is not in PLSC
- SC implies PLSC, PLSC does not necessarily imply SC
- Writes atomic w.r.t. one location need not be atomic w.r.t. accesses for multiple locations.
- Such behaviors break the notion of shared memory completely

Let us thus allow non-atomic writes as long as PLSC is preserved.

McGraw-Hill | Advanced Computer Architecture 52

So, T2 is recording that write to x, x = 1 happened first and x = 2 happened later, but T3 is recording that x =2 happened first and x = 1 happened later this means that x has ended up with two different final states. T3 says the final state of x = 1, T2 says the final state of x = 2. So, this completely breaks the notion of their being a single physical location associated with x. So, since they have two separate final states, we cannot allow the write to write order to actually get violated.

See even if we consider accesses to multiple variables we cannot say that look I mean a observer at a core which is seeing accesses to all variables, but its only recording accesses between two accesses to the same variable, it has to record a write and a write two writes possibly issued by different cores, it has to record a consistent global order between them.

If let us say one core says that I wrote x = 1 first and then somebody else wrote x = 2 after that, all the cores better agree otherwise x will end up in two final states. So, regardless of who issues the writes this ordering has to be global. So, what is this order again? Well, this order is between operations to the same variable and these operations can be issued by different cores.

So, what we are saying is that look for the observers what are they going to record. See if there are two writes to the same address which can possibly we issued by different threads they will still record the same order otherwise we will end up with two different final states not allowed. Let us say that some core reads and some other core writes after that. So, you read x = 0 and somebody else writes x = 1.

Since writes are globally ordered you will assume that one core records let us say W i. So, in a sense I write 0 = x, then I read 0 then I write 1. If all the cores record that this read happened before this write because the core that issued R i could not have seen W j else it would have read a different value. So, what I am trying to say over here is as follows, if let us say thread 1 read some value for x and thread 2 read some other value thread 2 then wrote some value to x.

If let us say another thread records that this read happened first and this write happened after that can any other thread record it in some other way. So, can any other thread record this specific interaction in some other way what we are saying is that as a direct consequence of PLSC there are more details in the book, but the crux of what I am trying to say is that this is not possible.

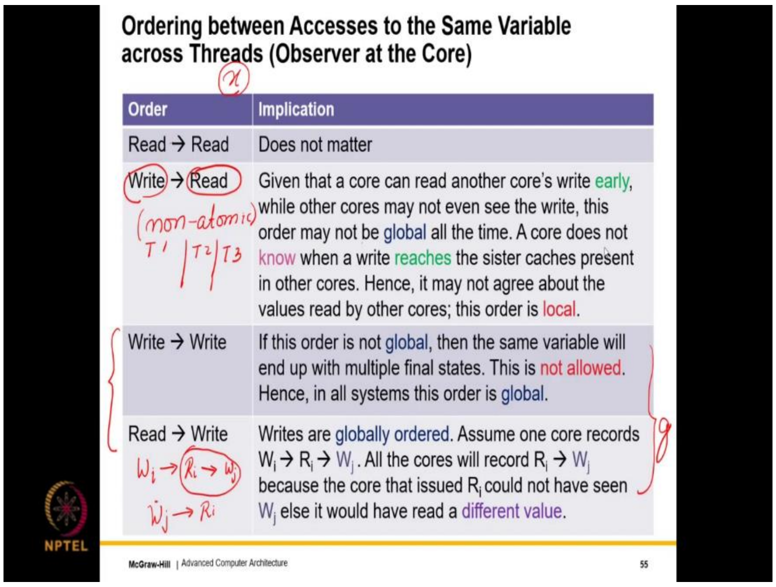
So, what I am trying to say is that both of these will be global and this is not possible and the reason that I am saying that this is not possible because if some other thread has recorded that R i. So, let us say some other thread has recorded a different order for example, it has recorded that it is recorded the reverse order, it will have some reason for recording it. The reason is that it would have seen some path that I wrote I did the jth write and then I did a sequence of actions and finally, thread 1 read x = 0.

