Advanced Computer Architecture

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

Chapter - 09
Lecture - 28
Multicore Systems Part - IV

(Refer Slide Time: 00:16)

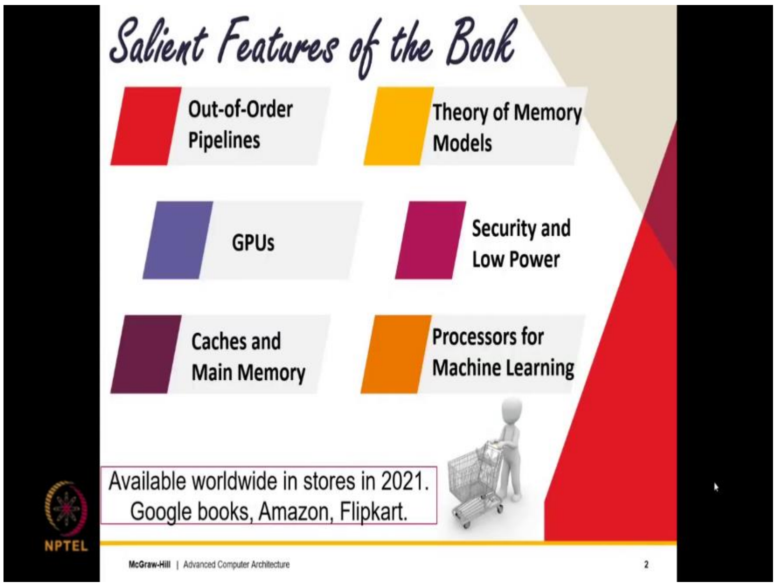
ADVANCED COMPUTER ARCHITECTURE

SMRUTI RANJAN SARANGI

Mc Graw Hill

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: 00:25)

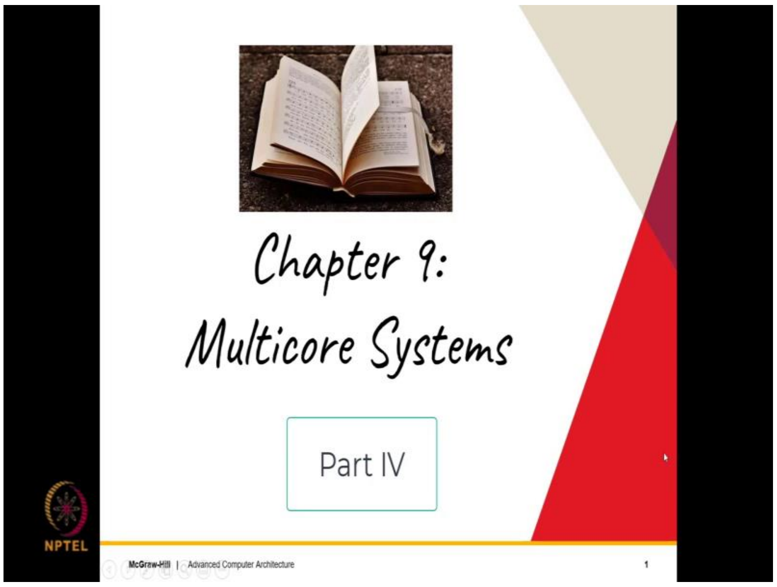

Salient Features of the Book

- 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: 00:31)

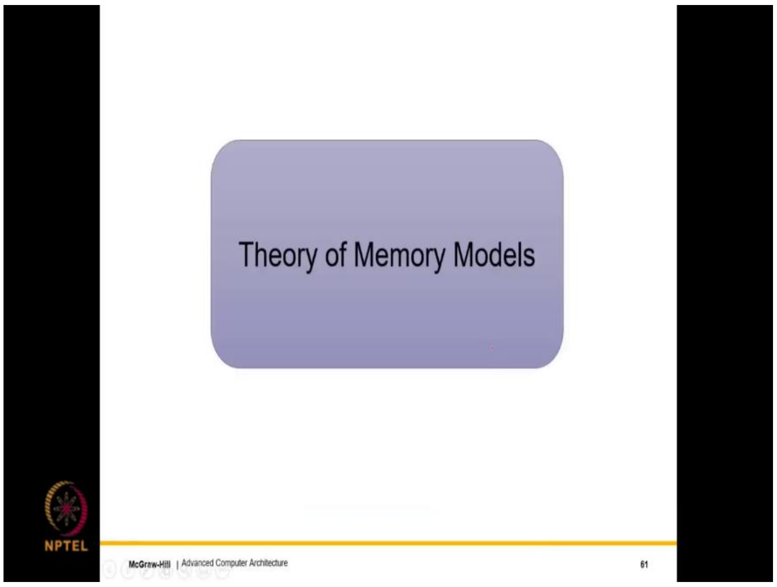


Chapter 9: Multicore Systems

Part IV

McGraw-Hill | Advanced Computer Architecture 1

(Refer Slide Time: 00:39)

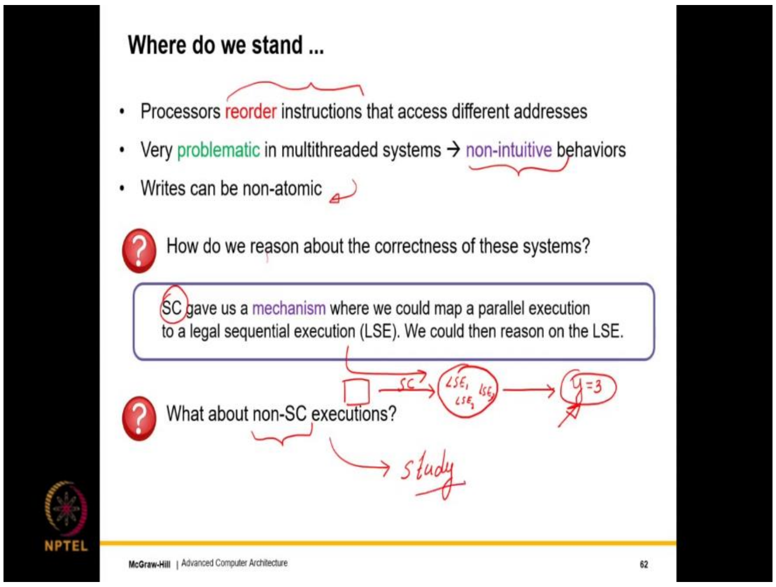


Theory of Memory Models

McGraw-Hill | Advanced Computer Architecture 61

Now, that we have assembled all the ingredients for describing a memory model. The next is to describe the entire space of memory consistency models.

(Refer Slide Time: 00:54)



Where do we stand ...

- Processors reorder instructions that access different addresses
- Very problematic in multithreaded systems → non-intuitive behaviors
- Writes can be non-atomic

? How do we reason about the correctness of these systems?

SC gave us a mechanism where we could map a parallel execution to a legal sequential execution (LSE). We could then reason on the LSE.

? What about non-SC executions?

So, where do we stand right now? So, what we have been observing is that sequential consistency is not practical. processors reorder instructions that access different addresses not the same, but that access different addresses for performance reasons. So, for uniprocessor, it does not matter, but for a multiprocessor it definitely does matter and we have seen very non intuitive behaviors because of this.

So, for example, some pieces of code some reordering which were perfectly normal and perfectly correct in a single threaded execution would appear to be incorrect in a multithreaded execution. furthermore, given the complexities of the pipelines, non locking caches and so on. Writes can be non atomic. In fact, they are non atomic in a lot of architectures such as IBM and RM. So, we need to have a model that also takes this into account.

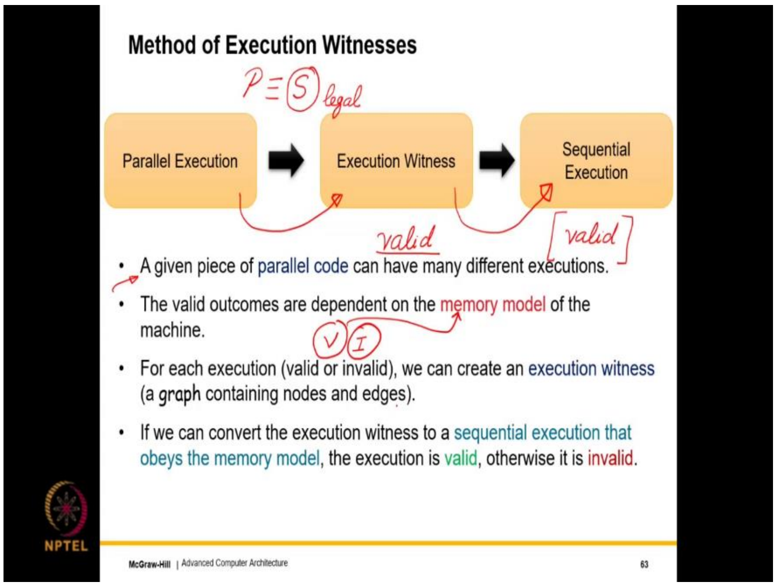
So, now the question is that given a non sequentially consistent system which is any practical system how do we reason about its correctness? Say SC pretty much gave us a mechanism, where we could map a parallel execution to a legal sequential execution establish its equivalence. and then we could say that look given a piece of code these are all of its legal sequential executions.

So, what we could do is that given a piece of code assuming the architecture follow the SC we could create a space of legal sequential executions LSE 1, LSE 2 so on and so forth. If all of them follow a certain property. For example, if all of them after the execution set the value of some variable = 3, so, we can say that regardless of how we execute because a parallel program can have many different outcomes, but that is ok as long as all in all the outcomes y = 3 we might be happy with it.

So, what can we do? well, we can reason that look regardless of whatever happens as long as our machine is sequentially consistent, we can enumerate all the legal sequential executions and we can say that look in all of them the common property is that y = 3 which is what I want.

