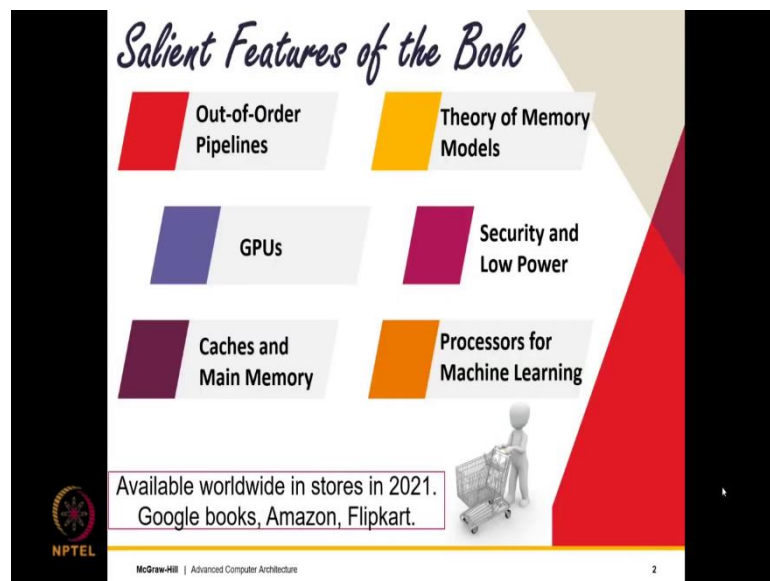


**Advanced Computer Architecture**  
**Prof. Smruti R. Sarangi**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**

**Chapter - 09**  
**Lecture - 26**  
**Multicore Systems Part - II**

(Refer Slide Time: 00:24)



*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

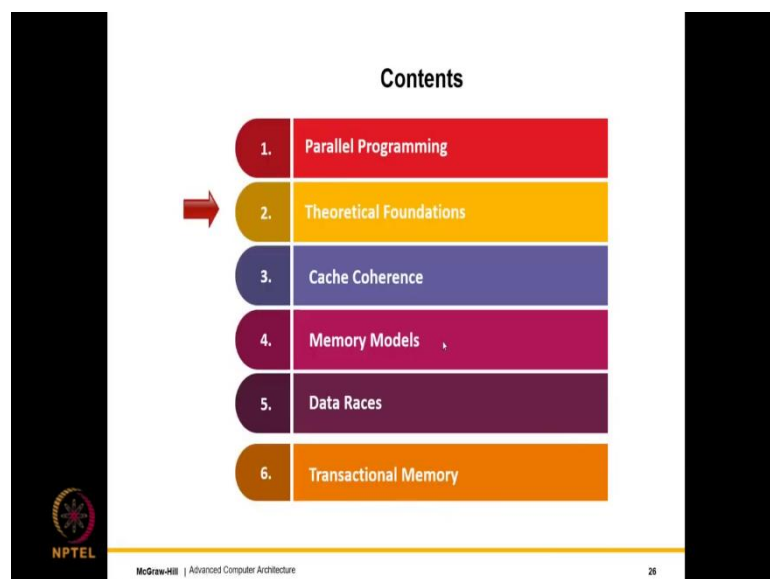
Available worldwide in stores in 2021.  
Google books, Amazon, Flipkart.

NPTEL

McGraw-Hill | Advanced Computer Architecture

2

(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

26

(Refer Slide Time: 01:13)

So, let us look at the main problems. The main problems are like this that we are dealing with large caches. So, because we are dealing with large caches what we discussed in the previous chapter is that, we create large multi bank caches where the banks are distributed all over the chip and to connect all of those cache banks there is a need to create an on chip network.

Furthermore, if we have a lot of cores that implies a lot of parallel accesses. So, what we need what we are dealing with is a large slow and multi ported cache. Of course, its internal organization can be into multiple banks, but if I see the entire L 2 or L 3 cache as one unified hole, then its going to be large its going to be slow and multi ported.