See if there are a sequence of actions between this write and this read, it clearly means that thread 1 given the global order writes would have been aware of this write. So, you would have read the value that W j actually wrote, but that is not happened that is not happened it is read the value that has been written by a previous write.

So, given then that has not happened we have a contradiction here and it will never be the case that a read to write order between operations to the same address I will be issued by different threads will actually be local. In the sense two threads will have conflicting views of the order. So, that will not happen. So, let me now summarize.

So, what I am saying is that if I have observers at a core at each core. So, observers at each core will record what they are seeing right pretty much what all they are seeing in terms of reads and writes. So, the reads and writes can be issued by different threads and they will record all of that and let us consider all the records that pertain to a single address.

(Refer Slide Time: 58:10)



Ordering between Accesses to the Same Variable across Threads (Observer at the Core)

x

Order	Implication
Read → Read	Does not matter
Write → Read (non-atomic) T1 | T2 | T3	Given that a core can read another core's write early, while other cores may not even see the write, this order may not be global all the time. A core does not know when a write reaches the sister caches present in other cores. Hence, it may not agree about the values read by other cores; this order is local.
Write → Write	If this order is not global, then the same variable will end up with multiple final states. This is not allowed. Hence, in all systems this order is global.
Read → Write $W_i \to (R_i \to W_j)$ $W_j \to R_i$	Writes are globally ordered. Assume one core records $W_i \to R_i \to W_j$. All the cores will record $R_i \to W_j$ because the core that issued R_i could not have seen W_j else it would have read a different value.

McGraw-Hill | Advanced Computer Architecture 55

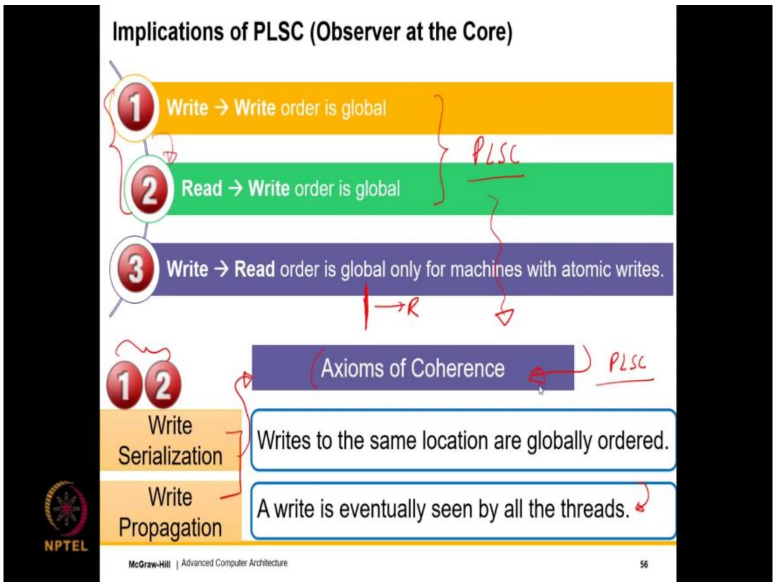
So, in this case the observer at the core will basically say that look the read to read order does not matter because in any case if there are no intervening writes its immaterial. The write to read order given that we allow non atomic writes will actually not be global in the sense an observer will basically say that look before a write was fully complete you went ahead and read it. So, I will not agree that the write actually completed first and then you read and I may at best say that they are concurrent, but I will never agree that a given write happened and then you read.

So, go back to our example of 3 threads T1, T2 and T3 where we clearly saw a case where PLSC holds, but atomicity does not. Now if we consider these two orders. So, regardless of the fact that you are seeing accesses to other variables as well, a write to write order will always be agreed try to write order to the same address will always be agreed to by all threads otherwise the same address might be might end up with different final states.

And same is the case with a read to write order. So, read and write are a symmetric over here because let us say that we have a write to a given location then we read its value then there is a subsequent write. So, all the threads will agree on this order why? Because if they do not and 1 thread actually records this then using this logic R i should actually written their value written by this write well that is not happening. So, there is a contradiction.