Hence I am happy. But, what about non-SC executions? Things become way more complicated in this case. So, in this case what exactly we see is that even enumerating all the executions and particularly the sequential executions is hard and many sequential executions also might not be legal. So, let us understand this. So, let us study this.

(Refer Slide Time: 04:14)



Method of Execution Witnesses

P ≡ S legal
Parallel Execution
Execution Witness
Sequential Execution
valid
[valid]
V
I
NPTEL

- A given piece of parallel code can have many different executions.
- The valid outcomes are dependent on the memory model of the machine.
- For each execution (valid or invalid), we can create an execution witness (a graph containing nodes and edges).
- If we can convert the execution witness to a sequential execution that obeys the memory model, the execution is valid, otherwise it is invalid.

McGraw-Hill | Advanced Computer Architecture 63

So, what we want to do is that we want to create a theoretical tool which is somewhat similar to what we did in the case of sequential consistency which basically was to convert a parallel execution or somehow equate it establish in equivalence to the sequential execution.

In the case of sequential consistency this was very easy to do and the sequential execution was very elegant. So, why was it very elegant? Well, number 1, it was legal; number 2, program order was preserved. So, we will not be that fortunate when we consider non-SC models. So, what we will do is from a parallel execution we will create a graph specialize graph called an execution witness and from that we will move to a sequential execution which may or may not be legal.

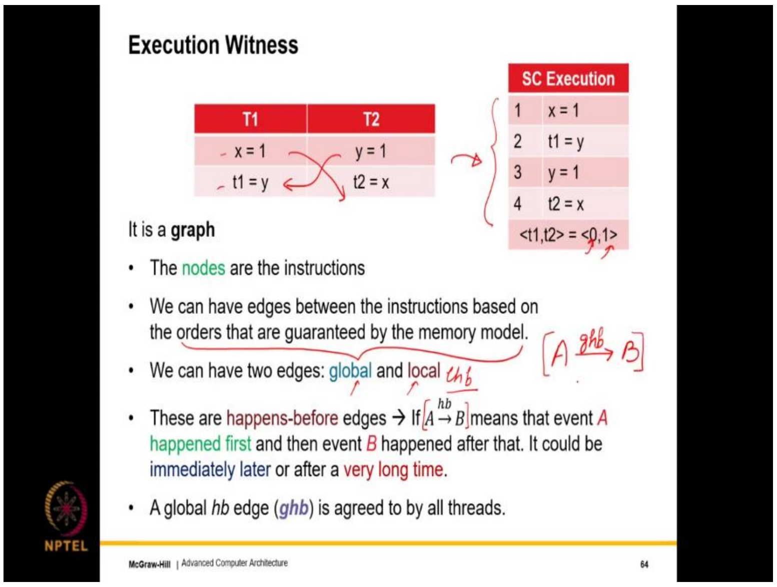
So, we will see when it is legal and when it is not, but this will at least give us several things. One is that if we can construct a sequential execution from a parallel execution via an execution witness what it can at least tell us that for a given memory model it is a valid outcome. So, near fact that we could traverse this we will tell us that look given this execution. So, let us say my question is that given an execution is it valid in the sense how the outcomes valid?

So, what I can do is I can create an execution witness out of it and then try to create a sequential execution out of the execution witness. If I am able to do it at least I can say the outcomes are valid. So, I can use this method to enumerate all the valid outcomes and as long as they follow a certain property I am happy. So, this is slightly more complicated than the simple two step process that we had in the case of SC.

So, in this case, it is a three step process, but we will see that this is very elegant and of course, the two step SC process is a subset of this. So, let us take a look at some of the broad principles. The first is that given a piece of parallel code it can have many different executions and it can have many different outcomes. So, the set of outcomes we can let us say classify them into valid and invalid.

See in valid outcomes we will not see in an architecture if it valids its memory model, but the valid outcomes we will see. So, for each execution as I said we concert an execution witness which is nothing but a graph. And if we can convert it to a sequential execution well it tells us something we will see what?

(Refer Slide Time: 07:05)



Execution Witness

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

SC Execution	
1	x = 1
2	t1 = y
3	y = 1
4	t2 = x
<t1,t2> = <0,1>	

It is a **graph**

- The nodes are the instructions
- We can have edges between the instructions based on the orders that are guaranteed by the memory model.
- We can have two edges: global and local
- These are happens-before edges → If $A \xrightarrow{hb} B$ means that event *A* happened first and then event *B* happened after that. It could be immediately later or after a very long time.
- A global *hb* edge (*ghb*) is agreed to by all threads.

McGraw-Hill | Advanced Computer Architecture 64

So, let us now try to construct an execution witness for an execution that is showed over here. So, here we have x = 1. We read the value of x, we set y = 1 we read the value of y and the SC execution is like this. We set x = 1. We read the value of y * t1. This is write to be 0. Then we set y = 1. We read the value of x and this is write to be 1.

So, what we can do is that we can convert this execution. So, which is also if you look at it is a parallel execution that we can that we have converted to sequential whether nodes are these instructions each one of these is the node. So, this graph will have four nodes. We can then have edges between the instructions based on the orders that are guaranteed by the memory model.

So, I will discuss this part in somewhat more detail, but let us for the time being just flow go along with the flow. So, what I am trying to say over here is the different memory models impose different kinds of constraints on the execution witness. So, we have different kinds of edges. So, we can have two kinds of edges, global and local. So, these are happens before edges, which means that let us say that if I write A happens before B, it means that event A happened first and then event B happened after that.

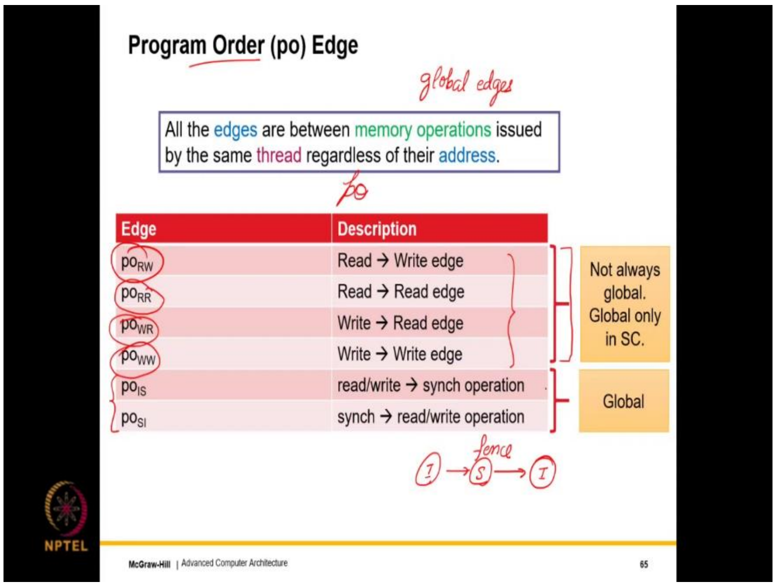
So, it does not matter when event B happened after event A. As I said we never look at exact time over here. So, B could have happened immediately later or it could have happened after a very long time. So, that does not matter. Whether it is just after A or after a very very long time this per se does not matter. The reason is that in all concurrent systems we are only interested in the order which means that it just happened after it or happened before it.

And the timing does not matter and mainly because we have so many nondeterministic events within the CPU that if we start putting constraints on the timing, these relationships will be very complex and also we will not be able to guarantee all of those timings.

So, now, let us come to global and local. Say global happens before edge is added if let us say it is agreed to my all threads. In the sense all thread says, that look event A happened before event B, so, we let us put the let us call it ghb. So, local happens before we will call it lhb local happens before and global happens before we will call it ghb.

So, if we say that A happened before B and this is globally agreed to then all the observers and in this case we have one observer on each thread. So, all the threads will agree with this otherwise they would not. So, this is a difference between global and local.

(Refer Slide Time: 10:22)



Program Order (po) Edge

global edges

All the edges are between memory operations issued by the same thread regardless of their address.

po

Edge	Description	
po_{RW}	Read → Write edge	Not always global. Global only in SC.
po_{RR}	Read → Read edge	
po_{WR}	Write → Read edge	
po_{WW}	Write → Write edge	
po_{IS}	read/write → synch operation	Global
po_{SI}	synch → read/write operation	

fence

$\textcircled{I} \rightarrow \textcircled{S} \rightarrow \textcircled{I}$

McGraw-Hill | Advanced Computer Architecture 65

So, continuing to execution witnesses there are different kinds of edges in the execution witness. So, all of the edges. So, we will consider the program order edge first. So, all of the edges are between memory operations issued by the same thread regardless of the address. So, they can be same address or it can also be different addresses, it does not matter. as long as these operations have been issued by the same thread it is fine.

So, let us now look at the kind of program order edges that we add. So, in general the program order edge will be represented with po in the execution witness and we will add a program order edge between a read and a write again issued by the same thread we will call it po RW then we will have po RR which is read read po write read po write write.

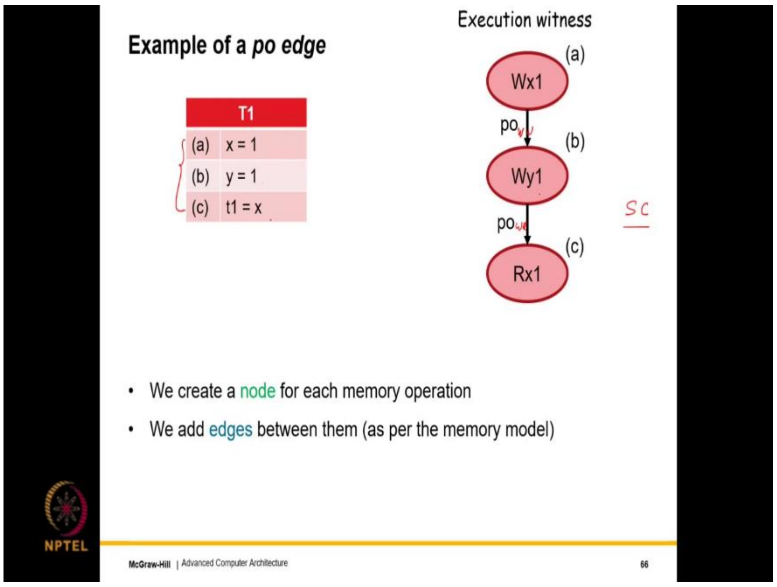
And from any instruction to a synch operation. So, if synchronization operation there is a global order. So, I need to further talk about this part. So, the program order edges as we have been seeing might be there or might not be there in a memory model. For example, the write to read program order edge is often violated. So, this is clearly not global and the execution witness will always contain global edges.