This well this is not possible to build let me add a further rider to it that if let us say at the L 3 level yes, it is possible to build at the L 3 level may be at the L 2 level also it is possible to build, the primary reason being that the number of requests that actually come to L 2 or L 3 is significantly lower because it gets filtered by the L 1 level, but consider the L 1 level that is the most the that is the trickiest.

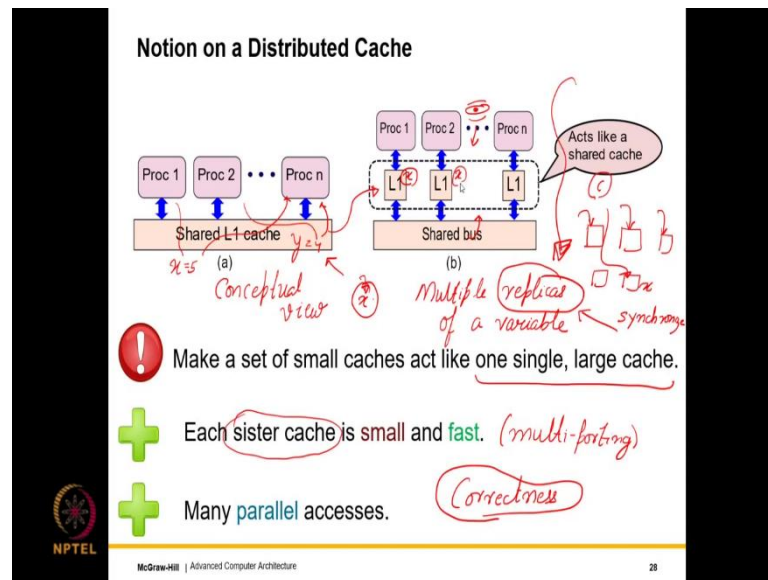
So, we have a core then we have an L 1 cache connected to it, we have one more core we have an L 1 cache connected to it. So, these are essentially they have to be the same L 1 cache otherwise what will happen is, I will write to a variable over here and the other core will not be able to see the write, but that is against our shared memory model of programming. So, our current understanding is that we will have one large L 1 and the course will be connected to it.

This is at least the conceptual model that we have been working with and definitely we will have one large L 2 and all the cores will also access that and as I said both at the L 1 and L 2 levels, it will end up being large slow and multi ported. This multi porting will not be that important for L 3 because the number of L 3 requests will be low that is because of the filtering that will happen.

So, in L 3 what matters more is the size, but clearly at the L 1 levels where definitely there are one or two accesses every cycle and let us say we have 16 cores. So and let us say two accesses per cycle. So, we will have a maximum of 32 accesses which is a huge amount of multi porting and also the addresses will be shared.

So, one core will write to one address one core will read from one address that is an issue. So, that is why we need to somehow create a distributed cache where we have a set of small caches such that they conceptually act like a large cache. So, this is the broad concept, but do not worry I will be coming back to this.

(Refer Slide Time: 05:10)



So, what I am trying to say is that, this is the conceptual view. This is the conceptual view where all the processors. So, it processors course I am using them interchangeably over here they are sharing an L 1 cache which is also what we want conceptually such that I set x to 5 over here this core sets it and this core reads it I set y to 4 this core set sets it this core reads it.

But the again, if I have such a large cache it will require a large number of ports even if I do have a banks there will be lot of bank conflicts. So, keeping all of these things in mind what I should actually do is that I should opt for this design where all the L 1 caches are private to each core and the set of L 1 caches they act as if. So, act as if means the software perceive or even the core perceives or anybody let us say who is outside this shaded rectangle perceives that all these L 1 caches are actually a single cache.

So, for an outsider whether it is a set of distributed L 1 for a single L 1 will not matter. So, this is different from banking I will tell you how there is a crucial difference. The crucial difference is that you might now argue that what we were doing for a large caches that we were splitting the address space and we were assigning the addresses to different banks.

So, we were saying that look if the upper KMSP bits are certain value you go to this bank otherwise you go here otherwise you go here. Why cannot we do this with a shared L 1 cache? Well, the reason we cannot actually do it is like this that. So, let us assume that we

do it. So, let us assume that we take a large L1 cache and we multi bank it and let us say that I write to a certain variable.

