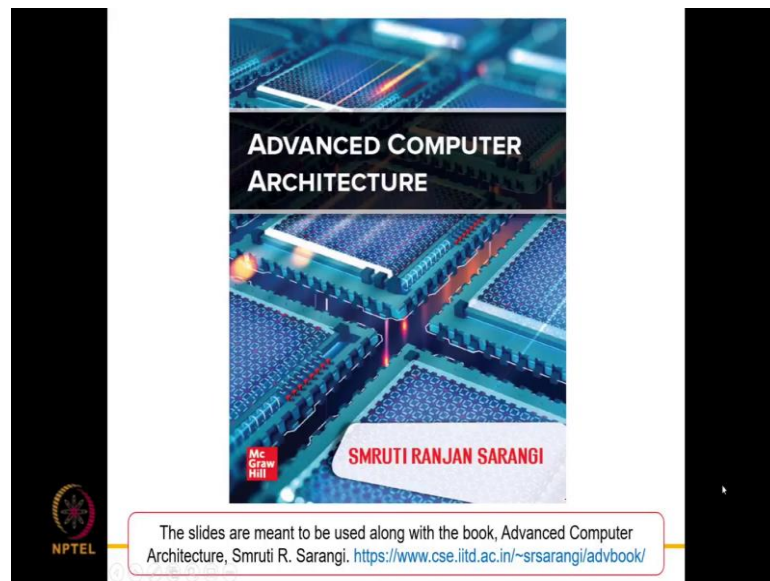


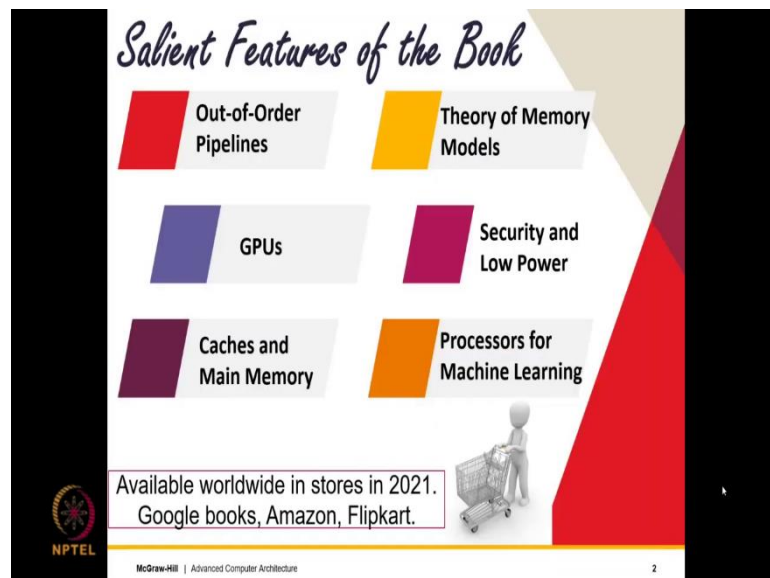
Advanced Computer Architecture
Prof. Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Chapter - 09
Lecture - 31
Multicore Systems Part - VII

(Refer Slide Time: 00:17)



(Refer Slide Time: 00:23)



(Refer Slide Time: 00:38)

Contents	
1.	Parallel Programming
2.	Theoretical Foundations
3.	Cache Coherence
4.	Memory Models
5.	Data Races
6.	Transactional Memory

NPTEL
McGraw-Hill | Advanced Computer Architecture
128

So, we will now discuss memory models and finally, cap our discussion with data races. So, we will find the data races and memory models in a certain sense are quite intimately connected and this will be explored in this lecture.

(Refer Slide Time: 00:56)

Memory Models

- The **memory model** is dependent on the **processor architecture**. If there are very aggressive optimizations, then the memory model has to be **very weak**.
SC
- PLSC **requires** the **ws** and **rr** orders to be global. All popular architectures **follow** PLSC, and many **disallow** thin air reads.
- The only orders that can be local are **rr** and **po**.
rr: (non-atomic)

NPTEL
McGraw-Hill | Advanced Computer Architecture
129

So, first let us do a quick recap of what is it that we learnt. So, the memory model or the memory consistency model is essentially a specification of what kind of executions are allowed and for a piece of code what are the valid outcomes. So, it basically determines the rules.

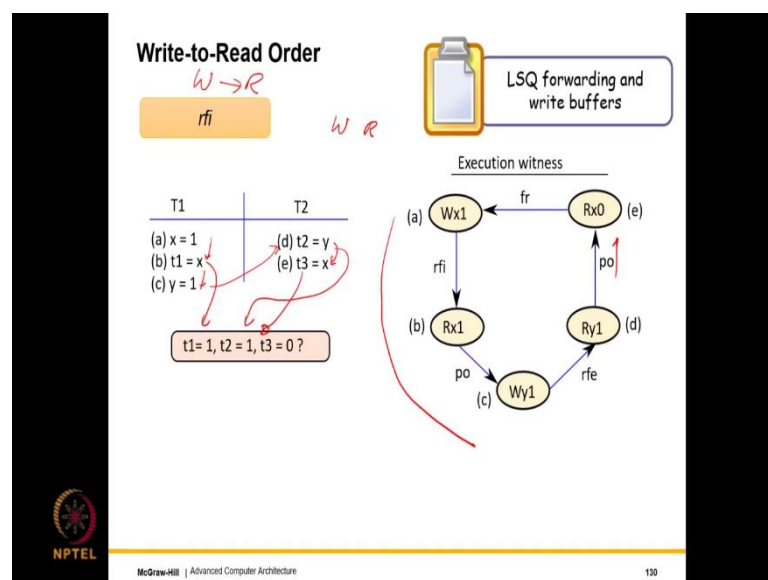
And the memory model is dependent on the processor architecture even though as we shall see it can we can also define memory models for virtual machines, for software, for compilers and so on ah, but the way that we have been explaining up till now it is primarily being defined in the context of a processor architecture.

So, we can have very aggressive optimizations. So, then the memory model becomes rather weak. And so, the goal standard and the strongest memory model is sequential consistency. So, PLSC requires the w s and f r orders to be global. So, this was a direct consequence of PLSC and all popular architectures follow PLSC and they implement coherence which ensures PLSC.

And many also disallow thin air reads which as we saw can happen if we have value prediction along with a couple of more pipeline optimizations. So, the only orders that we can actually change which are not global which are local are r f and p o primarily.

So, r f well as we saw, we can have 2 r f i and r f e; r f i is read your own write early and r f e is read others write early. So, reading others write early is same as non atomic writes and all kinds of program order relationships can be relaxed that is what we saw, relax means not followed.

(Refer Slide Time: 03:00)



So, let us now look at a series of execution witnesses and let us see how observing something and not observing something? how it actually allows or disallows an outcome?

So, we will discuss a couple of execution witnesses here, but my advice to the readers and viewer would be to look at this in detail in the book and actually go through these examples in great detail.

So, let us first look at the order which is relaxed almost all the time the write to read order and this is a direct consequence of having an out of order pipeline load store queues forwarding write buffers and so on, where essentially if there is a write and then a read comes after that well the write of course, executes at the end, at commit time the read executes earlier. So, the read in a sense is visible to other processors earlier than a write. So, this is a direct consequence of out of order processing as well as other kinds of optimizations.

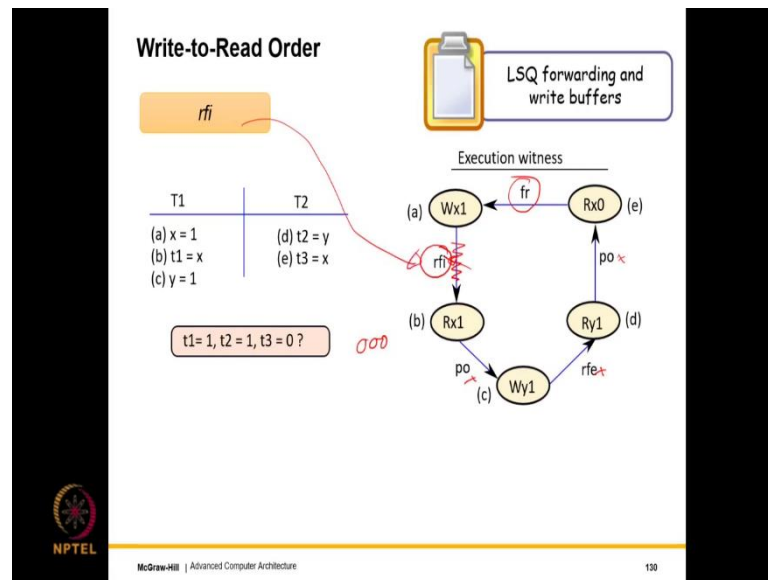
So, typically many architectural structures can kind of relax and order. So, let us look at one example and let us try to study it. So, we set first set $x = 1$ then there is a read instruction after this. So, the read instruction in a sense reads it early, but let us assume it does not do that. So, we will see what will happen if it does not do that.

So, let us just look at these three instructions a b and c where we write $1 = x$ we read 1 from x because this anyway we will have to read because it is the same thread and then we write $1 = y$. So, till this point it is fine after that what happens is that instruction d reads 1 from y which is also fine and then we have to read operations. So, let us assume the program order between them holds.