So, this needs to be kept in mind and the execution witness will normally contain only global edges and the write to read edge in most memory models is not global. But of course, in sequential consistency all of these edges are global in because we respect program order all of these edges have to be global in SC, but in all other memory models or let us say regardless of the memory model the following edges are always global which is from any instruction to the synch operation.

So, which means that the instruction fully completes, everybody gets to see the result of it and then the synchronization operation executes and similarly po SI which is again a synchronization operation to an instruction after it. So, we have also seen other kinds of synchronization operations which is not a like a pure fence. In that case these edges will of course, change their character, but any edge from an instruction to a synch operation or vice versa.

Of course, depending upon the type of the synch operation is always global. So, keeping these in mind let us proceed. So, what we need to broadly keep in mind is that we have some of these edges which are these four which are dependent on the memory model. So, they are respective they are said to be respected are global in SC, but the other two are global in all memory models.

(Refer Slide Time: 13:27)



Example of a *po edge*

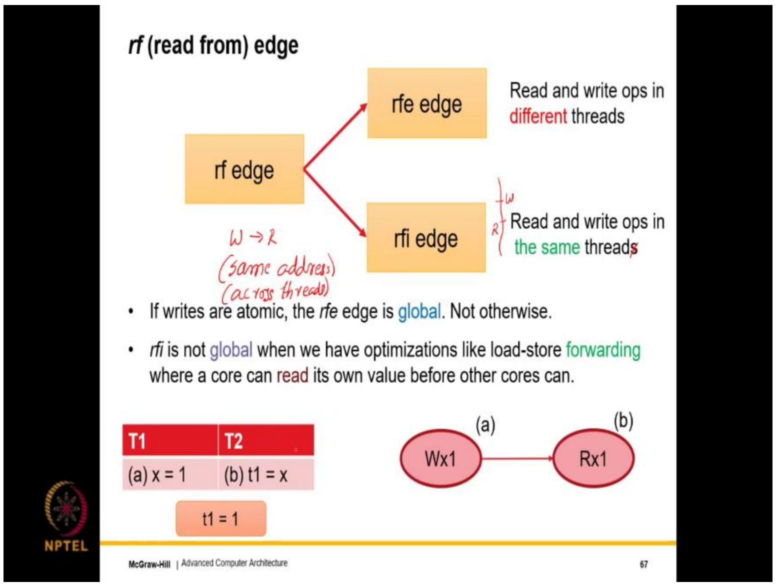
T1	
(a)	x = 1
(b)	y = 1
(c)	t1 = x

- We create a node for each memory operation
- We add edges between them (as per the memory model)

So, let us consider an example. So, nothing is clear without an example. So, consider the following execution. So, where we have a single thread. So, we simply add edges program order edges between the three different operations. If the memory model specifies that these edges are global. So, we have Wx1, Wy1. This is so, this is the write to write edge and then we have a write and a read. So, this is the write to read edge.

So, this execution witness will be valid the programming if the memory model is in SC, where we create a node for each memory operation and we add edges between them as per the memory model.

(Refer Slide Time: 14:20)



rf (read from) edge

rfe edge: Read and write ops in different threads

rfi edge: Read and write ops in the same ~~threads~~

- If writes are atomic, the *rfe* edge is global. Not otherwise.
- *rfi* is not global when we have optimizations like load-store forwarding where a core can read its own value before other cores can.

T1	T2
(a) x = 1	(b) t1 = x

t1 = 1

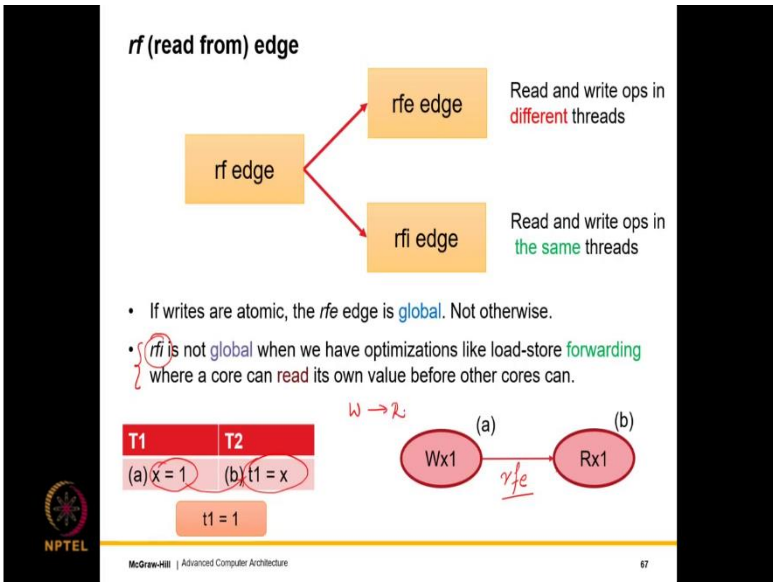
McGraw-Hill | Advanced Computer Architecture 67

So, let us now look at another kind of edge for the read from edge the rf edge. So, rf edge is write to read edge for the same address. Say program order edges were possibly for different addresses, but this is for the same address, but again a program order edge was in the same thread and this can also be across threads. So, the rf edge we can have two kinds rfe and rfi.

So, in rfe edge we have read and write operations in different threads. So, it is a write in one thread and a read from a different thread. And the rfi edge we have the read and write operations in the same thread. So, in the same thread we have the write and then a later read and needless to say there to the same address.

So, let us first look at a small example and then I will explain more. So, in this small example we have a write and we have a read and the final outcome is t1 = 1. So, in that case what it means is that we are writing and we are reading and that is the final and the value of one is received by the other thread.

(Refer Slide Time: 15:47)



rf (read from) edge

rf edge
rfe edge
rfi edge
Read and write ops in different threads
Read and write ops in the same threads

- If writes are atomic, the *rfe* edge is global. Not otherwise.
- *rfi* is not global when we have optimizations like load-store forwarding where a core can read its own value before other cores can.

T1	T2
(a) x = 1	(b) t1 = x

t1 = 1

W → R
(a)
(b)
Wx1
Rx1
rfe

So, we can have a Wx1, Rx1 and we can add an rfe edge. So, rfe edge can be added over here to signify the fact that we are writing and we are reading. So, what we have also seen in the past is that if writes are atomic the rfe edge is global. What this essentially means is that every write has one completion time associated with it then everybody will agree that look the write happened and then the read happened after that.

Say any read that is reading the value of the write all of us will agree, but If this write is not atomic then different threads will perceive this write to happen at different points in time. So, the rfe edge will sees to be global. In the sense not everybody would agree that this write had fully completed.

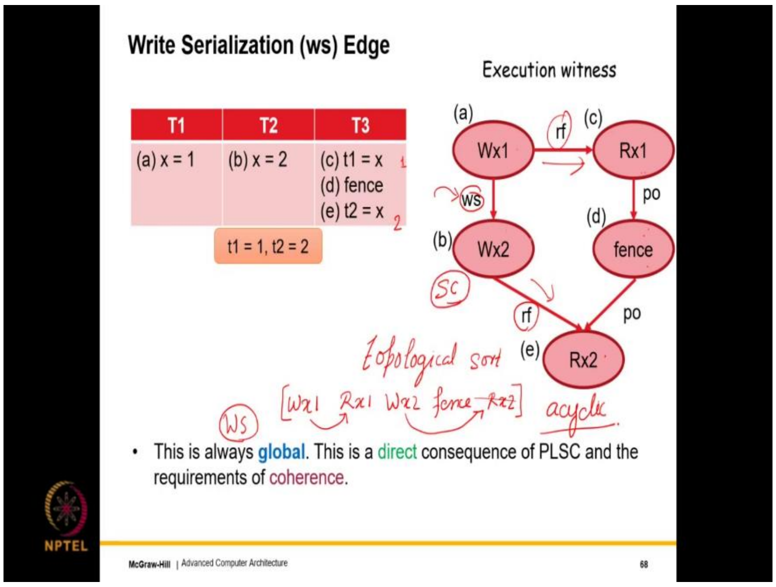
Fully completed means, everybody had seen the value of the write before the other thread actually write right. So, in this case the rfe edge will not be global. So, here of course, an astute reader can ask that look if the write was not done how did we read well it was done with respect to this thread, but not with respect to other threads. Hence, this write was not global.

So, if the rfe edge is global if writes are atomic if they are not atomic the rfe edge is local. Similarly, so, rfi is read from internally. So, this is typically not global. So, even in order pipelines that have forwarding and so on of the kind that we have been seeing.

So, of course, this is store to load forwarding. Say in this case if we have an lsq, rfi will not be global which means that internally we are forwarding the value of a write to a read within the pipeline. The rest of the world is seeing it much later and this is well this is not a globally visible operation.

So, this means that a core can read its own value before other cores can and whenever you read a value early the corresponding edge does not remain global. In the sense everybody does not agree that this is indeed a global edge or let us say the write genuinely happened genuinely completed before the read.