See if let us say write to x if I write to x, x might physically be in a place which is very far away from me and the write will take a lot of time and even worse the read will also take a lot of time. If x is located far away from the core and that will slow down the core to a large extent.

So, the L1 cache by definition should be close to the core and should be accessible and should be accessible within 1 to 2 cycles which means that if let us say multiple cores are accessing the same variable x, they will have to maintain multiple copies of it multiple replicas of it.

So, this was not there in a multi bank cache, but this is the key idea of a distributed cache that the reason we have a fast access is that we maintain multiple replicas of a variable that is a key idea. So, this is how a distributed L1 cache differs from actually a large shared cache with multiple banks.

Because if multiple banks you maintain a single replica, but the point is that replica might be located far away from you and it will take a lot of time. So, the core is over here the replica is over here, it will take a lot of time to come to this point and read the value. But let us say that if we have a multiple replicas they can quickly access the replica that is there in my private L1 cache and the accesses will be really fast.

But then if I have multiple replicas from the same variable I have to synchronize that otherwise my notion of shared memory will break. So, I need to make a set of small caches designed in this manner which of course, is very different from a multi bank design act and look like one single large cache to an outsider. The outsider would be somebody who is let us say let me draw an I looking at it from here or looking at it from here.

And of course, the L1 caches might be connected with a shared bus which is just a set of copper wire or they can be connected via an NoC as well that is not important. Say each of these L1 caches is called a sister cache it is small and fast say it is giving you the advantage of high speed further more since they all can be accessed independently, it is giving you the advantage of multi porting that advantage is that advantage is being derived

and further more we do not have to access a remote L 1 cache at least most of the time to access a variable that will always be there close to the core. So, it is fast.

So, the main issue here is actually with correctness that if we were to design such a distributed cache which will maintain multiple replicas and this replicas is a big issue this is the elephant in the room. If I were to maintain multiple replicas then how will they synchronize with each other that becomes a very important issue otherwise what will happen is at I will write to x another core will write to x then nobody else will see the rights or they might see it in one order somebody else will see it in another order it will all become a mess.

So, if I want external word to think that these are its actually a single cache where there is only a single location for the variable x, I need to do something you will see what I need to do.

(Refer Slide Time: 11:14)

**Shared vs Distributed Caches**

❓ We want a set of small L1 caches (sister caches) act like one, single, large L1 cache. How is this possible?

Consider accesses to a single memory address/variable,  $x$

- If we have a single physical location for  $x$ , then there is no advantage of a distributed design.
- If we have multiple locations for storing the variable  $x$ , then the replicas have to contain the same value.
- This is the problem of coherence.

Coherence → Make a set of locations behave like a single location.

NPTEL

McGraw-Hill | Advanced Computer Architecture

29

So, let us revisit this issue once again in some detail. So, what is the question? The main question is that we want a set of small L 1 caches known as sister caches to act like one single large L 1 cache. So, we are saying that this set of small L 1 caches is will give us everything.

So, it will be high speed, it will give us the ability to make many many parallel accesses by a large number of cores, the issue was at correctness and the issue with correctness was

that to give us all of the aforementioned properties we will have to maintain multiple replicas of a variable a variable and memory address are the same thing in this scale. So, let us consider accesses to a single memory address slash variable because they mean the same thing let us call it x.

See if you have a single physical location for x which is always the case in a shared cache, then there is no advantage of a distributed design. So, we need to have multiple locations to store the variable x and, but then the issue at that point is the replicas have to contain the same value otherwise our program will show non intuitive behavior and the distributed cache will not appear to be a single cache.

So, making a distributed cache appear as a single cache is the problem of coherence. So, coherence basically means make a set of locations. So, these are physical locations these are these are physical locations in the sister caches behave like a single location. So, this will ensure that regardless of however, howsoever we design our ensemble of L 1 caches, they will appear to any external observer as a single cache or in other words if there are multiple locations within this L 1 cache that are storing the same variable x, it should appear to an external that there is only a single location for variable x.