So, then what are we going to read? So, well we have two options either we can read 0 or we can read 1's if you see the outcomes we have read $t_1 = x = 1$, $t_2 = 1$. So, now, the question is what do we read t_3 to be? So, in a lot of architectures what you will practically see is that the core will read its own write early before the write is actually visible to other cores or other threads it means the same thing in the context of our current discussion.

So, what t_2 will actually do it when it tries to go and read the value of variable x, it will get an older value which is 0 and needless to say then the f r the form read order will hold between $r_x 0$ and $w_x 1$.

(Refer Slide Time: 06:17)

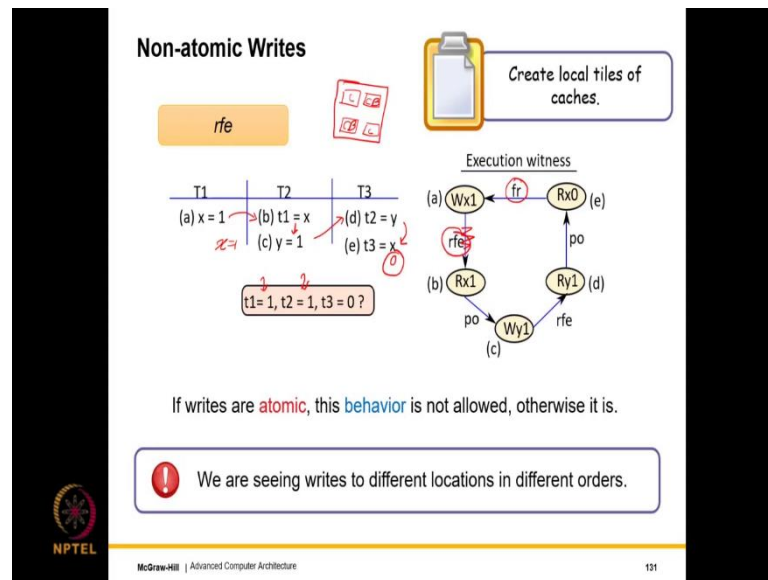


So, let us see out of this what is it we can relax. So, if you have atomic writes we cannot relax this. If we observe the read to write order, we cannot relax and most of the time we do and read to read also if we observe we cannot relax and f r is global anyway.

So, this execution will not be valid because there is a cycle if the r f i order is global, but most often it is not most of the time it is not when we are using an out of order pipeline that is why this edge will not be there. Hence, in any architecture where the r f i order is not global, but the rest of the edges are global this execution will be allowed. So, this in a certain sense is a fair execution; because we should be allowed to read or own writes early. So, this LSQ forwarding will give us this write buffers will give us this.

And then the write later on can be visible to the rest of the other course. So, this is the classic out of order pipeline optimization where we finish the loads early, primarily because loads are on the critical path and we would like to finish them early. So, that is a vital feature for a speed up. So, that is why this execution is built by and large we acceptable in almost all architectures so with any semblance of out of order execution. So, this is kind of the implication of the r f i order being global.

(Refer Slide Time: 07:54)



So, now, let us look at the *rfe* order. So, here of course, we look at non atomic writes. So, we have already seen something of this nature earlier also when we introduced the idea of non-atomic writes. So, here what we do is that. So, as we were discussing non atomic writes can come about in an architecture if we have a local tile of let say caches in the sense we divide our entire chip into tiles of core cache bank complexes.

So, it is possible for another core in the same tile to read the write of the sister core before other cores have seen it. So, this can happen particularly if the share is snoopy bus between them and the so, this is definitely a possibility. So, of reading others writes early and also what can happen is if the share some other structure like an a MSHR or something then also this can happen that before others have seen it one core will see the right and then take decisions on the basis of it.

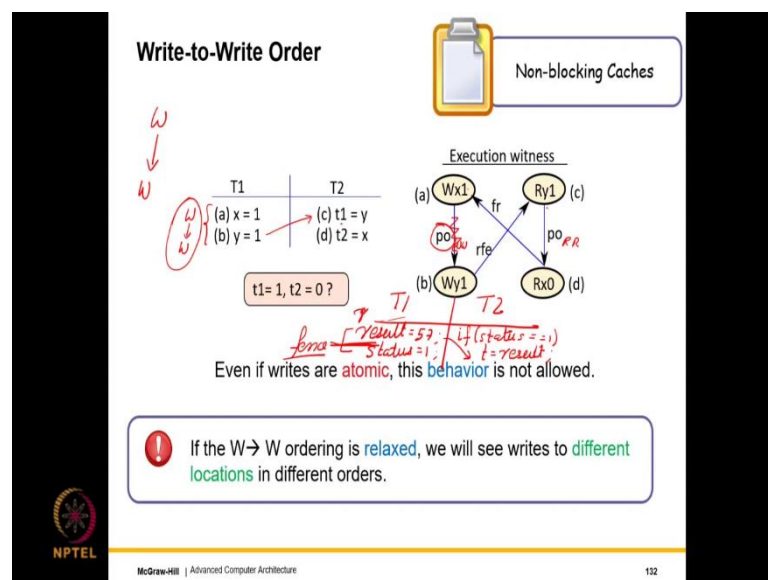
So, let us do a quick recap of the execution that we had seen earlier, we set $x = 1$ then we read the value of x we read it to be 1, then we set the value of $y = 1$ we read the value of y to be 1. We assume program model over here and then we read the value of x . So, if this write would have been atomic we would have read 1 as you can see clearly from this sequence, but we actually read 0. Consequently, there is a cycle in the execution witness because we need to add an *fr* between $Rx0$ and $Wx1$.

And needless to say if the write would have been atomic this behavior would not have been allowed, but given that writes are not atomic we will not have this edge, if you do not

have this edge you see that there is no cycle and this execution is allowed. So, basically what we are doing is that we are essentially seeing writes to different locations in different orders. So, t_2 sees x , so, x is written $x = 1$ first and then $y = 1$ whereas, t_3 sees that $y = 1$ first and then $x = 1$.

So, we are essentially seeing writes to different flow locations in different orders which is a hallmark property of non-atomic writes, but even in with non-atomic writes we still have PLSC. So, we see the writes to the same location in the same order, but for different locations that is where there is a little bit of mix up primarily because this edge is not global that is the main reason and that is why we do not have a cycle in this graph.

(Refer Slide Time: 11:14)



So, let us now look at the next stage which is typically relaxed which is the write to write order. So, this can happen in a non-blocking cache where what will happen is that the first write will kind of get stuck in the MSHR, but the second write for many reasons well the second write we might either directly affect the write or we might send it to the lower level by passing MSHR for a wide variety of reasons the first write might get stuck might get delayed.

But, the second write might reach the rest of the course. So, we will see that even for the same thread as long as the addresses are different the write to write order is being violated. And typically in non-blocking caches where we would stop the write at an MSHR and we might let the other write go through that is where such problems can happen. So, here also

a simple example would be we write $1 = x$ and $1 = y$. So, we have $W x 1$ and $W y 1$ we have a program model edge between them which is exactly what we are contesting.

Let us assume we have atomic writes. So, then we read $R y 1$ then let us assume we have a read to read order. So, this is not something that is being contested and then we read $x = 0$. So, that is the so that is essentially the crux that we read $y = 1$. In the world of atomic writes with write to write order we would have read $x = 1$, but we read $x = 0$ that is why we need to add an f r h between $R x 0$ and $W x 1$.

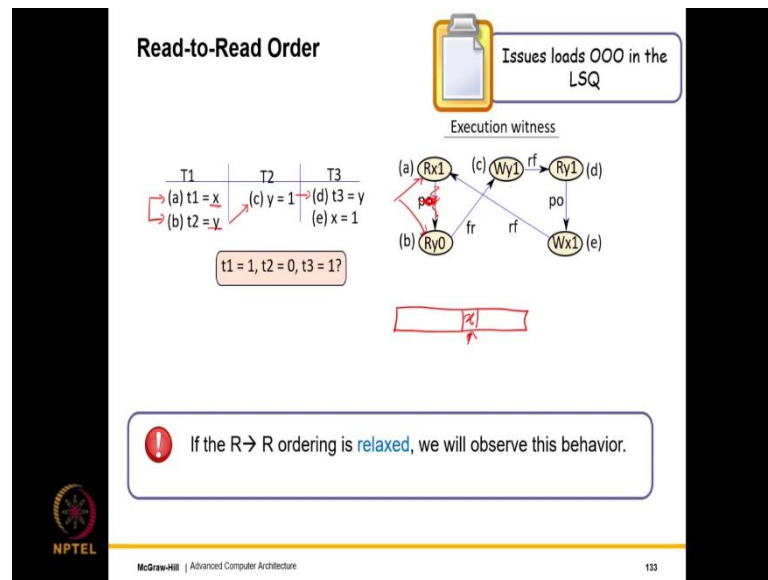
And we have a cycle. So, this execution is not possible, but if the write to write ordering would not have held then what would have happened is we would not have added this edge, then there would have been no cycle and this order would have held. So, this execution would have been fine.

So, what again is the summary of our discussion that T 1 will see the write to x first and the write to y later, T 2 will actually see them in the reverse order it will see the write to y first and the write to x later. So, here also we see write to different locations in different orders and this is perfectly fine if you write our code in this manner and our write to write ordering is not respected this is exactly what we are going to see?