(Refer Slide Time: 18:24)



Write Serialization (ws) Edge

T1	T2	T3
(a) x = 1	(b) x = 2	(c) t1 = x (d) fence (e) t2 = x

t1 = 1, t2 = 2

- This is always global. This is a direct consequence of PLSC and the requirements of coherence.

Let us look at a few more edges. So, of course, now our execution witnesses will become more and more expressive. So, let us look at this particular execution. So, thread 1 sets x =1, thread 2 sets x = 2 and then we read the value of x, there is a fence we read the value of x once again.

So, in this case let us see what is what is the outcome. So, the first outcome we read is 1 and then what we read is 2. So, let us now create the execution witness for it. So, we have the first write Wx1 then we have the second write Wx2. So, since we read in this order 1 and then 2 we add a ws edge which is a write serialization edge.

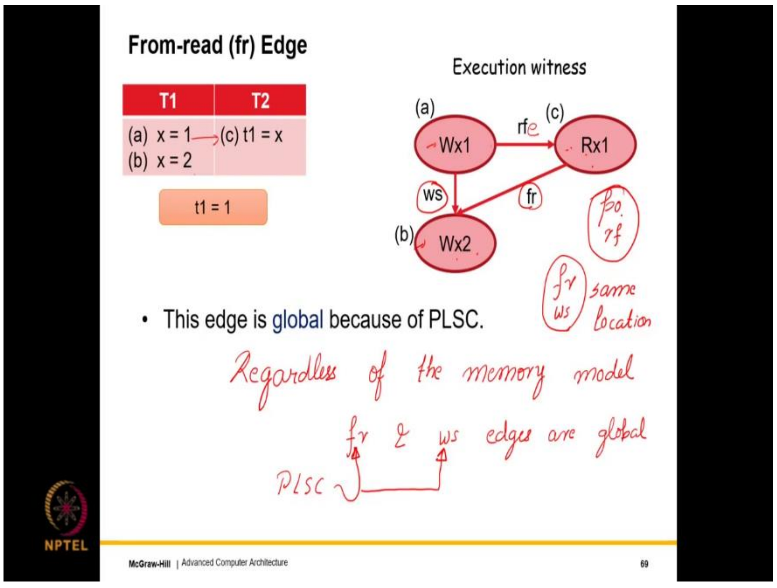
So, as we have seen in our discussion on PLSC this edge is always global. So, always this edge cores. So, this edge is always global and after that the first write is seen by Rx1. So, we add an rf edge going this way. The second write is seen by Rx2. So, we add one more rf edge and then we add a set of program order edges because we have a read and then a fence. So, there is a program order edge and then we have a fence and then a read. So, this is one more program order edge.

So, the ws edge is by definition always global which is a direct consequence of PLSC and the requirements of coherence that ultimately came out of PLSC only. So, what you further see is that this execution witness is acyclic. So, what an acyclic execution witness basically means that if we use the results of the topological sort algorithm, so, the topological sort algorithm basically says that if we have a director acyclic graph or a dag which is exactly the execution witness in this case and if there are no cycles it is possible for us to arrange all of these operations in a linear sequential order.

See as we can see this can be done we can have Wx1 then we can have Rx1 then we can have Wx2 then we can have the fence and then we can have Rx2. So, it is possible to create a sequential order which is sequential execution which is equivalent to the parallel execution in this case and as you can also see this order is also legal.

So, we are expecting program order in along with that this is a legal sequential execution. So, this execution is an SC and further more we saw how we can use three different types of edges namely the ws edge, the rfe edge and the program order edge to create an execution witness.

(Refer Slide Time: 21:42)



From-read (fr) Edge

T1	T2
(a) x = 1	(c) t1 = x
(b) x = 2	

t1 = 1

Execution witness
(a)
Wx1
rfe
(c)
Rx1
ws
fr
(b)
Wx2
po
rf
fr
ws
same location

- This edge is global because of PLSC.

Regardless of the memory model fr & ws edges are global

PLSC

McGraw-Hill | Advanced Computer Architecture

69

Now, we will introduce the last kind of edge which is a from read edge. So, let us first take a look at the execution and then we will see we will go further. So, in this case, what we have is that we have two writes to the same location we write 1 and then 2 and then we read the value of x. We read it to be 1.

So, what we can do? in this case, is that we have two write modes and then we have a write serialization edge between them and the way we create an execution witnesses that we add an rf edge. So, this will be an rfe edge its external write to a read. And the interesting part is that given in the fact that we read 1 and then to the same location we over write it with the value 2, we add the from read edge from Rx1 to Wx2.

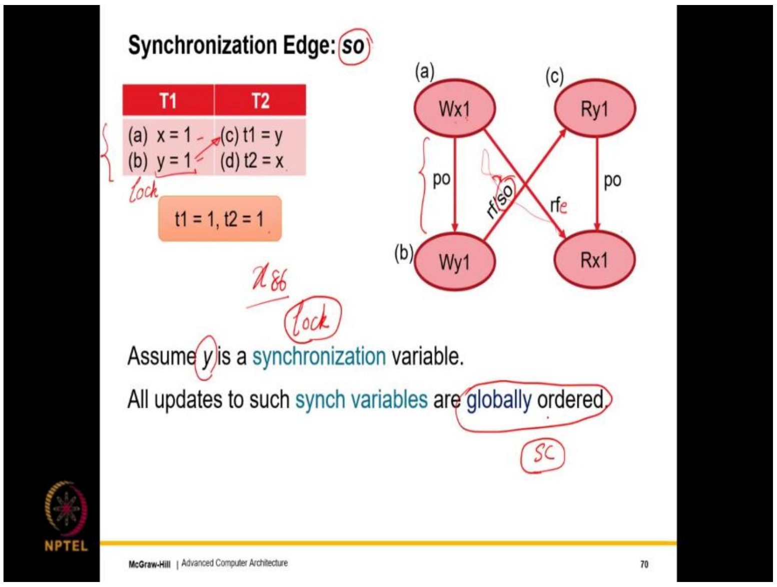
So, as I as we have discussed in the past as well in the execution witness ws and fr edges will always be global regardless of the memory model. So, let me write it. Regardless of the memory model the fr and ws edges are global. Why is this the case? Well, this directly follows from the requirements of PLSC if you go back and even so it does not matter we can have atomic writes non atomic writes does not matter at all.

From the requirements of PLSC we brought in the requirements of coherence that writes a one global order that is visible to all the threads and all of them agree. So, from that it follows that both fr and ws which is from read which is from read means from a read from a read to a write from a read to a write and between writes to the same location we observe a global order.

So, both fr and ws they are accesses to the same location and the execution witness which anyway has only global edges will always have ws and fr edges. Whether it has program order edges or rf edges that is dependent on the memory model and also kind of program order edges with we have terms of synchronization primitives they have different behaviors all of that is fine.

But the key point is that the requirements of PLSC always hold. Why is it that the case? Because we want to maintain the abstraction that there is a single location for every variable, that might not be the case in the sense we might have 20 physical locations for a single variable, but even if that is happening an external observer should never get to know from there ws and fr come.

(Refer Slide Time: 25:07)



Synchronization Edge: so

T1	T2
(a) x = 1	(c) t1 = y
(b) y = 1	(d) t2 = x

t1 = 1, t2 = 1

Assume y is a synchronization variable.

All updates to such synch variables are globally ordered

So, now we have looked at all the edges. So, we would further like to introduce the synchronization edge which is important. So, we will see it is extremely important to tie all of these things together and so, the synchronization edge can be mentioned very easily over here with this example. Let us say we set x = 1 and y = 1.

And let us assume that program order holds between the write to x and write to y. So, let us assume now that y is a synchronization variable. So, we will specialize this in the later lectures on how do we say that one is a synchronization variable, but let us assume for the time being that y is a synchronization variable.

So, the what we will do is that we will introduce the version of a write a version of a store which can be thought of as a synchronization store in the sense that any store of course, with additional directives. So, x 86 for example, allows you to specify the lock prefix. If that is if that is mentioned before a load or a store then the address automatically becomes a synchronization variable.

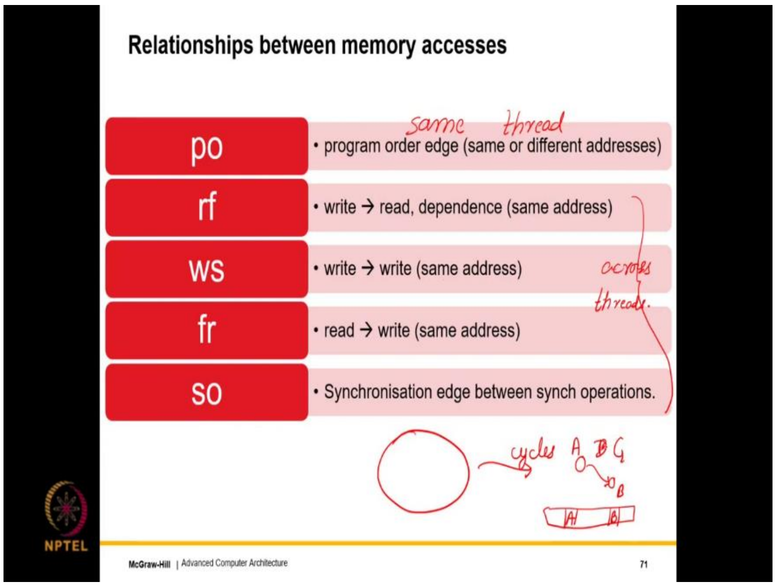
So, let us assume that that is specified. So, this is a synchronizing variable. So, then we read the value of x and we read it to be 1. So, what we do is we add a program model edge between the Wx1 and Wy1. So, recall that we will not have a ws edge because these are different addresses. Then we have an rf edge which is also a synchronizing edge the latter being more important. So, an edge can also be of multiple types. So, an edge need not have a single type, it can have many types.