If somehow we are able to ensure this or somehow if we are able to maintain this abstraction, we are deem to have solved the problem of coherence and if we solve the problem of coherence, then we can realize a fast distributed cache and that will give us high performance for multi threaded core.

(Refer Slide Time: 13:56)

**Consider Multiple Locations**

**Assumptions**

- All **global variables** are initialized to 0. They start with u, v, x, y, or z.
- All **local variables** are mapped to registers. They start with a 't'.
- The delay between the execution of **consecutive instructions** can be indefinitely **large**. Instructions across threads can execute in **any order**.

Thread 1	Thread 2
x = 1	t1 = y
y = 1	t2 = x

Is the outcome  $\langle t1, t2 \rangle = \langle 1, 0 \rangle$  possible?

It for some reason bothers us because no **sequential ordering** of instruction executions can produce this **outcome**.

*Handwritten notes:*  $t_1 = 1, t_2 = 0$  and "Many outcomes" with an arrow pointing to the table.

NPTEL | McGraw-Hill | Advanced Computer Architecture 30

So, let us now discuss a couple of assumptions that we will make while presenting our examples. So, these assumptions will hold throughout the presentation. So, the assumptions are like this that all global variables are initialized to 0 and they start with the letters u, v, x, y or z. So, they will be a global variable which means that these variables are shared across the threads between the threads across the cores and it is possible for a core to write to a global variable and another core to read it. In comparison all the local variables are mapped to registers they start with a t.

So, t for you can say temporary. So, they would be of the form t1, t2, t3 and so on. So, all the local variables any right to them these writes are not visible across the cores. So, these writes are not visible. Furthermore, what we will assume is that the delay between the execution of consecutive instructions can be indefinitely large.

So, we cannot for example, say that look we will execute this instruction then maybe we are guaranteed that within a window of a 100 or 200 or 300 cycles y equal to 1 will execute. The reason that we say that the difference can be indefinitely large is basically because it is possible that the process might context switch. So, the process after executing (x = 1) might context switch and then some other process may execute and for a long time and this might come back or the core might get stuck a lot of events can happen.

So, we can say that event a happened before event b or an event a happened after event b or we cannot say anything, but we can definitely we will not use any cycles for the timing.



So, this is important. So, most students make a mistake in this part where they assume that two events will happen either at the same time or within the window of a certain number of cycles, but that is not the case.

So, in a concurrent system particularly when it comes to a theoretical explanation, we just say we just use the words before and after before means before after means after we do not say one cycle before or 10,000 cycles before. So, let us now consider this piece of core.

So, in this piece of core thread 1 is writing to variable x and then to variable y and thread 2 we are reading them in the reverse order. So, we are reading y first and x later. So, one thing that should be immediately obvious is that given the fact that there is no synchronization between them in the sense that thread 2 is not waiting for thread 1 to finish anything.

So, we can have many possible outcomes for this multithreaded core, in the sense we can have many interleavings of the instructions say its possible this instruction executes first and then this and then this and then this or its possible this instruction executes first then this, then this and then this they have many interleaving and consequently many outcomes.

So, this is true for most multithreaded core that unless we do additional things we will see what we can have many possible outcomes, but out of all possible outcomes we are interested in this outcome which is basically  $(t_1 = 1)$  and  $(t_2 = 0)$  what we are interested to know is this outcome possible and that is the question that concerns us and the reason it concerns us is basically because we find something non intuitive with this outcome, what is it? We find that no sequential ordering of these instructions can actually produce this let us see why?

(Refer Slide Time: 18:32)

**Consider Multiple Locations**

**Assumptions**

- All **global variables** are initialized to 0. They start with u, v, x, y, or z.
- All **local variables** are mapped to registers. They start with a 't'.
- The delay between the execution of **consecutive instructions** can be indefinitely **large**. Instructions across threads can execute in **any order**.