Because of that these two orders are global and the first order is actually local from the point of view so the point of view is important extremely important from the point of view of an observer at a core or on a thread.

(Refer Slide Time: 60:28)


Implications of PLSC (Observer at the Core)

1. Write → Write order is global
2. Read → Write order is global
3. Write → Read order is global only for machines with atomic writes.

PLSC

W → R

	Axioms of Coherence
Write Serialization	Writes to the same location are globally ordered.
Write Propagation	A write is eventually seen by all the threads.

PLSC

NPTEL
McGraw-Hill | Advanced Computer Architecture
56

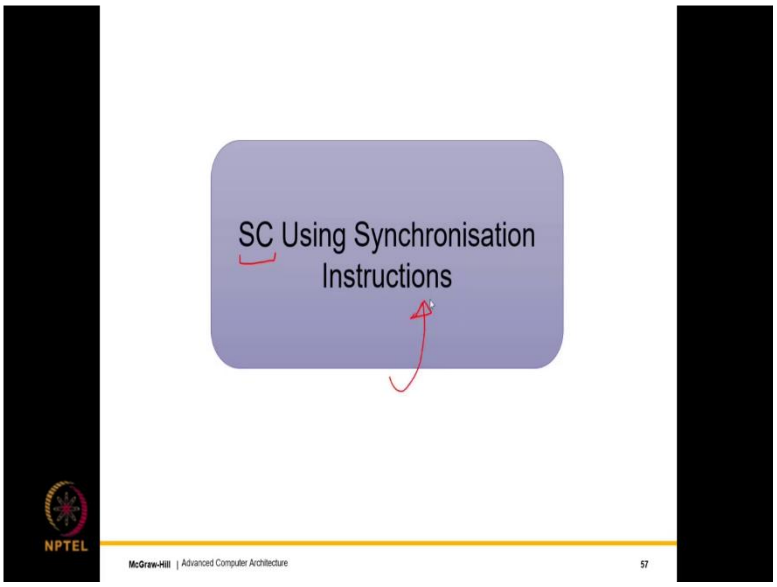
See if I were to summarize these things the write to write order write is global. So, read to write order is global and the write to read order is global only for machine only for machines with atomic write only if an atomic write then all of us will agree that look the write completed instantaneously and then another thread rate, but if the write is not completing instantaneously at one point in time the write to read order will not be globally agreed to, but the rest will follow from PLSC. So, these orders will be agreed to by everybody.

So, the axioms of coherence can sort of be distilled from PLSC and axioms of coherence basically say that writes to the same location which essentially covers points 1 and 2 are globally ordered. In the sense we all agree that for a given location I wrote 1 first, then 2, then let us say 3, then 5, then 7, then 17 and so on.

So, there is no between observers stated at different cores there is no discrepancy in what they have recorded. Given that writes are globally ordered the second follows and coherence in that sense is mostly equivalent to PLSC, but coherence has one additional constraint which is write propagation first is called write serialization where serialization means putting it in a sequence. So, we are doing that and write propagation means that a write is eventually seen by all the threads.

So, writes are never lost. So, write messages are never drop by the system they eventually seen by all the threads. So, these two are the basic axioms of coherence where one follows from PLSC and the other follows from the fact that we never drop a write. So, if we can ensure that our caches are coherent. So, they would automatically follow PLSC because they are capturing the gist of PLSC. And in addition they are also ensuring the fact that writes will ultimately propagate reach their locations and complete.