So, its synchronizing edge and again between reads let us assume that program order holds. So, in this case, this execution witness will be valid because if you add the final edge which is an rf edge and rfe edge to be more precise. Between the write and the read we will not have a cycle. So, this basically means that all of these operations can be arranged in a linear order and you will further more find that it will be legal as well and it will follow program order as well.

So, here what is the assumption that y is a synchronization variable. How did we specify that? Well, let us be slightly vague about it right now. All that I would like to mention is that we will use special synchronizing loads and stores in the place of normal loads and stores.

Say x 86 has the lock prefix. The lock prefix allows us to convert a regular load store to a synchronizing load store. All updates to such synch variables are globally ordered. So, let us assume. So, furthermore many memory models assume that all synchronizing accesses or updates to synch variables are in SC. So, we will make a similar assumption here in our discussion as well.

(Refer Slide Time: 28:30)



Relationships between memory accesses

po	• program order edge (same or different addresses)
rf	• write → read, dependence (same address)
ws	• write → write (same address)
fr	• read → write (same address)
so	• Synchronisation edge between synch operations.

same thread

across threads.

cycles A B C

McGraw-Hill | Advanced Computer Architecture 71

So, just to summarize what are all the edges that we have introduced. So, we have introduced five kind of edges primarily. So, we have introduced a program order edge which is between instructions of the same thread. So, that is important same thread. Does not matter it can the same addresses same set of addresses different addresses does not matter, but same thread.

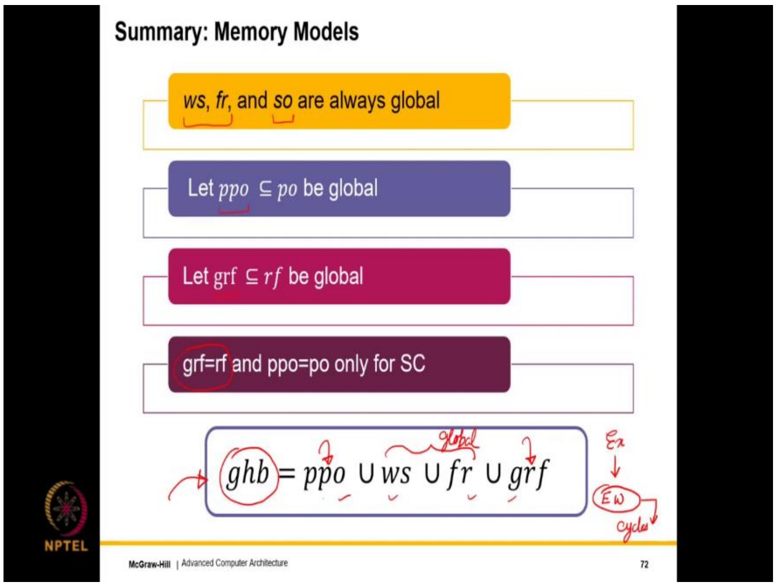
Write to read, well, it is a write to read dependence it can be all of these can be across threads these four across threads. So, the write to read operation there is a dependent same address across threads write to write read to write. So, across threads we do not have a read to read order because it does not matter in which order you actually read. So, that is not important.

So, what we do over here is that these are to the same address and also synchronization edge between synch operations. It should be the same the synchronization variable should have the same address. So, that would make it a synch operation. So, to summarize we will create an execution witness which is a graph and each operation memory operation is a node.

We will have these five kinds of edges and then we will analyze the entire graph most importantly we will search for cycle and we will use the results from topological sorting which says that if a graph does not have cycles it is possible to arrange all the operators in a linear order such that if there is a path from operation A to operation B in the graph then operation A appears first and operation B appears later.

So, we will keep on discussing this several times, but the idea of topological sorting should be clear.

(Refer Slide Time: 30:46)



Summary: Memory Models

ws, *fr*, and *so* are always global

Let *ppo* ⊆ *po* be global

Let grf ⊆ *rf* be global

grf=rf and ppo=po only for SC

$$ghb = ppo \cup ws \cup fr \cup grf$$

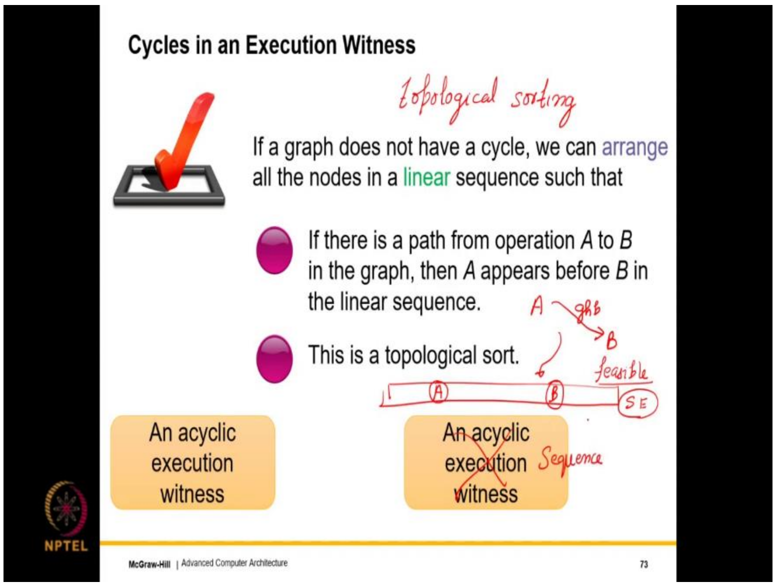
global
Ex
EW
cycle
McGraw-Hill | Advanced Computer Architecture
72

Even bigger granded summary. So, the ws, fr and so edges are always global. Ws, fr follow from PLSC and so follows from the way that our hardware implements synchronization operations. So, they are always global some subset of program order edges are global. So, let us call that subset ppo and some subset of the rf edges are also global let us call them grf. For SC grf = rf in the sense that both rf external and rf internal are global.

And ppo = po in a sense all program order edges are global. So, finally, our global happens before relationship which is exactly what the execution witness will capture will comprise all the program order edges which are global. All the rf edges also call the data flow edges that are global. Get a flow between the write and a read and the ws and fr edges which have to be global because of coherence and PLSC.

So, this is a global happens before order. So, what we do is we take an execution, a parallel execution of course. We create an execution witness. We add all the edges as per these relationships ppo and grf are prescribed by the memory model. So, we add those edges and then we search for cycles. So, this is where we have we are right now.

(Refer Slide Time: 32:28)



Cycles in an Execution Witness

Topological sorting

If a graph does not have a cycle, we can arrange all the nodes in a linear sequence such that

- If there is a path from operation *A* to *B* in the graph, then *A* appears before *B* in the linear sequence.
- This is a topological sort.

A
ghb
B
feasible
A
B
SE
An acyclic execution witness
~~An acyclic execution witness~~
Sequence
NPTEL
McGraw-Hill | Advanced Computer Architecture
73

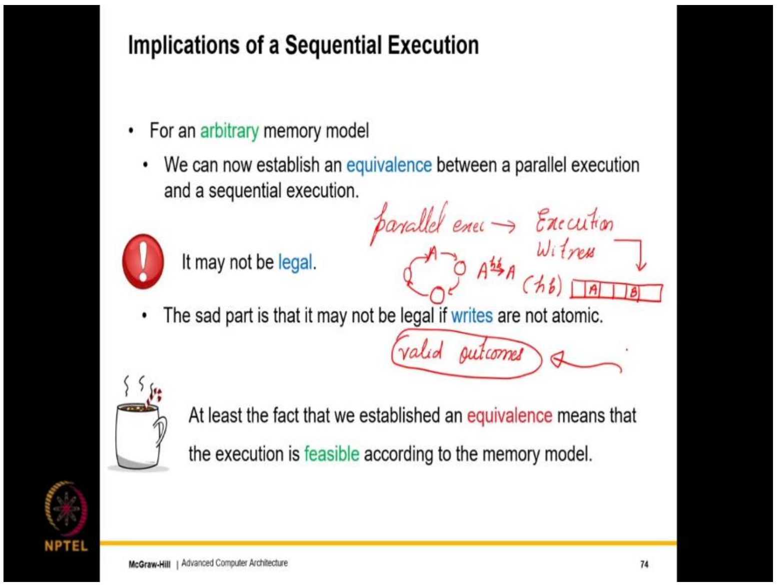
So, let us further elaborate on the fact that let us say after creating the execution witness why are cycles all that important. So, here we will use that classic computer science result of topological sorting which says the following. So, note that all of these graphs are directed graphs they are not undirected graphs.

So, if a graph does not have a cycle we can arrange all the nodes have a linear sequence such that such that such that if there is a path form operation A to B in the graph then A appears before B in the linear sequence. So, what does it mean that there is a path from A to B in the graph? It means that there is a sequence of happens before edges, global happens before edges between A and B and it means A happen first and then B happen later.

See if I can arrange them in a linear sequence of operations, so, that A appears first and B appears later and the same holds for all pairs. Then we can say that this is a possible order of completion times of these operations. In a certain sense what we can see is that this is most likely as it is a feasible execution, in the sense if let us say these operations where to complete like this then no happens before relationship would be violated. So, this is a sequential execution no doubt.

And the way that we convert a graph into a sequence is using the topological sort algorithm which maintains the same happens before orders. So, what we have an acyclic execution witness and that essentially leads to a sequence of operations.

(Refer Slide Time: 34:32)



Implications of a Sequential Execution

- For an arbitrary memory model
  - We can now establish an equivalence between a parallel execution and a sequential execution.

It may not be legal.

- The sad part is that it may not be legal if writes are not atomic.

At least the fact that we established an equivalence means that the execution is feasible according to the memory model.

McGraw-Hill | Advanced Computer Architecture 74

So, let us now look at the implication of mapping a parallel execution to a sequential execution. So, what we did is that we extended the discussion of the previous section and we took a parallel execution, created an execution witness out of it. So, in the execution witness the edges were added as per the constraints imposed by the memory model and then from this we first check for cycles.