Thread 1	Thread 2
x = 1	t1 = y
y = 1	t2 = x

- Is the outcome  $\langle t1, t2 \rangle = \langle 1, 0 \rangle$  **possible**?
- It for some reason **bothers** us because no **sequential ordering** of instruction executions can produce this **outcome**.

Handwritten notes on the slide:

- $x = 1$
- $y = 1$
- $t_1 = y$
- $t_2 = x$

NPTEL

McGraw-Hill | Advanced Computer Architecture

30

Say we first set  $x = 1$ , then we set  $y = 1$  and then after that it appears that in the ordering we are looking at a situation like this  $x = 1, y = 1$  then we are reading  $y$  which is we are reading 1 and then we are reading  $x$ . So, if you are reading  $x$  there is simply no way by which we can read the initialized value of  $x = 0$  because we have already read  $y = 1$  we have already read  $y = 1$ . So,  $x$  has to be read to be 1, but we are reading it to be 0.

So, we are saying that yes there might be when many outcomes, but this particular outcome in our opinion is not possible because it does not meet our standard of intuitiveness. It does not appear to be very intuitive because our logic is that look first we set  $x$ , then we set  $y$  then we read  $y = 1$ . So, we should definitely read  $x = 1$ , but we are reading it to be 0 and this is where we have a big issue.

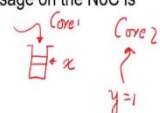
(Refer Slide Time: 19:42)

This outcome is indeed valid on many machines

Thread 1	Thread 2
x = 1	t1 = y
y = 1	t2 = x

- Thread 1 sets x to 1
- It sends an update message on the NoC. The message gets caught in congestion.
- Thread 1 sets y to 1. The corresponding message on the NoC is swiftly delivered.
- Thread 2 reads y to be 1.
- Thread 2 reads x as 0 (initialized value)

This is indeed possible!



NPTEL

McGraw-Hill | Advanced Computer Architecture

31

Let us look at this once again. So, just verify its the same core see here is what can happen? Assume that thread 1 is running on core 1 and thread 2 on core 2. So, this is a generic assumption that we will continue to make in this presentation in the slide set that thread I unless specified otherwise always runs on core i. So, what will happen is that thread 1 sets  $x = 1$  which means the memory address corresponding to x, a store instruction is set to set it to 1 and the default values the initialize values are all 0.

So, then what it does is, it sends an update message on the NoC to the location of x if its there in its private cache that is great, but then the point is that needs to be visible to the other core. So, let us not get into the details but, let us just assume that  $x = 1$  is sent via a message on the NoC is the network on chip and the message gets stuck in congestion its enters a traffic jam a message jam.

Then thread 1 sets  $y = 1$ , it again sends a message, but y is of course, stored in a different place in the shared L1 cache or can be a distributed L1 cache it does not matter in the large L1 cache. So, the message to update y actually gets delivered and that through it gets delivered very quickly.

So, then what will happen? After that when thread 2 reads y, it will actually see  $y = 1$  because the message to set  $y = 1$  would have been delivered and it will be visible to all the threads, but when it tries to read x given that the update message to x is kind of stuck in congestion, it will read  $x = 0$  which is the initialized value.

So, what you can see is that in a large system with course cache banks distributed caches and NoC's this is indeed possible. So, indeed there is a possibility that this can happen. So, I will give you an example which is closer home. So, assume that we have a write buffer. So, as we discussed in chapter 7, a write buffer is one of those simple simplistic structures we just accumulate writes to the same block.

So, assume that x is stored in the write buffer, but y is not stored. So, what will happen as at the moment the store to x comes, it will get buffered in a write buffer, but other cores will not be able to see it. So, from core one's point of view this write is done, when it sets  $y = 1$ , y might bypass the write buffer and go and update y in some other location and subsequently when we read y in core 2 core 2 will get this value, but since the write to x is still stuck in the right buffer of core 1 the write to x will not be visible.

And then what will happen is that x will be read to be 0 and as you can see we will end up with a non intuitive outcome, but on a real machine if we have to have these complex features of a non chip network and write buffer and all of these sophisticated things, then this outcome is indeed possible. We might not like it, but what we will see is that if we have the complexities that we have mentioned this outcome is indeed indeed indeed possible.