So, that is why when multiple addresses are concerned unless we add fences others will not be able to see the data and this does have a lot of implications when you are writing for example, communicating values. So, what we might do is we might set. So, let us consider two threads write T 1 and T 2 we might set result is equal to let us say some value let us say whatever 57, set status = 1 which indicates the competition is done.

Then we read the status if status is equal to 1 which is exactly what we are doing over here and then we read the result. So, we set some temporary is equal to result, but we are not guaranteed to see this result, the reason we are not guaranteed is that these writes might get flipped. So, because its ordering is not guaranteed this kind of a code pattern will not work it is not sequential consistency. So, we need to add a fence over here. If we add a fence over here then of course, we are guaranteed to see the correct behavior.

(Refer Slide Time: 15:25)



So, the next tricky one is the read to read order and this is very easy to violate because well we are talking of two different reads to two different addresses and this ordering is always relaxed in an out of order processor with an LSQ, where we might have two reads and then they are simply issued in different orders write as simple as that that 1 read is stuck for some reason. So, let us say we are using we have an LSQ and we have some sort of a predictor of whether we will have a dependence or not.

So, we have two reads read to x and read to y. So, the read to x we predict that there will be a collision. So, it is kind of stuck. So, we do not let it read any data and we do not send it to the cache, but the read to y we predict there will be no collision. So, we send it to the cache.

So, in a certain sense violating this read to read order with aggressive optimizations particularly a load store queue where we employ dependence prediction its actually very easy and we have seen that happening in chapter 5 where the read to read order got violated pretty easily.

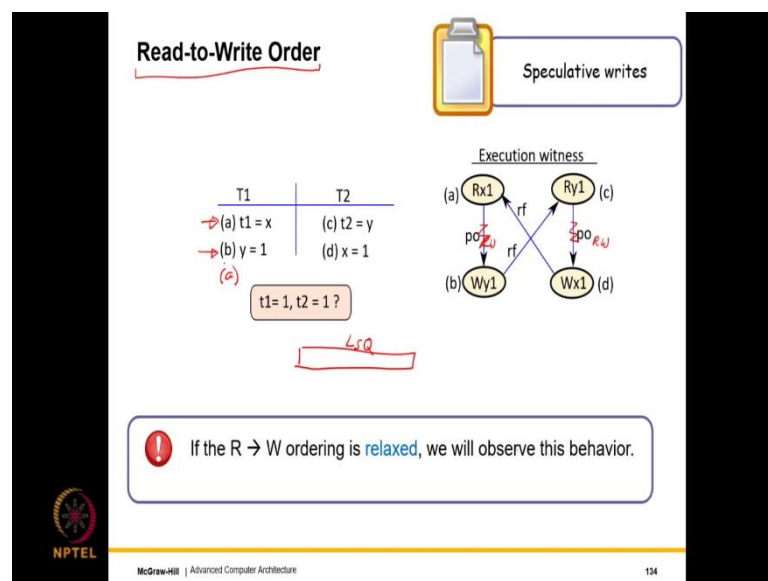
So, in that case, what we need to do? let us consider this example first where we read x and y. let us say we read x = 1 we read y = 0. So, from R x 1 to R y 0 then we set y = 1. So, we add an r r h then we read y = 1. So, we add an r f h and then we have a program order edge over here. So, the program order edge is between R and W read and a subsequent write and W x 1 then provides the data to R x 1.

So, in this case, it is a read to read ordering is relaxed, what will essentially happen is that we will essentially reorder the reads and that will cause us some trouble. So, the read to y if it goes earlier it would read an early value of y which is 0 and then of course, x will return 1 and as you can see there will be cycle in this graph.

And any kind of an of a processor with an aggressive out of order load execution scheme with of course, an LSQ where we need not we are not bound to issue loads in the order in which they were fetched. And this can happen particularly when we predict dependencies and so, on this will happen this a definitely a possibility this will happen and if this happens then this execution will actually have a valid outcome.

So, now, we can have two kinds of processors Intel processors for example, enforce a read to read order in this case this execution is not allowed this outcome is not allowed. But, in most other out of order processors this execution, this edge is not there. If this edge is not there you can easily see that this outcome will be allowed.

(Refer Slide Time: 19:00)



Now, let us come to a read to write order, so a read to write order is pretty hard to violate actually. So, what does this mean? This means there is a read and there is a write, to violate this order what will happen is that we need to execute this read after this write.

So, we need to execute kind of this read over here. So, that is not really the way that this will actually happen because just look at when the write will execute, the write will execute

when it reaches the head of the ROB and by that time the read would be gone and so read will not be in the pipeline. So, it must have finished its action it must have gotten its value and it must have gone.

So, there is simply no way that we will actually in a sense overtake it, but it is possible in several scenarios. So, one of the scenarios in which it is possible is if let us say we have a speculative write in the sense that we have an aggressive optimization where we actually do a write before an instruction reaches the end of the pipeline. So, we kind of do an aggressive write. Another example could be where let us say these are two separate hardware threads and they share the same load store queues that is actually stores physical addresses.

In this case writes can be visible early. So, they can be kind of speculatively visible so in that case, you will have such outcomes where let us say we read first. So, essentially our problem is with the outcome $t_1 = 1, t_2 = 1$ that is because in both the cases the write instructions the write operations are the later operations. So, both the reads cannot be 1, otherwise, will have two program order edges write over here, two r f edges which we presume to be global because writes are atomic and we have a cycle.

So, in a lot of processors where the read to write ordering will hold in lot of simple processors you will not see this execution actually because in then the read to write ordering will hold, but this will be visible in many aggressive implementations where the write is visible early because of some of the reasons that I mentioned, in that case, this outcome will be allowed because we will not have these edges. So, there will be no cycle in the execution witness.

So, given that in this case this order will not hold we will find this outcome to be legal, but as I said this is the most difficult order to violate because this lies in the phase of whatever we have been teaching in on the out of order water pipeline. So, unless we have extremely aggressive optimization it is rather unlikely that this order will get violate.

(Refer Slide Time: 22:13)

Special case of *rfi* in SC.

Can the *rfi* relation be relaxed in SC?

read its own write early

Yes!

W → R

The proof is there in the book →

NPTEL

McGraw-Hill | Advanced Computer Architecture

135

So, let us now look at an important corner case. So, the corner case is like this that we have up till now been maintaining that in sequential consistency all the relations are global.

But, now I would like to ask a simple question that let say we relax the *rfi* relation and everything else is global, which means that a code can; core can read its own right early. Will it break SC? Well, the answer is no. So, this relation can be relaxed in SC and the proof is there in the book. So, the broad contour. so, the proof is proof are like this that let us say we have a write and then we have a read.

So, the read reads the value from the write. Then another core sees the effect of the read of course, via other instructions could be in the same thread as well and then it does a couple of other things and finally, there is a cycle that goes back to the write. So, this is how a cycle would form and then you would say that look this execution is not an SC, but what I am saying is that and this is something that is in argued in the book that this will actually not happen.

The reason it will actually not happen is because there will be an ordering between let us say this instruction and instruction in the other thread and the write because either the dependency will go via this read in this case the only dependency from a read to another will be to the same address a read to a write. So, you will have one edge like this.

So, we will not have a cycle or it will go via other instructions in all cases it is possible to prove that we will actually not have a cycle. So, this is an important aspect of sequential consistency that it does allow the r f i relation to be relaxed. The primary reason being that other course will not be able to make out that it actually has been relaxed because they will not see the effect.

So, the execution will be indistinguishable from a situation where it is not relaxed and other course for them to see they will either see it via other instructions or via the read instruction. But, then again the write instruction will have to execute before them there will be happens before relationship between the write and those instructions those other instructions and that is why we will actually not see the effect, but I do not want to go very deep into it the proof is there in the book, but this should be kept in mind because we will use it in this next figure.

(Refer Slide Time: 25:09)

Summary of Memory Models

Relaxation	W → R	W → W	R → R	R → W	rfe	rfi
SC	-	-	-	-	-	✓
TSO (Intel)	✓	-	-	-	-	✓
Processor consistency (PC)	✓	-	-	-	✓	✓
PSO	✓	✓	-	-	-	✓
Weak Ordering/RC	✓	✓	✓	✓	-	✓
IBM PowerPC	✓	✓	✓	✓	✓	✓
ARM	✓	✓	✓	✓	✓	✓

✓ Ordering is relaxed
 force
 do not
 Co-allocate
 release
 CS

NPTEL

McGraw-Hill | Advanced Computer Architecture

136

So, this actually shows a summary of memory models. So, sequential consistencies are at the very top whereas, you can see nothing is relaxed it has atomic writes this is for atomic writes and we have relaxed the r f i relation in sequential consistency because we just showed in the previous slide that is possible to do so and get away with it.

Then we come to Intel's memory model which is a total store order memory model in the sense the only two relationships that are relaxed is the write to read order. This of course, has to be relaxed because we are dealing with an out of order pipeline.

And we also relax the r f i relationship which incidentally all models here do primarily because without that the benefits of an out of order pipeline are not going to be there. Next we have processor consistency which is also quite popular and distributed systems because you know a memory model is not just limited to a processor architecture we can extend it to a distributed system or a programming model.

So, the additional advantage over here the additional relaxation over here is that writes are not atomic. So, in processor consistency it is the same as TSO the write to read order everybody is violating, but the writes are in addition the writes are not atomic, but the important thing is that write to write orders are maintained. So, one processor even with non-atomic writes it maintains its own write to write order and that is why that is where the term processor consistency comes from.