So, we said that look if there are cycles, we can never have a cycle of happens before relationship, it is not possible. Otherwise, it will be a given operation happened before the same operation itself. So, a cycles cycle happens before relationships would look something like this. So, regardless of the other operations finally, it will come back to it. So, it will basically mean that a given event A happened before the same event which of course, is not possible.

So, we cannot have a cycle of happens before relationships. Because of that if the execution witness does not have a cycle then we are saying that the execution is feasible. Well, why? Because we can do the topological sort on the graph and arrange the operations in such a way that if there is a path from operation A to B in the execution witness A appears before B and this holds for all pairs.

So, this is a possible order in which the instructions complete, but this is not unfortunately a legal sequence all the time. It will be legal of course, if the writes are atomic, but if the writes are not atomic which means that if a write appears to complete a different time to different threads this will not be a legal sequence. So, legality will be violated. So, this is the problem.

But the key important theoretical point over here is that we establish an equivalence between a parallel execution and a sequential execution not necessarily legal. And the execution is feasible according to the memory model. So, this still our original aim of finding the set of valid outcomes is still satisfied.

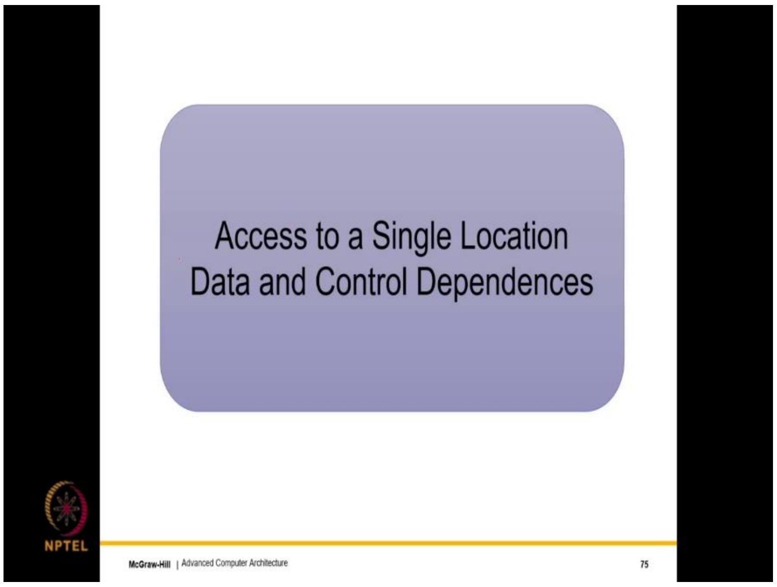
The reason is that look what we do is that we look at all possible ways that a given piece of code can execute. We create the execution witnesses. All of this can be done automatically using a software tool and then for each one of them we can check for cycles. So, there is no reason to sequentially enumerate those operations. We can if you like and further more if the writes are atomic and their execution is legal even though the program order does not hold that is still fine.

If that gives us more tools and more theoretical insights we can do that, but even at the level of the execution witness itself. If it does not have cycles it means at the execution in terms of the outcomes is valid. So, this does give us a very strong and powerful tool to find out when a given execution is valid for a particular memory model.

So, this is where we would like to end this part with regards to the validity of executions with respect to certain memory model. And then there are also other aspects of an execution that we shall look at next, but at least from the point of view of memory models we established an equivalence and then as long as the execution witness does not have cycles.

Which means that there is no cycle of global happens before edges we are fine. This essentially indicates to us that a given execution is per se valid right the outcomes are valid they are allowed which is the key thing that we wanted to theoretically investigate in this part of the lecture. So, let us keep going and look at a few more criteria for validity this was one, but there are a few more as well.

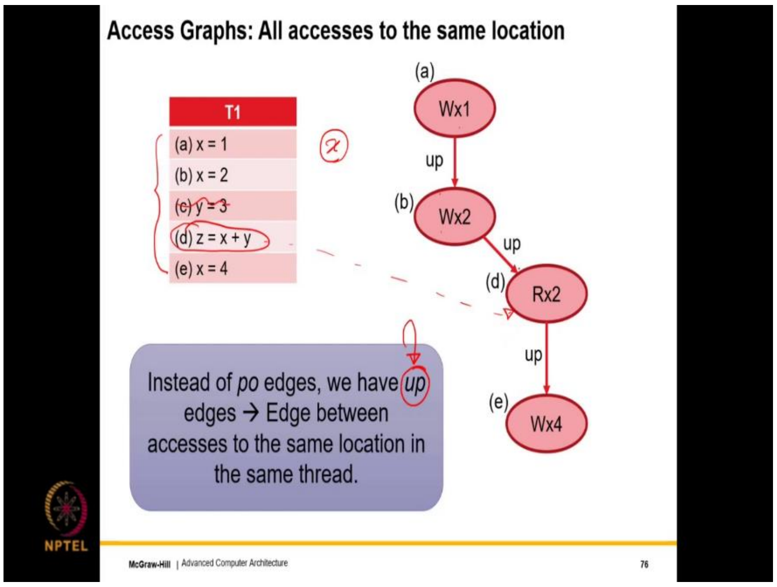
(Refer Slide Time: 38:55)


Access to a Single Location
Data and Control Dependences

McGraw-Hill | Advanced Computer Architecture 75

So, next we will look at accesses to a single location and we will also look at data and control dependencies.

(Refer Slide Time: 39:05)


Access Graphs: All accesses to the same location

T1
(a) x = 1
(b) x = 2
~~(c) y = 3~~
(d) z = x + y
(e) x = 4

Instead of *po* edges, we have *up* edges → Edge between accesses to the same location in the same thread.

McGraw-Hill | Advanced Computer Architecture 76

So, let us create a new tool called an access graph which in principle is similar to an execution witness, but it is different. So, it contains all accesses to the same location all accesses are to the same location. We can have multiple threads. I have shown a diagram for a single thread, but we can have multiple threads as well no issues. Say consider all the operations issued by a single thread.

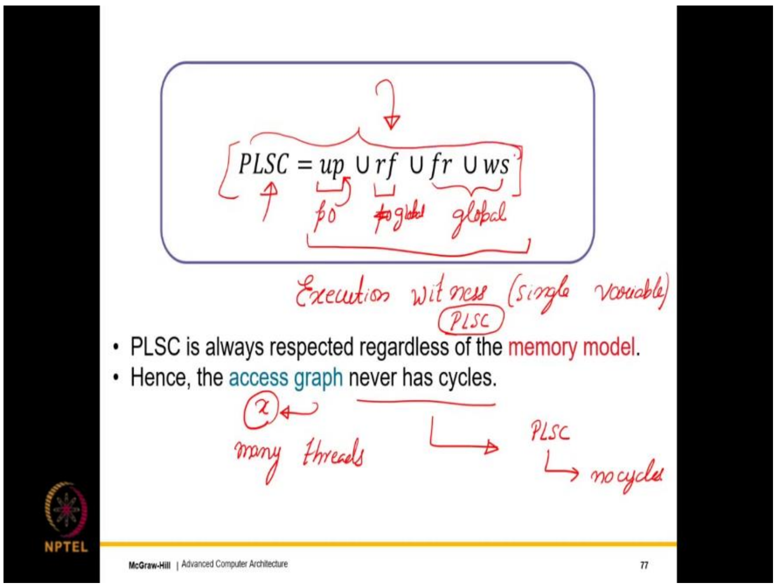
So, we will consider all the accesses initially for address x. So, clearly this will not be there. So, I will have Wx1 and Wx2. So, we will introduce a new kind of edge. So, instead of program order edges we will have up edges called uni processor edges I will discuss in a second while.

So, it is edge between accesses to the same location in the same thread. So, mind you, this is accesses. So, this can be read write write, write, write, read, read, read anything, its two accesses to the same location within the same thread. So, we will create a new kind of edge and we will call it the up edge which of course, does capture program order, but it is again accesses to the same location within the same thread.

So, what we see over here is that we have we write 1 first then 2 we add a up edge and then so, I am treating this simplistically as you know several instructions, but simplistically as one instruction in the access graph where we just read the latest value of x = 2. So, we add one more up edge and finally, we write 4.

So, as you can see we have extracted four instructions, I have added uni processor edges and let us take it slightly further.

(Refer Slide Time: 41:01)



$$PLSC = \underbrace{up}_{po} \cup \underbrace{rf}_{\neq global} \cup \underbrace{fr \cup ws}_{global}$$

Execution witness (single variable) (PLSC)

- PLSC is always respected regardless of the memory model.
- Hence, the access graph never has cycles.

x ← many threads

PLSC → no cycles

McGraw-Hill | Advanced Computer Architecture 77

So, here is what I claim which is a simple generalization of the discussion that we had in the previous section on SC. So, pretty much in SC we had program order. So, we are now looking at PLSC which is all the accesses for a single location. See here also ws and fr are global and program order in this case is up. And in SC since all of rf is global, so, we have program order here as well.

So, what is it that we are trying to do? What we are trying to do is that we are trying to look at all the accesses to the same location, does not matter it can be one thread or it can be several threads that does not matter. So, similar to SC we can define PLSC over here.