(Refer Slide Time: 23:26)

One More Example

Thread 1	Thread 2
$x = 1$	$y = 1$
$t1 = y$	$t2 = x$

Can we read  $\langle t1, t2 \rangle = \langle 0, 0 \rangle$ ?

**Yes!** The loads are issued **early**, and the writes happen at **commit time**.

Handwritten notes:

- $b_1 = 0$
- $b_2 = 1$
- $x = 1$
- $y = 1$
- $t_1 = y = 1$
- $t_2 = x = 1$
- $t_1 = 0$
- $t_2 = 0$
- $\langle 0, 0 \rangle$

NPTEL

McGraw-Hill | Advanced Computer Architecture

32

Let us look at one more example and you will again see how what we think of as common sense is actually not all that common or all that sensible when it comes to a multi core

system. So, let us assume we have again two threads thread 1 sets  $x = 1$  reads  $y$  thread 2 sets  $y = 1$  and reads  $x$ . So, can we have this outcome? What is the outcome?  $t_1 = 0$ ,  $t_2$  is well again howsoever you reordered them you will not see that.

So, let us say we have  $x = 1$  because one of them will execute first. So, with no loss of generality let us say  $x = 1$  executes, then let us try to read the value of  $y$ . So, this value will be read to be 0 fine fair enough, but after that we will have to do this which is  $y = 1$  and then read the value of  $x$  in  $t_2$ , but this will be 1 because this instruction is already executed.

So, you can try it the other way. So, you can always read. So, in this case what is happening?  $t_1$  which is  $y = 1$ ,  $t_2 = 0$  you can always have another execution where we will read  $t_1 = 0$  and  $t_2 = 1$ , we can always have another execution where we do  $x = 1$  first  $y = 1$  because as I said they can interleave in many different manners and then we read  $t_1 = y = 1$  and  $t_2 = x = 1$ .

But this outcome that both are 0 this does not seem to be possible. How is it possible? One of them one of these two instructions has to execute first and then at least one of these two has to read 1 both cannot read 0, but as I said common sense is not that common and nothing is sensible when it comes to multi cores. So, let us take a look at the internal out of order pipe lines of core 1 and core 2 that will give us the answer and incidentally you can write this piece of code in openMP or P threads and I invite you to run it on an arm or x 86 machine.

So, the normal standard way that you would do it is that you write this piece of code and run it a million times and just record the outcomes, I claim that you will see this outcome the question is why?

(Refer Slide Time: 26:20)

One More Example

Thread 1	Thread 2
<i>commit</i> $x = 1$	$y = 1$
$t1 = y$	$t2 = x$

*(early)*  $\circ\circ\circ$   $\downarrow$

? Can we read  $\langle t1, t2 \rangle = \langle 0, 0 \rangle$  ? *ooo execution*

**Yes!** The loads are issued *early*, and the writes happen at *commit time*.

*Code*  $\rightarrow$  *Many outcomes*

*Sense*  $\rightarrow$  *No sense*  $\rightarrow$  *[ooo, lsa, lwb, mhw, mc]*  $\rightarrow$  *No sense*  $\rightarrow$  *[Dist. cache]*

NPTEL

McGraw-Hill | Advanced Computer Architecture

32

So, I have a big yes over here with an exclamation mark. So, the question is that consider this when does the store happen? Store happens at commit time when does a load happen even though a load is coming after a store the load will actually execute early when its address has been resolved and the other LSQ conditions all of them hold that is when a load is going to execute early.

See in a certain sense the load will overtake the store because we have an out of order machine and in any case they are two different addresses. Had they been to the same address? Yes, you could have forwarded, but in this case they are two different addresses. So, the load will overtake the store in a similar manner the load will overtake the store and both low and behold are going to read 0.

So, this outcome in any system which has out of order - execution where loads are sent early and you have an LSQ to take care of dependencies, you will see this outcome. So, we will give a formal name to this it's called a write to read ordering, but I do not want to confuse you right now.