Then we have PSO. So, PSO is kind of similar to what we call PC processor consistency it is just that it has atomic writes, but it does not obey the write to write ordering. So, that is the that is the swap over here that internal within the thread write to write ordering is not respected mainly because we will have non-blocking caches and so on and but writes are nevertheless atomic. Then we have weak ordering and release consistency. So, what is the difference?

So, weak ordering we do not respect anything which means we do not respect any of the orders, but we still have atomic writes and in this case we have a fence you have a generic fence instruction, but as we have seen in release consistency we have a pair of fence instructions we have a release an and acquire sorry it should be in other order it should be an acquire and then the release. So, the idea of acquire and release is that we will have the just take an acquire a lock and release a lock.

So, the instructions on a critical section are within it. So, the reason that we need a fence before and after a critical section or let us say in generic terms and acquire. So, the idea of an acquire is that first I acquire the lock and then I execute the instructions of the critical section or in other words no instruction or the critical section can execute before the lock has been acquired.

Or let us say this particular instruction has executed the acquire instruction is executed. So, that is the idea. The release basically means that what we do is that we can after

executing the release instruction it means that all the instructions in the critical section including the acquire have finished.

So, release signals to the outside world that all the instructions on a critical section have finished so you can go and read their data, unlike a traditional fence it does not prohibit instructions after it from completing before it. So, that much of out of orderness it allows as compared to a regular fence, but acquire and release essentially limit the behavior of the critical section that lies within them.

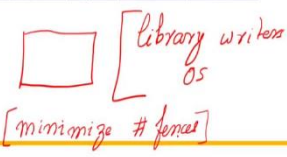
So, even release consistency it does not obey any of these program orders nevertheless has atomic writes and no programming, no memory model actually respects r f i. The IBM and ARM models are outliers in the sense in the sense they do not respect anything. So, they are non-atomic writes and additionally they also do not respect any of the program orders. So, none of the PO's they respect, but they do have fences though. So, the reason we can write correct code in the IBM and ARM memory models is because they have elaborate support for fences.

And not just regular fences, but all kinds of fences between stores between loads and so on which allows us to write fairly efficient code. So, who needs to know about memory models? Well the answer is pretty much everybody who works with multi-threaded code otherwise they will make mistakes because they will assume that the system operates in a certain way, but it will actually not operate in that way mainly because internally the processor can reorder instructions and you will see different outcomes.


Furthermore, what can happen is that maybe a certain piece of code which is actually buggy will work correctly on a certain machine, but on another day it might not work correctly.

(Refer Slide Time: 31:00)

Summary of Memory Models						
Relaxation	$W \rightarrow R$	$W \rightarrow W$	$R \rightarrow R$	$R \rightarrow W$	rfe	rfi
SC						✓
TSO (Intel)	✓					✓
Processor consistency	✓				✓	✓
PSO	✓	✓				✓
Weak Ordering/ RC	✓	✓	✓	✓		✓
IBM PowerPC	✓	✓	✓	✓	✓	✓
ARM	✓	✓	✓	✓	✓	✓



Library writes OS
[minimize # fences]



Ordering is relaxed

McGraw-Hill | Advanced Computer Architecture

136

So, we might have this parallel code which on one-day works fine, but on the other day it produces a non-intuitive outcome which the programmer will not be able to explain.

So, most library writers and people who deal with low level code. So, library writers, OS writers, compiler writers all lot of people who deal with low level code they need to understand the intricacies of the memory model in great detail because the main aim is to minimize the number of fences.

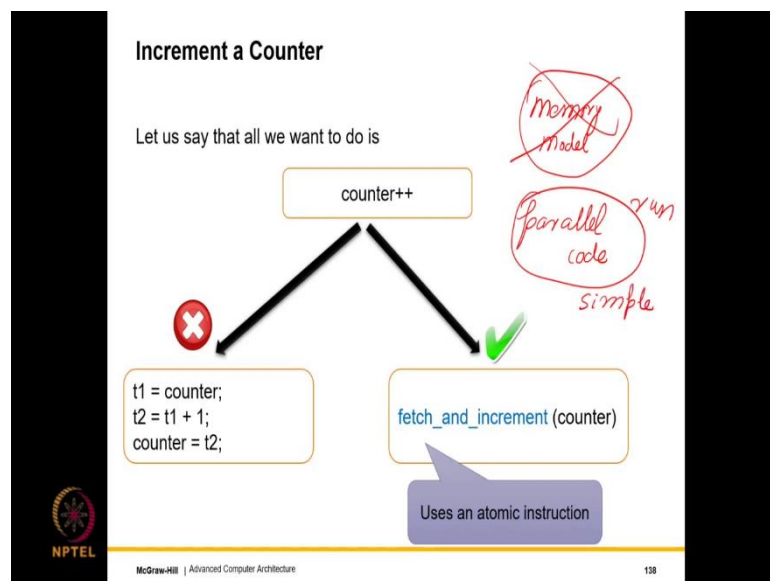
So, minimizing the number of fences as such is an undecidable problem and it is actually very hard to solve, but there are variants of it there are tools that can suggest where to avoid a fence and so on. So, clearly minimizing the number of fences is a research problem in its own and there are many tools to help us minimize them, but also black belt programmers know where to have a fence and where not to have a fence because they understand the memory model in great detail.

(Refer Slide Time: 32:10)



So, now the question is that we have used the word a black belt programmer what that means? is it is the programmer who needs to understand the memory model in great detail to actually write correct parallel code, but the question is that do we expect most programmers to be at the black belt level? The answer is no. So, we need to design something far simpler for the average programmer such that he or she can write correct code which will run regardless of the memory model and which will also run on all machines regardless of their memory models.

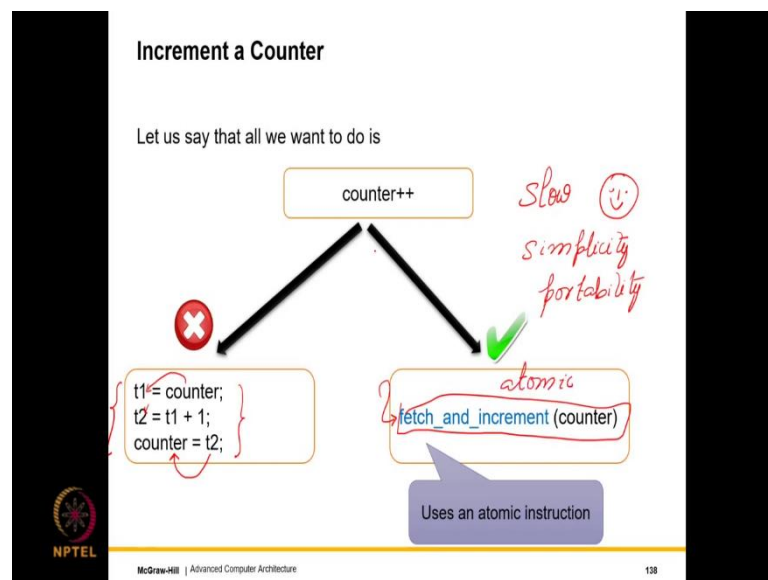
(Refer Slide Time: 32:46)



So, what is the solution? So, let us go back to the drawing board and say that look the memory model is great I totally understand it, coherence is great I totally understand it and I totally understand SC RC PC and everything else you have thrown at me, but at the end of the day I do not care.

I want to write a piece of parallel code which will run on all machines regardless of their memory model it will just simply run and it should be simple to program and I do not want to learn a memory model I am not interested. So, and I am not a black belt programmer I do not care that much about performance. So, I am willing to tolerate a little bit of slow down you kindly give me something which is slow, but reduces my headache and runs everywhere.

(Refer Slide Time: 33:45)



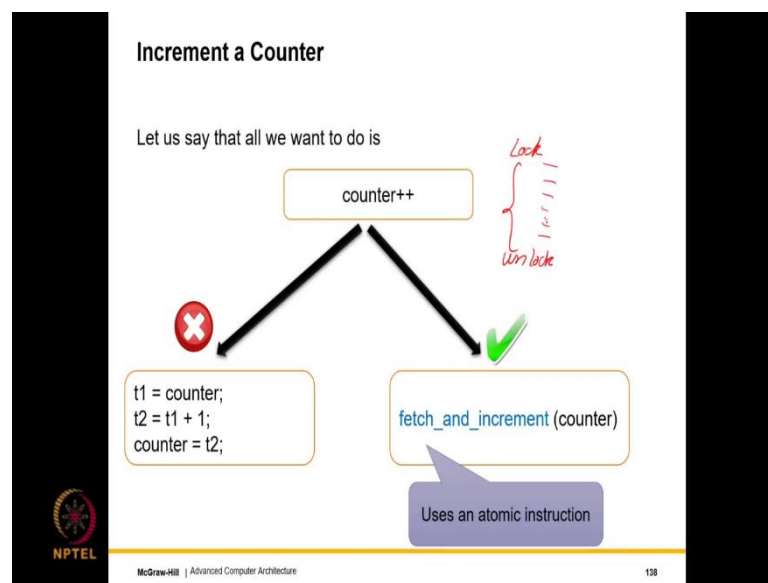
So, basically performance for me is not that become a concern. So, I am ok if it is slow or I am ok if it has a lot of fences, but I just need simplicity and portability. It has to be simple and it has to be portable in the sense I write it on an ARM machine the same code should run on an Intel machine and vice versa.