(Refer Slide Time: 62:52)

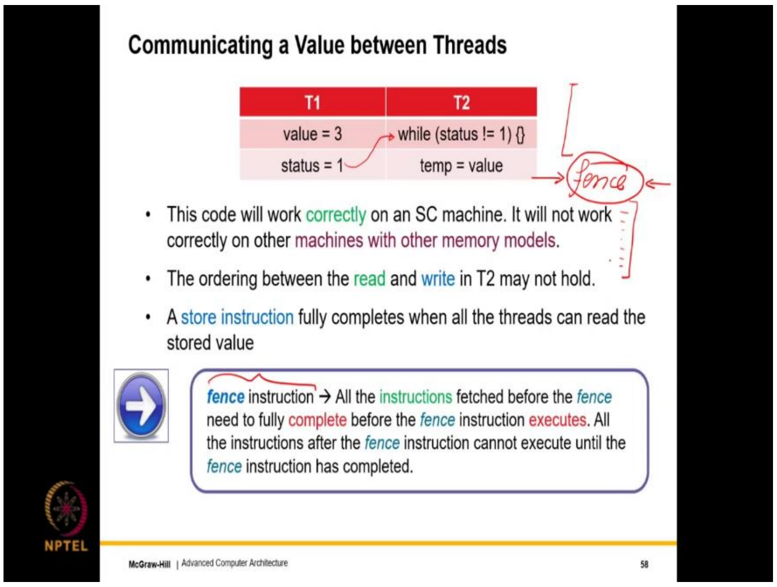


SC Using Synchronisation Instructions

McGraw-Hill | Advanced Computer Architecture 57

So, now, next we will discuss how to enforce sequential consistency or how to kind of force sequential consistency in systems that have a wide variety of optimizations using what are called synchronization instructions.

(Refer Slide Time: 63:18)



Communicating a Value between Threads

T1	T2
value = 3	while (status != 1) {}
status = 1	temp = value

- This code will work correctly on an SC machine. It will not work correctly on other machines with other memory models.
- The ordering between the read and write in T2 may not hold.
- A store instruction fully completes when all the threads can read the stored value

fence instruction → All the instructions fetched before the *fence* need to fully complete before the *fence* instruction executes. All the instructions after the *fence* instruction cannot execute until the *fence* instruction has completed.

McGraw-Hill | Advanced Computer Architecture 58

Say SC using synchronization instructions would look like this. So, let us look at some of the code that we have been looking at in the past. So, what we have is, we have value = 3 we set the status = 1. So, this would be a typical way of communicating across threads which unfortunately breaks if you do not have SC. So, then we key have a while loop which keeps on looping to see when the statistics becomes = 1 and the moment we see the status = 1 we read the value of value.

This will unfortunately not work on a non-SC machine, the reason being that number 1 the writes may be reordered if program order does not hold that is point number 1, point number 2 is the write may not be atomic. So, it is possible that this write at this point over here may not be visible. So, because of this appears to be a very easy way of transferring a value across threads, but this is not going to work.

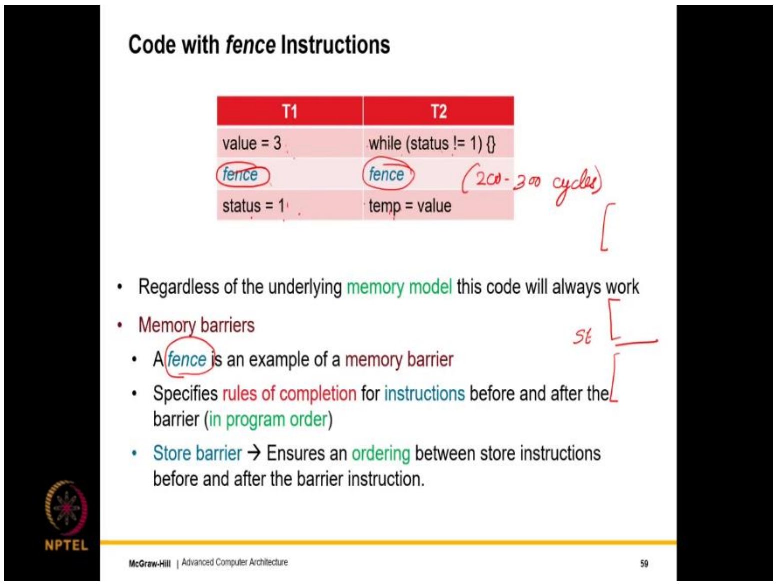
You can try it and see. So, this is not going to work with 100% accuracy had it being sequential consistent sequentially consistent it would have worked, but it will not work. So, the ordering between the read and write in T2 also may not hold also as I said atomicity is an issue this store will may not have fully completed. So, T1 might have said that look the store is done, but well it might not have reached T2 while status may have reached.