So, I am not giving it a formal name, but the truth of the matter is that regardless of how non intuitive we think the outcome is, we will see the outcome on a multi core x 86 or arm machine and you are welcome to try it and as I said use openMP or P threads write the core run it for a million times and then just collect all the outcomes. So, you will definitely see this as one of the sets of outcomes.

So, that is the fun part. So, where are we where we are being basically that regarding a multi-threaded core here is what we can see. So, assume we have multi-threaded core, we will have many outcomes. Some of these outcomes make sense some of these outcomes do not make sense. Now the thing is that why do these outcomes do not make sense?

Well, that is because we have a complicated set of out of order processors we have LSQs we have write buffers we have mshr's we have an NoC, god knows what we have thousand other things? And to make matters even worse we also have a distributed cache in the place of a shared cache and that makes matters even worse.

So, given that we have all of these things what we will actually read and what outcomes make sense becomes a very important question to answer and further more we have placed a requirement that we want a distributed cache to act like a shared cache which at the moment it does not appear to be acting in that manner.

But we will see how to make it act as a single shared cache and also for when you are dealing with multiple addresses. So, the previous discussion regarding distributed and shared is like a single address, but even in multiple addresses something needs to be done. Otherwise, what will happen is that if we run something on a parallel machine on a multi core machine we might end up with really bizarre outcomes for our programs none of our programs will work.

(Refer Slide Time: 30:06)

**Set of Valid Outcomes**

- Given a parallel program, what are the set of valid outcomes?
- This depends on the system on which the program is running. It depends on ....
  - The pipeline
  - Memory system
  - NoC
  - Memory controller (for off-chip memory)
- Every processor has a set of specifications that specify the allowed outcomes/behaviors. If the behavior is consistent with the specifications, the execution is said to be consistent.

This specification is known as the memory model or memory consistency model.

NPTEL

McGraw-Hill | Advanced Computer Architecture

33

So, the key question that we should ask is given a parallel program what are the set of valid outcomes and all of these entities the out of order pipeline memory system NoC and memory controllers for the off chip memory all of them can place foils pore. So, let us have a little bit of a legal definition over here to provide some order to such a disordered word. Every processor has a set of specifications let us specify the allowed outcomes and behaviours.

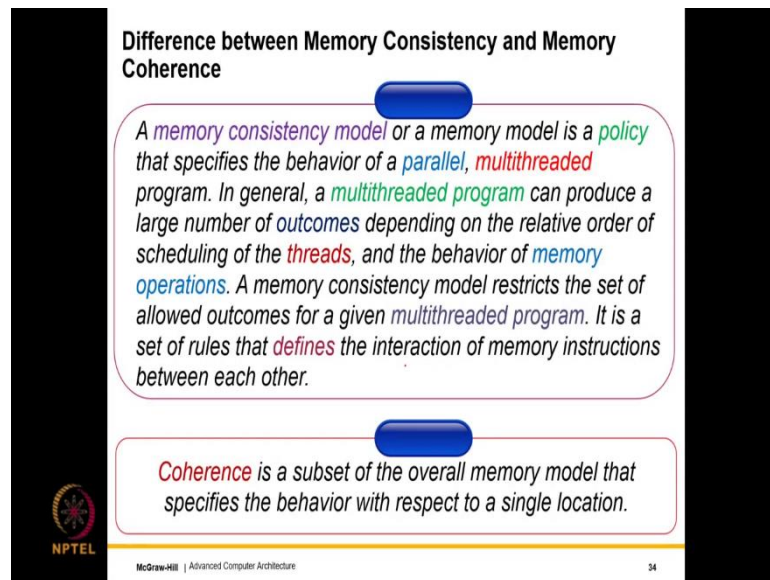
So, these set of specification specify what is allowed as an outcome and what is not allowed. So, in a certain sense it creates a sense and no sense boundary or specifying what is allowed and not allowed and so, we say that if the outcome or the behaviour is consistent with this specifications, then the outcome is consistent otherwise it is not consistent. So, any execution, execution means the set the values that all are instructions read and the values that all are instructions write, this is said to be consistent if it is if it follows a known specification.