So, let us look at the same logic or same code `counter++`, see if I were to break it down into three assembly statements and I am just writing it in the three analog of assembly I am not writing the assembly. I will read counter put it into a register, increment the register and then write the register back to the counter variable. So, this is how risk I say would do

it by breaking into three instructions and if I cannot run multiple copies of this code in parallel because the output will be wrong and we have already seen this.

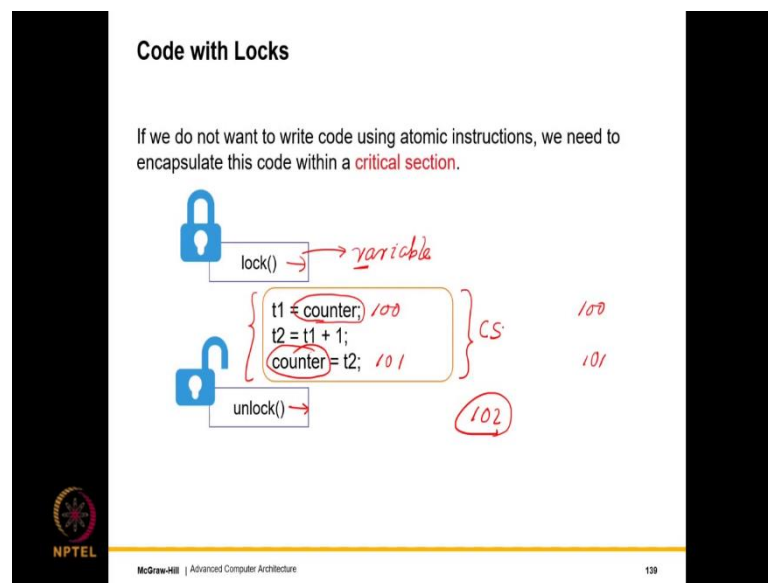
So, what I can do is? I can use an atomic instruction like fetch and increment. So, this is guaranteed to work it will take counter replace it with counter + + and the entire thing happens appears to happen instantaneously. So, this is atomic. So, I have no problem at all instead of using that heavy piece of code I will just use this. I could do this because this is simple.

(Refer Slide Time: 35:13)



But, if I had a piece of large elaborate logic I actually could not have done this, then you would have argued that I need to have lock and lock and then unlock mechanism to ensure that this piece of code executes correctly which is something that I will show in the next slide.

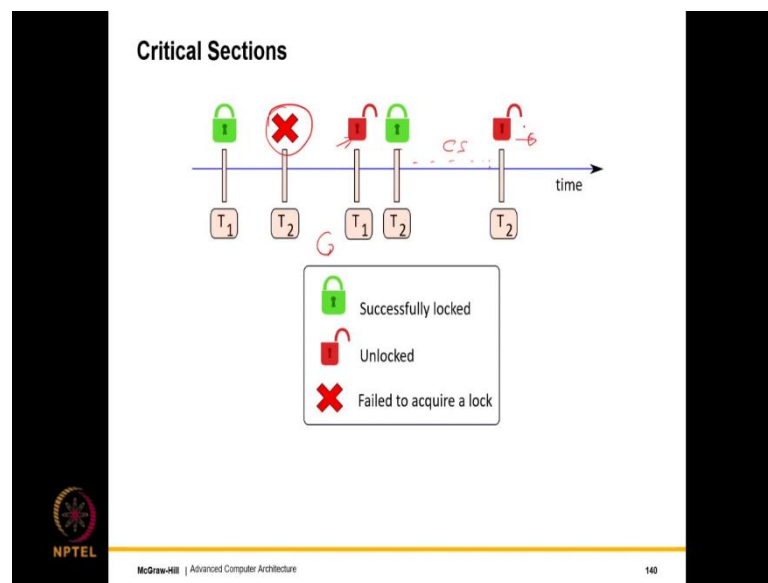
(Refer Slide Time: 35:31)



Say if we do not want to write code using atomic instruction that in many cases will not be able to do the logic might be very complicated. We can encapsulate them in what is called a critical section which will begin with a lock we have seen how it works which will end with an unlock we have seen how that works and then we will just execute will have a lock and an unlock and the code in between is called a critical section and every lock has a lock variable associated with it.

So, as long as we lock the lock variable nobody else can get in and further more if any other place counter is being used in some other context, if that is also locked with the same variable then that will also ensure that nobody can get in and update the value of counter and cause an error basically.

(Refer Slide Time: 36:29)



So, what are we doing? So, what we are doing for this critical section is if you have two threads T₁ and T₂; T₁ locks T₂ tries to acquire the lock it is not able to acquire, T₁ does not unlock.

So, T₂ keep spinning as we have seen, then T₂ gets the lock it executes for some time and finally, T₂ releases. So, this is the typical way that we work where one thread acquires the lock executes the critical section other threads keeps spinning. Once a thread relinquishes the lock some other thread gets it; it gets exclusive control of the critical section executes it again unlocks so on and so forth.

(Refer Slide Time: 37:17)

Why do we need to use locks?

? If there are no shared variables, do we **need** to use locks?

Answer: No!

? When do we need locks then?

Answer: Two blocks of code need to be making **conflicting** and **concurrent** accesses to the same address.

Conflicting accesses → At least one access is a **write**

The slide includes handwritten red annotations: 'W W' with arrows pointing to the first question, and 'R R' and 'R W' with arrows pointing to the second question. A diagram shows a red box labeled 'Conflicting accesses' with an arrow pointing to the text 'At least one access is a write'.

NPTEL
McGraw-Hill | Advanced Computer Architecture
141

So, now the question is why do we need to use locks? Well so, let us start with two philosophical questions. So, assume that we have a large piece of multithreaded code, but there are no shared variables. If there are no shared variables do we still need to use locks, the answer is no. The reason is that if there are no shared variables so, we will not have the kind of concurrency bugs that we have actually seen.

So, why were the concurrency bugs happening? They were happening because we wanted this piece of code to execute atomically, but it was not. So, let us say we will read two threads will read counter to be 100, then they will set it to be 101, but 101 should not be the final state; the final state should be 102. So, this is happening only because the counter variable is shared, if it was not shared there would have been no issue.

So, this is number 1 if there are no shared variables we do not need to use locks. So, when do we need locks then? We need locks when two blocks of code make conflicting and concurrent accesses to the same address the same address is the same variable. So, what are conflicting accesses? They are a pair of accesses where at least one of them is the write because if there are two read operations we do not really care in which order they are happening and they will not cause any correctness issues. So, we do not really need to care.

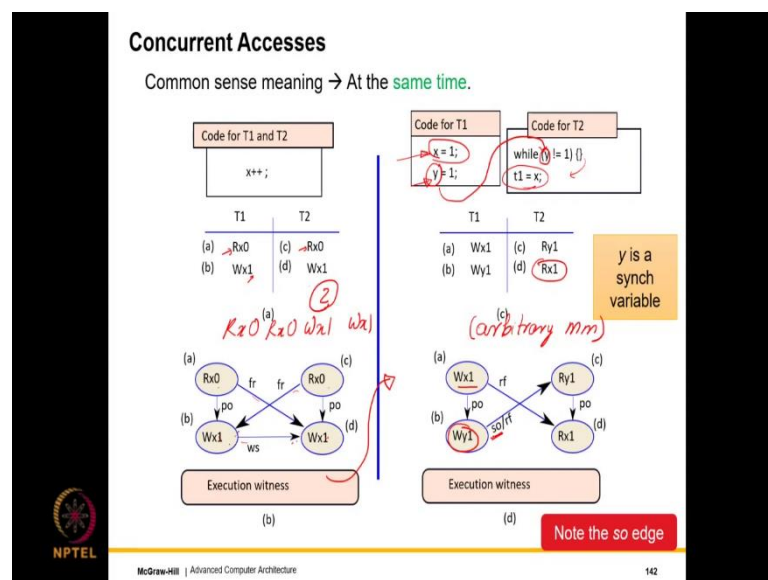
The problem comes when one of them is a write, if you go back the problem is only coming because we are updating the state of counter. If we did not update the state of the counter,

there would have been no issue in just reading the value of the counter we did not need a lock and unlock function.

The problem is coming because we are updating the value of counter of the counter variable and that is where we need this mutual exclusivity of the critical section. So, basically the codes of course, have to access the same address and make conflicting accesses which means at least one of them is the write, if both are a read there is no problem at all, one of them is the write like it is like this or both of them are writes.

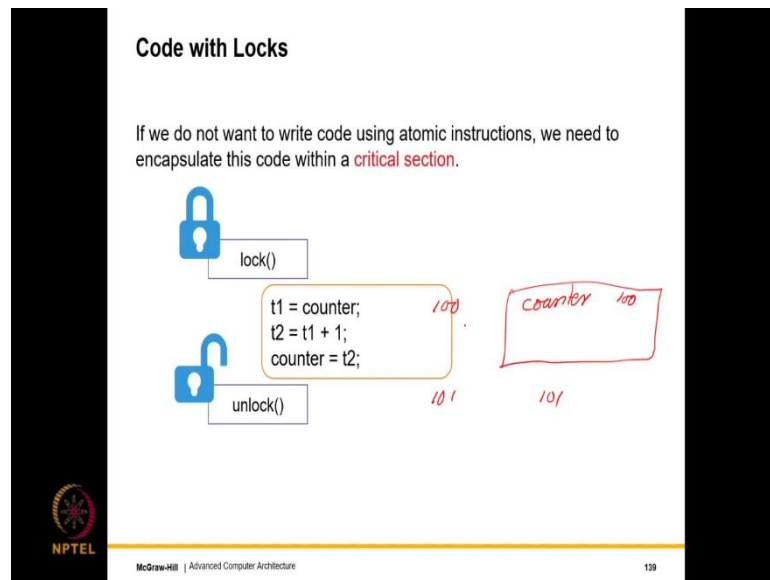
So, in that case, it is a clearly conflicting access because the final state is dependent on their order and also it is a large block of codes. So, we might have many reads and writes the entire thing will not be correct unless we have a critical section.