So, that is why we add a fence instruction where all the instructions fetched before the fence need to fully complete before the fence instruction executes, which essentially means that if I have a fence which I will show in the next slide, this ensures that before the fence even you start executing everything before it reads writes everything all the reads need to get their value all set and all the writes should have committed and not only committed, but the value should have reached all the cores.

In the sense, it would have fully completed in the and it would have updated all the physical locations that the variable is associated with. Furthermore, after the fence no instruction in program order after the fence would have actually started its execution all the instructions after the fence cannot execute until the fence instruction is completed.

So, all architectures support fence instruction where we see everything before me fully complete nothing after me starts.

(Refer Slide Time: 66:18)



Code with *fence* Instructions

T1	T2
value = 3	while (status != 1) {}
fence	*fence*
status = 1	temp = value

(200-300 cycles)

St

- Regardless of the underlying memory model this code will always work
- Memory barriers
  - A *fence* is an example of a memory barrier
  - Specifies rules of completion for instructions before and after the barrier (in program order)
  - Store barrier → Ensures an ordering between store instructions before and after the barrier instruction.

McGraw-Hill | Advanced Computer Architecture 59

So, how would this piece of code look like with fence instructions? The way it would look like is that we set the value to 3 we have a fence. So, this ensures that when we set status to 1 the value is already fully written it is also called globally completed or fully completed.

But in layman terms what it ensures is that look when I write the status the value is fully completed and then when I read it over here, it basically tells me that status is 1 which means value is 3 and to ensure that these instructions are not reordered I add a fence.

So, then if I add an another fence over here when I actually come here I am sure that this instruction is done this instruction knows the status is 1 and because of the fence we are sure that value has been set = 3. So, regardless of the underlying memory model whatever is respected in terms of program order or in terms of atomicity it does not matter this code will always work.

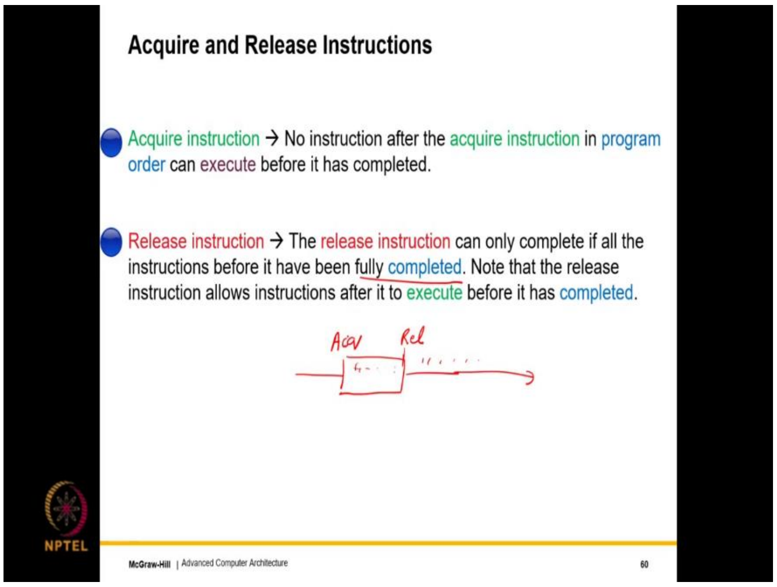
So, the way that an engineer is supposed to write parallel code is by inserting fences and fences are expensive operations. So, a fence would take 200 to 300 cycles, it is like a pipeline flush in the sense everything before it needs to fully complete and also everything before it we need to send either we get acknowledgement messages from the memory system its fully done or we need to flush the write buffers and MSHR, such that no write is there in any temporary structure.

So, a fence is the broad is an example of a memory barrier, say barrier is basically something that put some sort of a precondition on instructions before it before it actually executes. So, there are many other kinds of barriers you can have what is called a store barrier it will say that look before I execute all the stores need to fully complete.