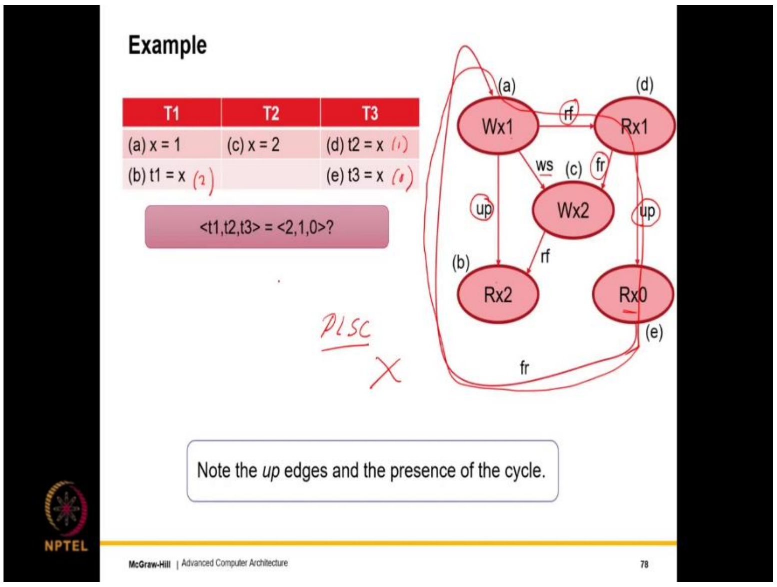
Of course, PLSC will be confined to everything for the same location. Say instead of po we will use up and since all rf relations are global in SC they will be global in PLSC also at least in the definition. So, this is global and finally, we will add make fr and ws global. So, this for us is PLSC.

So, what we have seen before this discussion is that PLSC is always respected regardless of the memory model. So, regardless of the memory model we always respect PLSC that is point 1. And because of that when we create an access graph which will only have accesses to let us say a single location, does not matter many threads we will have a very similar graph as the execution witness.

So, like the access graph is pretty much in execution witness confined to a single location and so, in that case we will add our fr ws edges of course, rf edges of course, they will be global and our po edges and we expect. So, there is no memory model with respect to accesses to a single location because PLSC needs to hold.

So, we expect that the access graph will not have any cycles. Why? Because if it has cycles PLSC does not hold, recall our discussion in slide number 74. So, here the access graph or PLSC to hold it implies that there are no cycles.

(Refer Slide Time: 43:44)



Example

T1	T2	T3
(a) x = 1	(c) x = 2	(d) t2 = x (1)
(b) t1 = x (2)		(e) t3 = x (0)

<t1,t2,t3> = <2,1,0>?

Note the *up* edges and the presence of the cycle.

McGraw-Hill | Advanced Computer Architecture 78

So, let us now look at data and control dependencies as well and then try to create a different kind of graph. So, we have looked at the access graph in the previous slide and access graph was same as an execution witness. So, an same execution witness for a single variable.

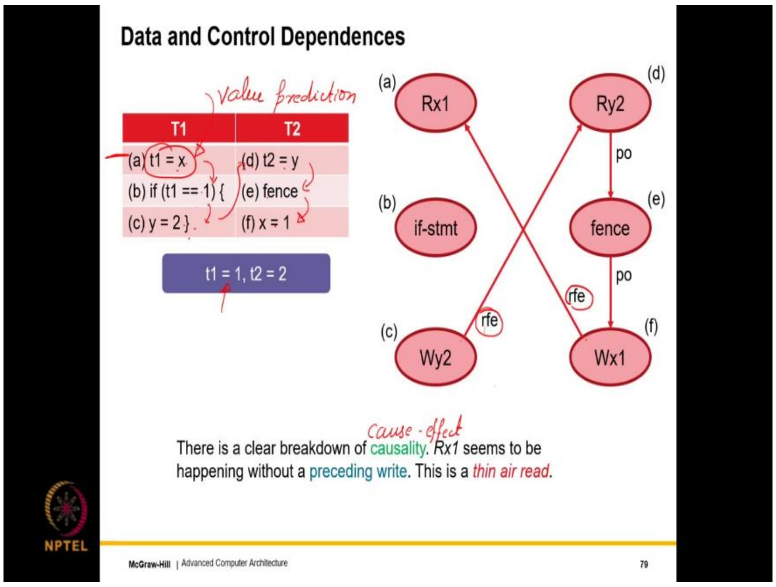
So, in this case of course, PLSC has to hold which is essentially sequential consistency for a single location and this part should be self explanatory by now. So, let us now consider one more example. So, in this example we will have up edges uni processor edges and we will have multiple threads; the threads t1, t2 and t3. So, in thread t1 we write to x and then read, t2 we just write and t3 we read twice. So, is this outcome allowed, let us see.

So, t1 t2 t3. So, t1 = 2. Here we read one first and then we read 0, is it allowed? Well, so, note the up edges. So, note the up edge from Wx1 to Rx2. So, they are a part of the same thread. So, given the fact that they are a part of the same thread. We see this uni processor edge to enforce the dependency and then we have Wx2 write serialization edge over here and then write to read edge because instruction B reads the value produced by instruction C.

So, there is no cycle over here, but we will see a cycle when we consider thread t3. So, thread t3 reads 1 first. So, we have seen rf edge and then fr edge. So, why an rf edge? Because there is a write to read dependence over here and an fr edge because there is a read to write. So, we read an older value and then Wx2 with later value and then the up edge to the subsequent read where we read a 0.

So, this is where the fun is. So, because we read a 0 there will be an fr edge from this access to Wx1 because we read 0 first and then 1 is returned to the location. So, we do we do see a cycle and a cycle goes all the way up like this, this way, this way and this way. So, given the fact that we have a cycle in the access graph, PLSC is violated and consequently this execution is not allowed. So, now we have seen PLSC which is very important if you consider accesses to a single location.

(Refer Slide Time: 47:08)



Data and Control Dependences

value prediction

T1	T2
(a) t1 = x	(d) t2 = y
(b) if (t1 == 1) {	(e) fence
(c) y = 2 }	(f) x = 1

t1 = 1, t2 = 2

cause - effect

There is a clear breakdown of causality. *Rx1* seems to be happening without a preceding write. This is a *thin air read*.

McGraw-Hill | Advanced Computer Architecture 79

So, we are not done yet. There is something called a thin air read which we shall see next. So, let us now also consider data and control dependencies. So, while considering data and control dependencies what we see is that so, we are looking at a program with two threads. So, here we read x first. So, let us assume that this read.

So, this read will be affected by a load operation. So, let us assume that we also have value prediction over here. See if you have value prediction so, what happens in value prediction? While in value prediction we predict the value early. And later we read the value from the memory system and we verify.

So, in this case, we read the value, but that might take time. So, in the time being we predict and we verify later. So, for uniprocessor executions we have been clamming that value prediction is absolutely correct, just the paradigm itself. The reason is that we will never read a wrong value because we are always verifying a value before committing. Is that the case of multiprocessor executions, let us see.

So, let us assume that the value is stored in the register t1 and if t1 = 1 which is the if statement. So, in this case, what we do is we set y = 2. So, let us assume we find it to be 1 in this execution and then once we set y = 2, let us assume we have atomic writes. So, in this case, we will have an rfe edge from Wy2 to Ry2.

So, let us further assume that our memory model is such that we do not have a read to write ordering between accesses to different addresses in the same thread and a lot of memory models which are called weak memory models as we shall see they will not have this ordering.

So, we have not added any edges in t1. So, in t1 no edge per say has been added. Now, let us look at thread t2. So, in this case, we have a fence. So, program order edge two of them are required. So, in this case, what we do is we read the value to be 2. So, that justifies the rfe edge and then we have two program order edges and then we write 1, Wx1. And then what we actually get to see over here is that Wx1 supplies the value to Rx1 via an rfe edge.

So, first let us look at the outcome. So, the outcome of course, since the t1 = 1 as you can see and the outcome is t2 = 2 because of the rfe edge. So, this execution witness over here does not have a cycle. So, we might be tempted to say that this is a legal and feasible execution, but the answer is that it is not. Why is that the case?

The reason that that is the case is that there is a clear break in what is called causality. A causality is essentially a cause effect relationship. So, any time there is a cause and effect relationship we call it causality and there is a clear break, let me explain why?

So, the reason is that we read the value of x * t1. So, we assume that at this point x = 1. After that since x = 1, we set y = 2 and then we read y = 2 then of course, we have a fence. So, we have a strict program ordering and then we set x = 1. So, there is a as you can see there is a causal link which goes like this. So, one event after the.

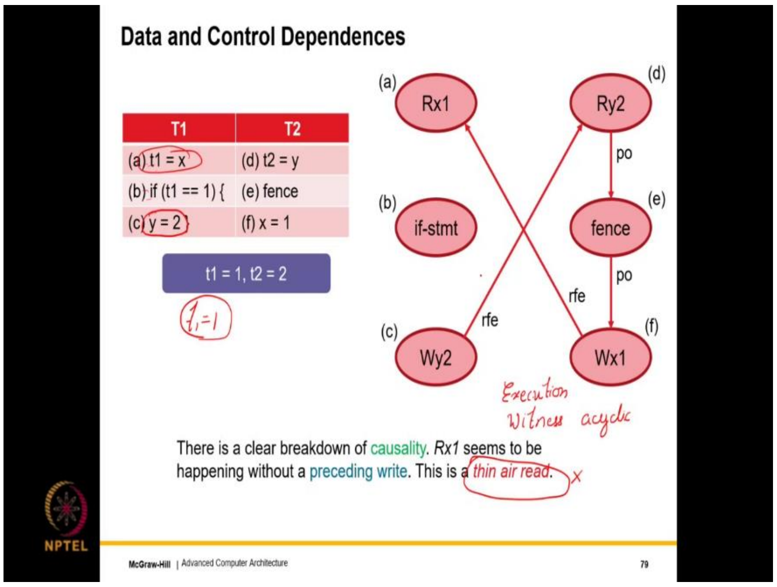
So, this first this event then this then this then this then this then this. Now, the point is at this point x ≠ 1 if you think about it. The reason is that we set x = 1 at the end of the chain of this causality at the end of the causality chain. In the sense, first something happens then something then something and then at the end we said x = 1, but x was not 1 at the beginning. But how can this execution happen?