(Refer Slide Time: 39:54)



What is concurrent? Concurrent means at the same time so you go back and let us look at informally; informally what concurrent meant over here.

(Refer Slide Time: 40:07)



Is that two threads execute at the same time both read counter at the same time in the sense they read both the values to be 100.

Then ultimately they just write 101 since there is an overlap in their execution they in a certain sense are concurrent informally defined and that causes all the trouble. So, the common sense meaning is at the same time, but I think we can do better. So, let us look at the code for incrementing the counter. So, let us say that we read the counter to be 0, we read it to be 0 and we update it to 1.

So, what are we doing $R \times 0$ to $W \times 1$ let us add a program order edge and $R \times 0$ to $W \times 1$. So, this is the execution witness over here we again add a program order edge ok and then what you see is that we have two r edges like this and we have a write serialization edge because let us say this write happened after this write. So, this is pretty much the execution witness that you see over here and this does not really convey a lot of information.

And all that it tells us is that look the execution witness is fine, but in spite of that the behavior is not satisfying us. So, there are no cycles in execution witness as you see. And also the execution witness if we actually see it is sequentially consistent in a sense we can order the operations like this. So, it is an SC, but even if it is an SC we are still not happy because we do not want the final value of $x = 1$ we want it to be 2.

So, clearly we need some tool or some additional rider on the execution witness to actually make it more expressive such that this particular case of this concurrent conflicting access this can be taken care of. So, let us now look at a slightly different kind of program such and this might give us an insight. Here we want to transfer the value of a variable from thread T 1 to thread T 2. So, what we do is we set $x = 1$ and then we set. So, let $x = 1$ be the result of our computation.

Then we set $y = 1$ and then we read the value of y . So, as long as $y \neq 1$ we just keep looping and when it is 1 we come down and then we read the value of x . So, if we have atomic writes then what will we actually see? So, what we will actually see is that we will see the value of this to be $R \ x \ 1$.

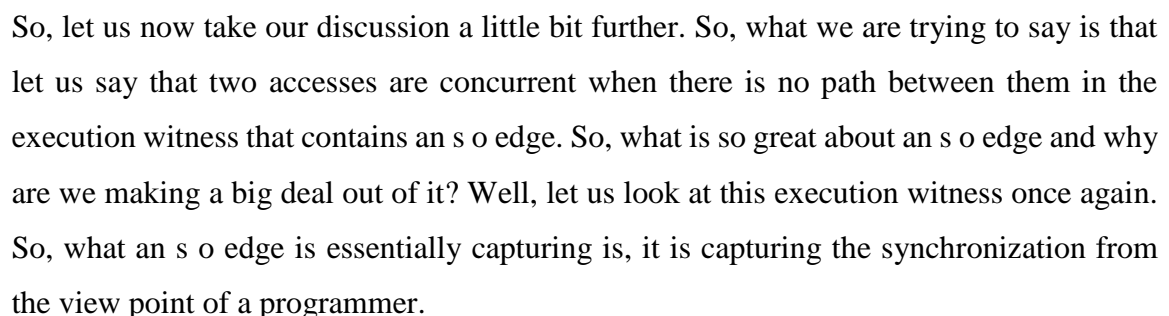
But of course, we need a few program orders to hold as well and only then we are going to see. But, there is also an important point that I would like to mention here assume that y is a synch variable in the sense that y over here is updated only using atomic instructions the kind that we have seen before.

Say in this case what you will actually see is that we will write $1 = x$ there will be a program order edge to $W \ y \ 1$. So, so let us for a time being say that this is for an arbitrary memory model I do not really care. So, then what we will see is because y is the synch operation in all memory models you will have the $W \ x \ 1$ to $W \ y \ 1$ edge then you will also have a synchronization edge and an s o edge this being the most important between $W \ y \ 1$ and $R \ y \ 1$.

Because we are reading so, recall that all the synchronization operations between them are sequentially consistent. So, we will have an s o edge between them and then we will have a program order edge between $R \ y \ 1$ and $R \ x \ 1$ and given that there is a synchronization edge so, $W \ y \ 1$ would have expected there is a fence also inside it the $W \ x \ 1$ would have completed.

So, regardless of whether $r \ f$ is global or not we are guaranteed to read $R \ x \ 1$ at this point of time. So, what we see is that in this execution witness we are able to see some patterns if we look hard enough there is some pattern that guarantees that look regardless of the memory model if $x = 1$ here x will be read as 1 over here and our exact memory model does not work that is not important and the reason is that y is a synchronization variable.

(Refer Slide Time: 45:57)



So, the programmer wants that before an access to this synchronizing variable y all previous accesses are done. Then another synchronizing access is made over here and this has to strictly be after setting $y = 1$ and all subsequent actions are after that. So, this is what the programmer wants and the way that the programmer is enforcing this wish of her is basically via having these synchronization instructions and these synch edges these s o edges.

So, the important point over here is that if let us say between any two conflicting accesses to the same variable we have an s o edge it basically means that the programmer's intent is being captured that we make one access then we pass through a synchronization and make the second conflicting access.

So, this means that the conflicting accesses in a sense are controlled and controlled by whom control by the instructions inserted by the programmer and this means that this execution is going according to the wishes of the programmer that is broadly speaking an informal representation often of the situation.

So, we are trying to slightly formalize it. So, we say let us consider an execution witness and let us say within the execution witness two accesses are concurrent which essentially means that there is no synchronizing access between them. So, consider a write and a read to the same address. So, let us say there is an r f edge between them. So, we say that it is concurrent because primarily it is not passing through any programmer specified structure it is not passing through any kind of a synchronizing access like a lock or unlock.

So, as far as we are concerned we can have another execution where we could very well have this and because it is not controlled and let us say if other variables are dependent upon this order they also need to reflect this order and that might not happen. And this is exactly why we want all such interactions between accesses to the same variable one of which being a write passed through the synchronization edges which essentially are put to capture the intention of the programmer.

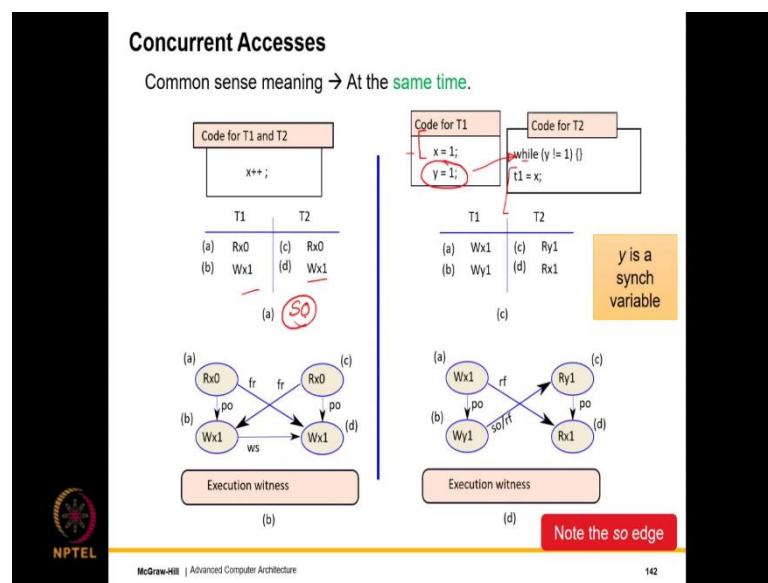
So, a critical section is one example of such a synchronization edge, but it is not the only example. Here we have a critical section we unlock and then another thread enters the critical section and there is an s o edge. So, all of the accesses over here are strictly before all of the accesses over here and they happen before just because of the s o edge and this ensures that let us say for example, we are setting $x = 1$ and we are setting $y = 1$ and then here we are setting $x = 2$ and we are setting $y = 2$.

So, it is never the case that we observe some kind of a partial state in the sense that for all other threads they will either observe they will observe $x = y$ all the time and because they cannot make any other access they cannot access x and y without acquiring the locks and once you acquire them we will always see $x = y$.

So, this is how the synchronization edge and critical sections capture the programmer's intent? However, it is possible that while writing the program the programmer might make a mistake and there might be a pair of conflicting and concurrent accesses to the same regular variable it is a non-synch variable. There will be conflicting accesses which are also concurrent by this definition.

They are set to constitute a data race and the execution over here is actually a data race because we see concurrent conflicting accesses to this to the variable x and there is no programmer intent anywhere. So, there is no effort to synchronize them and so, that is exactly why we are not seeing? So, we are trying to increment a counter but the final result is wrong because the program has not added any synchronization. So, what we should see is the final value should be 2, but in this case, it is 1 which is not correct.