I do not care about loads, but all the stores need to fully complete only then a store after me can began. So, in this case instead of a fence we can have a store barrier which would be more efficient write. So, similarly we can have a load barrier which will be between two loads.

So, we can have barriers of different types fence is of course, very generic it does not differentiate between the read or write, it just says before me all complete after me nobody start.

(Refer Slide Time: 68:59)



Acquire and Release Instructions

- Acquire instruction → No instruction after the acquire instruction in program order can execute before it has completed.
- Release instruction → The release instruction can only complete if all the instructions before it have been fully completed. Note that the release instruction allows instructions after it to execute before it has completed.

McGraw-Hill | Advanced Computer Architecture 60

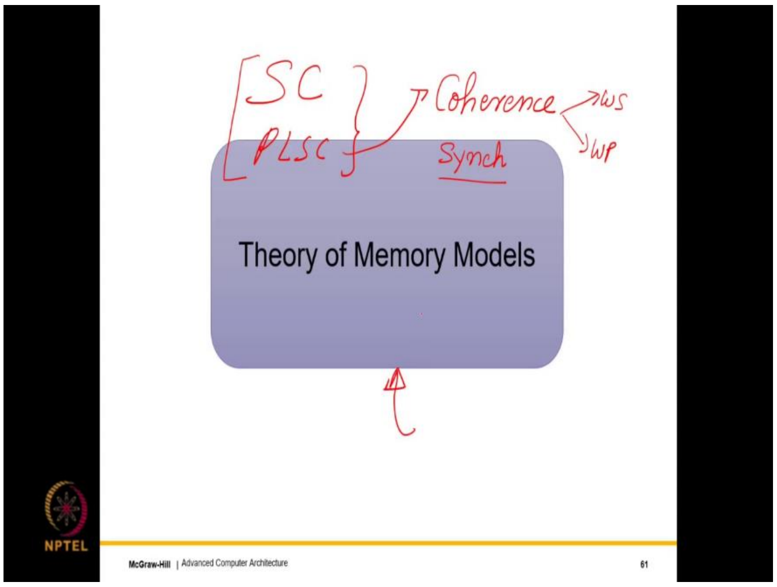
So, we can have a special kind of fence operations. So, these actually operate in pairs. So, we will see the importance of this when we discuss critical sections locks and unlocks. So, there are two kinds of fences here an acquire instruction and a release instruction.

So, an acquire basically says that when I want to start what is called a critical region of core, I have started with an acquire which means no instruction after the acquired instruction program order which is essentially these ones can execute before the acquire has completed. So, I need to do acquire first, then everything after it starts which means that either I acquire some permission or some lock or we will see what exactly we acquire, but let us put it this way that something needs to be acquired.

Similarly, when I am done with this piece of code I call release, the release instruction can only complete if the all if all the instructions before it which means all the writes and everything write reads writes everything have been fully completed, fully completed for a load is when the result comes and for a store is when the value is sent to all the cores.

So, note that the release instruction allows instructions after it to execute before it has completed in the sense instructions after it can execute before the release as completed, there is no issues there are you know absolutely no issues in that, but acquire says nothing after me starts until I am done and release says that I will finish only if everything before me finishes. So, we will see this there is a model called release consistency that uses this, but let us just remember it for a time being.

(Refer Slide Time: 71:01)



Theory of Memory Models

So, now we have discussed everything about sequential consistency PLSC. So, what have we discussed? We have discussed SC, we have discussed PLSC and we have discussed PLSC and how it relates to coherence you have discussed two axioms of coherence the serialization axiom which means writes to the same location as a global order and how exactly what are the implications of that and the fact that writes are never lost.

We have write serialization axiom and a write propagation axiom and then we discussed synchronization instructions synch we will call them synch, we will not add a dot we should, but we will take a little bit of liberty with English. So, we will call them synch instructions. So, now, we have all the basic ingredients to dive right into the theory of memory models.