Well, the way that this execution can be can happen is as follows. We predict the value of x. So, we predicted to be 1 then we keep executing. After predicting it to be 1, what we do? So, what we do is that we go in the speculatively perform the write. So, in this case, you form we perform the write early and we do not wait for the instructions prior to the write get committed.

So, this does happen in extremely high performance processors where they this read to write order is violated and they do perform early writes. And after that what we see on the side of t2 is there in t2 we perform the load and then the fence and then we write 1 = x. And so, at the time of verification of this instruction we will always read x to B 1 and this instruction would deem to have been verified, but that will actually not be the case.

The reason that will not be the case is basically because when we originally read x, it was actually 0 and then this is kind of a spurious value.

(Refer Slide Time: 53:24)



Data and Control Dependences

T1	T2
(a) t1 = x	(d) t2 = y
(b) if (t1 == 1) {	(e) fence
(c) y = 2 }	(f) x = 1

t1 = 1, t2 = 2

$t_1 = 1$

There is a clear breakdown of causality. *Rx1* seems to be happening without a preceding write. This is a *thin air read.* x

(a) Rx1
(b) if-stmt
(c) Wy2
(d) Ry2
(e) fence
(f) Wx1
po
po
rfe
rfe
Execution witness acyclic
NPTEL
McGraw-Hill | Advanced Computer Architecture
79

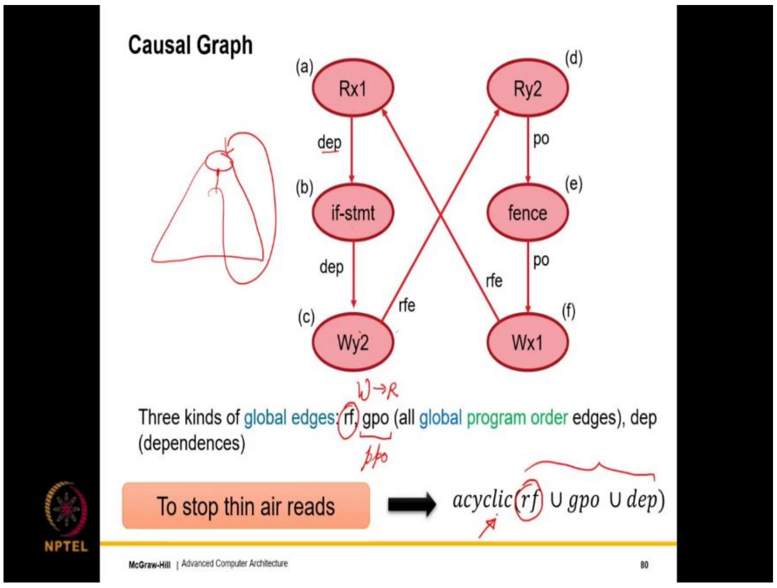
So, x = 1 or rather t1 = 1 appears to be appears to have been read from nowhere and so, that is the reason it is called a thin air read. Because if you think about it t1 = 0 because x became 1 just because we assume that x = 1 over here, but that was the wrong assumption because this point x = 0. So, this value kind of has been picked out of thin air. So, that is the reason it is called a thin air read.

And such kind of an execution would be valid as per our execution witness logic that this execution witness is acyclic, but we would still like to prevent such kind of thin air reads. So, I would further like to say that this the problem of having thin air reads also does arise when for example, we are implementing a virtual machine by the Java virtual machine and so on.

So, that is why this is a general theoretical concept where essentially there is a break in causality and the reason being that we assume a variable to have a certain value and later on the assumption is proved to be correct just because we assumed. So, that this is of course, not the right way. It is kind of a circular logic which is incorrect and that is why such thin air reads have to be prohibited.

So, what I would request all the listeners of this video to do is to take a look at this example, convince themselves that this example is actually not correct, but our execution witness method says that it is correct. So, we have to do something in addition to this to stop this from happening.

(Refer Slide Time: 55:08)



Causal Graph

$$\begin{tikzcd}
\text{(a) Rx1} \arrow[d, "dep"] & \text{(d) Ry2} \arrow[d, "po"] \\
\text{(b) if-stmt} \arrow[d, "dep"] & \text{(e) fence} \arrow[d, "po"] \\
\text{(c) Wy2} \arrow[ur, "rfe"] & \text{(f) Wx1} \arrow[uul, "rfe"]
\end{tikzcd}$$

Three kinds of global edges: rf, gpo (all global program order edges), dep (dependences)

To stop thin air reads ➡ $acyclic(rf \cup gpo \cup dep)$

Let us now create a causal graph. Creating the casual graph, we will have three kinds of edges. So, we will have creating graphs. So, we keep creating graphs of different kinds.

So, let us compare a casual graph with the previous figure. This is the execution witness this is fine, but we take the execution witness we add three kinds of edges. So, we keep the rf edges because they determine data communication.

So, there is they determine causality in the sense I write and then I read. So, this causality is maintained then we add all the global program order edges. So, called gpo. So, I have also referred to this as ppo previously in the slides. So, ppo is called preserved program order which for that particular setting was correct, but what should have probably been used and which was more relevant was gpo all global program order edges even the ppo was not wrong.

But we will use we will prefer the term g po and in the book also I have use gpo. So, what are the edges we add again? We add rf because so, what is the point of view. It is important to define the observer here and the point of view. So, the observer is basically an observer all on all the threads is only looking at the communication of data in the sense that I read something and then what are all the operations that are dependent on the read. So, one is that I might be communicating some data the way that I am doing from Wy2 to Ry2.

So, then after that so, the observer will basically change the point of observation Wy2 to Ry2. All the global program order edges in the sense that Ry2 needs to complete then the fence then Wx1 and also a set of dependencies which means one instruction needs to complete then the dependent instruction in this case the if statement and then the instruction in the body of the if statement.

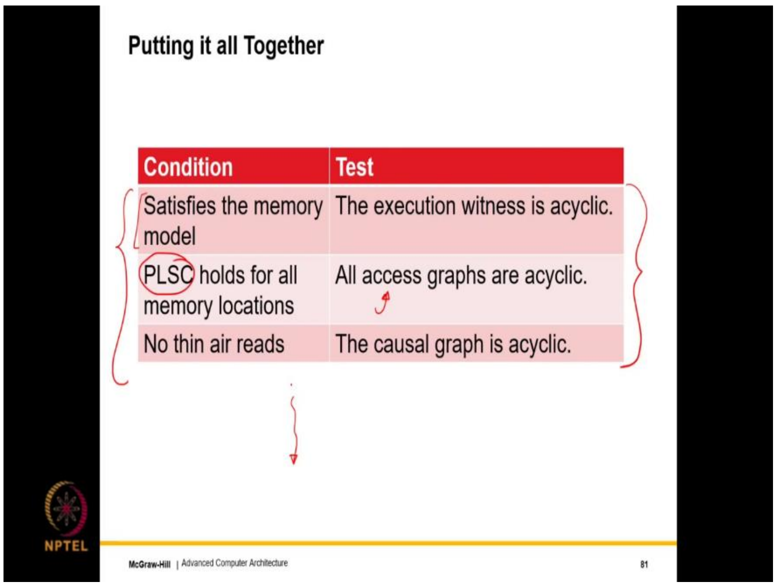
So, this is a different kind of graph. We have not really seen this kind of a graph before. So, this is called a casual graph. So, here we have rf edges which in this case are global. The reason is that the observer does is not really looking out looking at other threads about what they think about a write.

It is only looking at the flow of data. So, think of the observer pretty much as sitting on the forward slice of the read. So, basically wherever whoever is affected by the read data or control the observer is sitting on the forward slice and tracking it.

So, pretty much what a thin air read basically means is that a read is a part of its own forward slice which is never the case which will never happen that no instruction will kind of be kind of be a part of its own forward slice via a path like this. So, that is exactly what we are seeing that from the read there is a path from the read to the read or so, read is a part of its own forward slice and there is a path from the read to a read that passes through other nodes.

So, what the observer would do in this case is sit on the read and simply start observing all the instructions that are dependent on the outcome of the read that would include data and control dependencies, communication behavior write to read as well as global program order edges. So, we will create a causal graph out of these three kinds of edges and this needless to say needs to be acyclic because if it is cyclic it means that there is a break in causality. So, there is the thin air read basically.

(Refer Slide Time: 59:15)



Putting it all Together

Condition	Test
Satisfies the memory model	The execution witness is acyclic.
PLSC holds for all memory locations	All access graphs are acyclic.
No thin air reads	The causal graph is acyclic.

McGraw-Hill | Advanced Computer Architecture 81

So, now the time is come to put all of it together. So, to put all of it together, so, we are looking at the validity of an execution. So, we look at the validity of an execution from several points of view in the sense we define different kinds of observers with different points of view and different capabilities.

So, first we check if it satisfies the memory model. So, different architectures have different memory models. In this case, the execution witness needs to be acyclic you have seen that. Then does PLSC hold for all memory locations? Yes, it has to. Otherwise, the notion of a single shared memory will stop to hold.

So, in this case, all access graphs are acyclic. Then again, we do not want thin air reads. So, in this case, the causal graph will be acyclic. So, we have to ensure that for every execution these three hold. So, in a certain sense these three conditions hold that we are satisfying the memory model we are satisfying PLSC and further more we do not have any thin air read. So, there is no break in causality.

So, these are pretty much the three most important conditions of any execution being correct on a multiprocessor system. And we also think about it for a uni processor system all that we need is PLSC because for every read we just need the latest write or if there is a single thread running on a multiprocessor system we also need PLSC not the others. So, they are not really relevant in this case.

So, in this case, all single threaded programs will work correctly on a multiprocessor system. That is something that is given if PLSC holds. But in addition if you have multiple threads and multiple cores you have to satisfy the memory model and avoid thin air reads.

(Refer Slide Time: 61:25)

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
82

So, the next in the next lecture we will move on to cache coherence. So, we have finished the first two parts of this chapter.