(Refer Slide Time: 51:35)



That is primarily because there are no s o edges in the picture. So, data races are bad things they are essentially bugs and they prevent the successful execution of a program. So, now let us look take a deeper look at data races and see what they actually mean. So, here I would like to slightly specialize this definition.

Because there are several things and this is a very very theoretically deep area. A lot of the proofs are there in the book, but I am deliberately going to gloss over them and present them in very 20000-foot level informal fashion. Primarily, because I do not want to bore

you or bog you down with all the theory. So, the book is there for it and I would request all of you to read these sections pretty thoroughly.

(Refer Slide Time: 52:33)

Data Races

Two accesses are said to be **concurrent** when there is no path between them in the **execution witness** that contains an **so** edge.

Data Race

A pair of **conflicting** and **concurrent** accesses to the same regular variable constitute a **data race**.

! If a piece of code does not have data races, what does it mean?

Execution (mm) Program

data race mm

NPTEL

McGraw-Hill | Advanced Computer Architecture

143

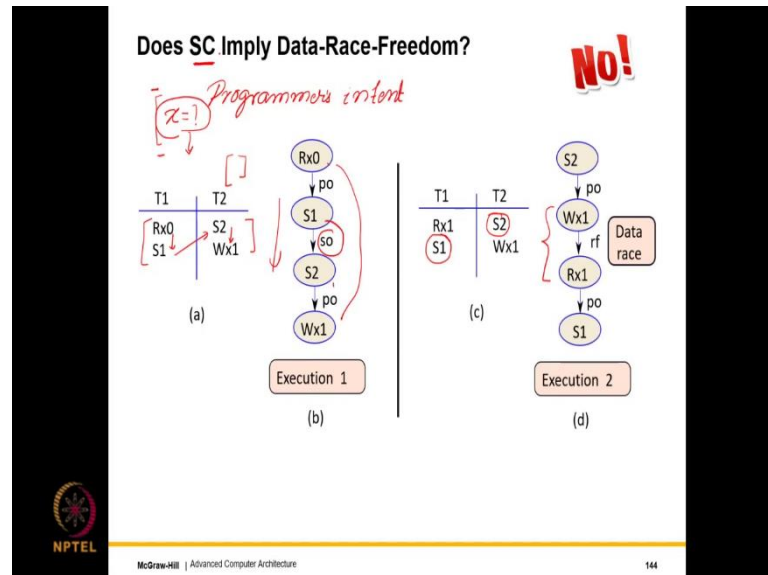
But, I will explain the broad idea. So, now, I want to differentiate here between two concepts; one is the data race in an execution other is the data race in a program. They actually mean different things, but we will see that they are beautifully connected. So, that is the beauty of this entire theory over here. One is the first concept is the data race in a given execution. So, what we do is we create an execution witness for it and we look at all pairs of conflicting accesses to the same regular variable.

If they are concurrent we say that look this execution witness has a data race and the other is so, this is where a data race is the function of an execution. So, for a given program we can have a lot of execution some might display data races some might not. So, that is why we have a lot of data race detection tools and some sometimes some executions have data races, sometimes the executions do not have data races as we saw in the in this example with sorry we will see one example later that shows this.

The other is that we will try to determine what actually is a data race preprogram and there we will need some theoretical tool. So, this is not very clear because the execution witness by definition has adjust that belong to a certain memory model. So, the data races we will be you will be tempted to think that they have something to do with the memory model, but actually they do not. But, I would like to go slow. So, I would like to introduce theorem

after theorem and I will only explain the main result not the proof because as I said this entire stuff is quite deep.

(Refer Slide Time: 54:20)



So, let us now try to answer some basic questions. So, where are we here we said that look data races are a bad thing. Why are they bad? Because a conflicting concurrent access in a certain sense is not appearing to capture the programmer's intent because whenever parallel code is being written and we have a conflicting concurrent access it basically means that it you can read write variables in the same variable in any order.

So, then the final outcome is questionable. So, that is why the programmer's like to regulate any conflicting access to the same regular variable that they will say that you want to modify x no problem go ahead and modify, but then before doing this do something like acquire a lock, after I do something like release the lock next time you want to access again go via this mechanism.

So, this will ensure that at least a set of variables or maybe if you have read or written it. So, a set of accesses and a set of accesses here they internally make sense the program on a whole makes sense. So, that is why we define data races because we thought that they are a good tool that captures the programmer's intent as far as a parallel program is concerned. Otherwise, we saw that if with had data races even a simple counter update by multiple threads that was turning out to be wrong, but the moment we introduce synchronization variables it turned out to be correct.

And so, now, let us take it further. So, let us answer a few questions does sequential consistency imply data race freedom in the sense let us take a program and let us say that we run it on an SC machine will that imply that there are no data races, but take a look at this piece of code. Since this piece of code we have $R \times 0$, $S \ 1$, $S \ 2$ and $W \times 1$. So, this is sequentially consistent and as you can see there is an s o edge between these two accesses. So, this execution is data race.

Now, let us consider one more. So, we have not assigned a meaning to the synchronization accesses $S \ 1$ and $S \ 2$, but if let us say they happen to execute like this where $S \ 2$, $W \times 1$, $R \times 1$. So, I write and then i read and then I have the second synchronization access. So, here we have a data race. Why do we have a data race over here? Well the reason we have 1 is because there is no synchronizing edge between this write and this read and this to us represents a data race which is something we wanted to avoid.

So, SC does not imply data race freedom because as we can see this is SC machine, this is a legal sequential execution, but there are data races. So, data race is something which is most likely at least it appears at this point something stronger than SC. So, we will go into watch stronger and weaker means, but at least sequential consistency does not give us data race freedom. So, what does let us see.

(Refer Slide Time: 57:56)

Does Data-Race-Freedom imply SC?

Yes!

Refer to the book for the detailed proof.

Salient Points

- If there are two **conflicting** accesses, there will be an so edge on at least one path between them in the execution witness.
- They will thus be **ordered** by the so edge.
- Let us add all the SC edges to the execution witness.
- A **cycle** implies that there is a **cycle** between **sync operations**.
- This is not **possible**. **Sync operations follow SC.** = **po + atomicity**
- Proof by contradiction.

(powerful) Exec = po + atomicity

NPTEL

McGraw-Hill | Advanced Computer Architecture

145

Let us say that I have a program which is data race, we will see what this means in slight detail in next few slides, but let us say that all executions of the program are data race

column does it imply that all of these executions are in s.c. So, so let us say that we take a program and we just execute it without caring about the memory model. So, we will we will come up with a bunch of executions I claim that all of these executions are in SC.

So, knowing we have all this actually happens to be correct. So, this is a fantastic result which is which I would say is arguably the most important theoretical result in all of computer architecture some might not agree with me, but at least as far as I am concerned this is the most important result important with a capital I result in computer architecture which says that look if all of your executions or data race free then data race freedom does imply SC.

And of course, I am not all the bells and whistles I am not mentioning of what exactly the other constraints and so on, but they are all there in the book.

But, we are looking at it broadly from an angle where you not interested in the corners. So, the salient point is that look if I if let us say all accesses conflicting accesses for the same variable across threads have an s.o edge between them so at least one path with an s.o edge between them. there will be an s.o edge at least one path which means they will be ordered by a synchronization edge.

Now, let us do something let us take an execution witness regardless of the memory model and add all the SC edges. So, what are they? So, the SC edges will be basically make r.f.e global and all the program order edges will be global and then let us see what happens to the execution witness, no problem. So, if the execution witness is acyclic it means that the execution is an SC. Let us assume just for the sake of assuming that there is a cycle then it is easy to prove that if there is a cycle then there will be a cycle of synch operations as well.

So, there will be a cycle that contains synch operations which happens before relationship between them. So, this is a synch operation that will be happens before relationship a global happens before relationship S_1, S_2, S_3 so on and so forth and ultimately back to S_1 .

We claim that this is not possible this is not possible because that the claim is that synch operations by themselves follow sequential consistency and the reason that we say. so, is

basically because sequential consistency is program order plus atomicity. Synchronization operations do follow program order because they have a fence within them.

And atomicity because they are atomic operations. So, they follow atomicity that is obvious. So, they follow SC. So, they cannot have a cycle between them. So, there we have a contradiction. So, this is the broad outline of the proof. So, the proof basically says that look do not care about the memory model or even memory model as such is not important. If you are guaranteed not to exhibit any data races even in the most even with the most restrictive memory model which is SC it is guaranteed that you will not show any data races.

Then this does imply that the execution will be sequentially consistent. So, what would this exactly mean? this would exactly mean that whenever there are two conflicting accesses to the same variable you do something in a sense you enclose them in a critical section where we have one synchronization operation at the beginning, one at the end again one at the beginning, one at the end such that any order between them is passing through an s o edge like this.

If that is the case, let us not worry about the memory model any time we execute this regardless of the underlying model the execution will always be in SC. So, this is very very powerful this basically says that look the memory model should be designed by the architects it can be as weak as possible to enable performance. We simply need to add as many fences as is required fences and atomic instructions with this synchronization capability as is required even I did not like to minimize them because they are expensive.

But, as long as we are avoiding data races we are good. So, we can reason about our program in terms of sequential consistency which we wanted to do originally we never liked parallel executions we always wanted a parallel execution to be somehow equivalent to a sequential execution one after the other, but we were not able to do that with our execution witness base method because our sequential execution was not legal. But, in this case, if let us say it is data race free we can equate it to a legal sequential execution and we can reason about it.

So, it will become very easy to write parallel code because as human beings we love to think sequentially and we do not think in parallel. So, once it is an SC we our proofs and our tools everything start working and our reasoning becomes very very easy and simple,

but this is not the only theorem. So, where are we? SC does not imply data race freedom, but data race freedom implies SC.

(Refer Slide Time: 64:26)

What does having data races imply?

Theorem

If we have a **data race** in a program, we can construct an SC execution that also has a data race.

Proof → refer to the book

? What does it imply?

If an automated tool **cannot construct** an SC execution that has a **data race**, then it means that the program is data race **free**.

Method to detect data races

Handwritten notes: "data race" with a downward arrow, and "SC exec. data race" with a rightward arrow.

NPTEL logo, McGraw-Hill | Advanced Computer Architecture, 146

What else this is one, the other is that let us say I have a data race what would that imply.

So, we are now in the previous slide we looked at the question that if I do not have a data race what happen if you do not have a data race well that is great you are sequentially consistent. But, now the question is if I have a data race then what? If I have a data race, then there is another theorem which is again a beautiful theorem and I would also rate this very highly in our computer architecture world because typically unlike the mathematical sciences we do not have a lot of theorems in computer architecture.

And the number of theorems that we have are far and few, but a few that we have are reasonably powerful. So, this is one, this is the second one. So, it says that look if you have a data race in a program you can construct an SC execution that will also exhibit a data race, which means that under any memory model if let us say you have a data race in a sense it is possible to have two accesses conflicting accesses to the same variable without an s o edge between them. You can then create an SC execution that will also have a data race.

So, proof again refer to the book this kind of an involved proof, but what does it actually imply. So, this implies something significant. It implies so let me write it down data race

in a program implies that SC exec with a data race. So, I can kind of flip this around and consider the contra positive well let us say let us take a program and let us let us consider all of it sequentially consistent executions if let us say there are billions of them we cannot do it manually, but an automated tool can.

So, it can construct all of these sequentially consistent executions, if none of them exhibit a data race we can say that the program does not have a data race. So, this is very important I do not mind repeating this twenty times because this is capturing a large part of our argument.

So, hear this again. So, what do? we know that we say that look under some memory model if you are seeing a data race you can always construct an SC execution with a data race. Consider the contra positive of this, in contra positive what we do is that we reverse the sign of the implication.

(Refer Slide Time: 67:26)

The slide is titled "What does having data races imply?". It contains the following sections:

- Theorem**: If we have a **data race** in a program, we can construct an SC execution that also has a data race. (Handwritten note: $\alpha \rightarrow \beta$)
- Proof → refer to the book**: Accompanied by an icon of an open book and handwritten notes: $\sim \text{data race}$, \uparrow , $\sim \text{SC exec. data race}$.
- Question**: What does it imply? (Indicated by a red question mark icon). The text below says: If an automated tool **cannot construct** an SC execution that has a **data race**, then it means that the program is data race **free**. (Handwritten notes: $\sim \text{mm}$, bug , Prog in a circle).
- Method to detect data races**: A red box at the bottom.

The slide also features the NPTEL logo in the bottom left corner and the text "McGraw-Hill | Advanced Computer Architecture" and "146" in the bottom right corner.

And then what we do is that we say that if. So, this stands for the not right see if there is no SC execution with a data race it implies that the program does not have a data race. So, in this case, data race becomes the property of a program and what it basically implies? or what it basically tells us is that if an automated tool cannot construct an SC execution with a data race. So, an automotive tool can always see billions of executions and see if those executions have data races or not.

But, assume that it cannot construct an SC execution with a data race, then it means that the program is data race free. In other words, we are defining a data race as also a property of a program and it basically says that look consider all of its executions notably its SC execution, if there is no data race then the program itself is data race free.

So, what you should do is that you should write programs that are data race free they will run on all memory models and that through in a sequentially consistent manner. So, basically the point is the underlying architecture is free to determine its memory model.

You write your program do not care about the memory model and then just ensure it is data race free run it on the architecture it will run, but what happens is sometimes when we write large programs we can make bugs we can have bugs we can make mistakes these will be concurrency bugs notably we will have a data race. When we have a data race it is possible that maybe in 100 executions. we do not see a problem, but in 100 and first execution or the 1000 and first execution we might notice the problem that is where we need an automated tool.

And there are many such tools available which will basically sequentially enumerate all the sequentially consistent executions search them for data races and finally, verify that a given piece of code is data race free. So, we will also look at other approaches other kind of semi-formal approaches later on, but the basic idea is that it is possible to verify with an automated tool whether a program has a data race or not it is very if it has one it will definitely show up in a sequentially consistent execution.

(Refer Slide Time: 70:04)

Summary of All the Results

Properties

- SC does not imply data-race-freedom.
- Data-race-freedom implies SC
- Non-SC execution implies data races
- If an automated tool cannot construct an SC execution that has a data race, then it means that the program is data race free.

Moral of the story → If there are no data races, the memory model doesn't matter

- Programming languages need to define data-race-free memory (DRF) memory models that provide synchronization primitives.
- Programmers need to use them to write properly synchronized programs.
- This means that all accesses to shared variables are protected with critical sections such that there are no concurrent, conflicting accesses.

Handwritten notes: C++ Java, L, U, L, U

So, now it is time to summarize what all we have learnt? So, what we have learnt is that data race freedom is good because it helps us capture the intent of the programmer in. so, far as conflicting accesses to the same variable is concerned are concerned.

SC as such does not imply data race freedom, but the brilliant result is the data race freedom regardless of the memory model implies SC which is great it allows us to write one program and run it everywhere. I can take the contra positive of this and say non SC execution implies data races right which would just be the same thing.

And the next theorem is that if an automated tool cannot construct an SC execution that has a data race, then it means that the program is data race free or alternatively if the program has a data race then you will definitely see a data race in an SC execution. So, the moral of the story is that if there are no data races you are good you have essentially written a sequentially consistent program, but of course, you have added a lot of synchronizing instructions throughout the program which will reduce your performance you should minimize them.

But, assuming that is done the memory model does not matter. So, what should be programming languages do; programming languages also define a memory model for example, C++ 11, C++ 17 do they do define a memory model and they also define these synchronization primitives and they tell the programmer how to write code which is data race free.

So, almost all modern languages C + +, Java and so on have their memory models these are software memory models, where they tell the programmer? what they need to do to pretty much right code which is free of data races. So, we have provided these atomic synchronization operations and programmers are expected to use these primitives which the programming language provides to write what are called properly synchronized programs? And properly synchronized programs are one this is one method of implementing a data race free program.

Here the idea is that whenever I access or share variable or a critical section it protected with a lock and then after executing the critical section I unlock it and of course, for the same variable I always protect it with the same set of locks even if it is accessed by some other thread it is never the case that two different locks are actually protecting the same variable. So, one variable is always associated with one set of locks that will ensure that I can access it only after acquiring those locks.

So, this ensures a strict synchronization edge. So, if I write my code in such a way where all my shared variables are encapsulated in critical sections in the manner that I described the program is properly synchronized and consequently, data race free and this will further mean that all accesses to shared variables are protected and there are no concurrent conflicting accesses.

(Refer Slide Time: 73:34)

Contents	
1.	Parallel Programming
2.	Theoretical Foundations
3.	Cache Coherence
4.	Memory Models
5.	Data Races
6.	Transactional Memory

arch.
+ OS
+ library

PS
code

performance

NPTEL

McGraw-Hill | Advanced Computer Architecture

148

So, in a certain sense what we have done is that we have built a lot of theoretical foundations, this is primarily for the architecture. Furthermore, this is also for the OS writer and library writer who really care about low level performance and they and these people do not want to write code that has a lot of fences or synchronizing instructions. So, the after all somebody needs to write the code for the locks.

So, if let us say we can omit a few synchronization instructions the performance of our locks will improve substantially. So, a lot of low level programmer's need to be aware of the memory model because they cannot afford that many fences, but any high level programmer should use the idea of data races to essentially write properly synchronized code let us call it PS code where all shared variables are wrapped within critical sections and there are no conflicting concurrent accesses.

So, that is how the code is structured? if that is the case regardless of the underlying memory model the code will work in a sequentially consistent manner everywhere as long as it is data race free, but of course, the downside is that the performance will be low because we will be adding a lot of these slow fence like instructions. So, performance will reduce. So, as oppose to these nice handcrafted libraries where we have few synchronizing instructions.

Because we also use the features of the underlying memory model to establish dependencies. So, there is a trade of over here between simplicity, programmability and efficiency, but we did build up. So, efficiency is more important for the low level programmer's, but for high level programmer's they need not be bothered with the memory models they just need to ensure properly synchronized parallel code.

So, we will take the argument a bit further and look at an even simpler method of parallel programming. So, a simpler than let us say what we have seen up till now which is critical sections and also this open MP code that we wrote that does embody all of these all the discussion regarding critical sections within open MP directives.

So, programmer's typically do not get to see that, but of course, your multiple threads will access the same variable they need to we do have a lock and unlock kind of mechanism. So, we will now make it even simpler and introduce an even simpler programming model which needless to say is data race free and it substantially eases the burden on

programmer's and shifts a large part of the burden to either system software or hardware we will call this transactional memory. That will be the topic of the next lecture.