So, this specification is issued by the designers of the processor. So, let us say if Intel is designing a processor, it will release its specification this is known as the memory model Intel's memory model, IBM's memory model or the memory consistency model which says that look I have a parallel program, these are the rules for deciding the set of valid outcomes if an execution is consistent with these rules it is a consistent execution else it is not. And needless to say if I run any program on an Intel machine it will always produce a consistent execution.

If it does not the yes that then there is a bug, but if we assume that there are no bugs then it will always produce a consistent execution which is consistent as per Intel's memory model. So, the memory model of Intel IBM and R these memory models are different, but as I said it's a specification which says either something is according to the memory model or we can say memory consistency model or it is not.



(Refer Slide Time: 32:43)



**Difference between Memory Consistency and Memory Coherence**

A **memory consistency model** or a **memory model** is a **policy** that specifies the behavior of a **parallel, multithreaded** program. In general, a **multithreaded program** can produce a large number of **outcomes** depending on the relative order of scheduling of the **threads**, and the behavior of **memory operations**. A memory consistency model restricts the set of allowed outcomes for a given **multithreaded program**. It is a set of rules that **defines** the interaction of memory instructions between each other.

**Coherence** is a subset of the overall memory model that specifies the behavior with respect to a single location.

NPTEL

McGraw-Hill | Advanced Computer Architecture

34

So, now a little bit of this legal definitions. So, this is looks like a paper of log, but well it's not even though it's something not all that different because we are talking about specifications. So, basically we will discuss consistency and coherence. So, a memory. So, I am just reading it out, a memory consistency model or a memory model is a policy that specifies the behaviour of a parallel multi-threaded program.

In general, a multi-threaded program can produce a large number of outcomes depending on the relative order of scheduling of the threads and the behaviour of memory operations. A memory consistency model restricts the set of allowed outcomes for a given multi-threaded program, it is a set of rules that defines the interaction of memory instructions between each other.

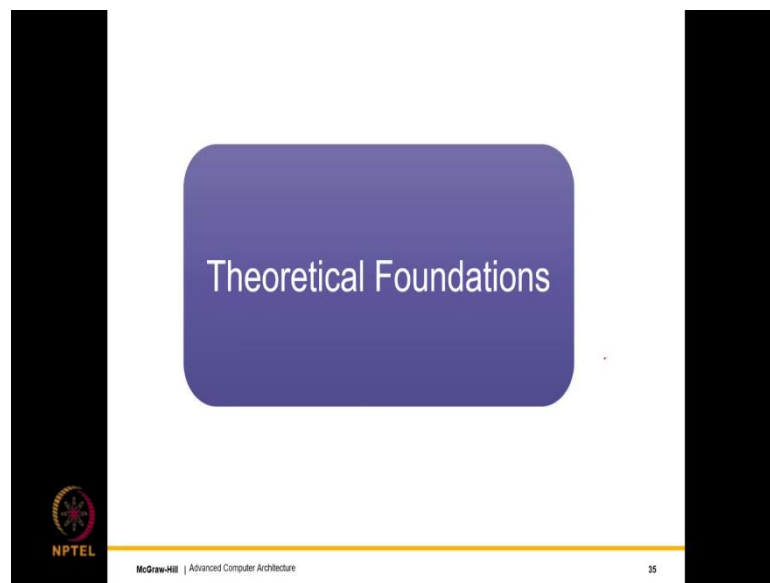
So, the nut shell of this huge definition basically is that a memory model or a memory consistency model just gives a set of rules of how you determine the valid outcomes of a program and needless to see any program running on IBM's processes will follow IBM's memory model.

Coherence is just a subset of the overall memory model that specifies the behaviour with respect to a single location and coherence is typically used along in work. So, even the coherence is a generic term, but it is typically used for equating a shared cache with the distributed cache, in the sense we say that for an external for an external observer a shared cache and distributed cache should look alike which means that even if there are multiple

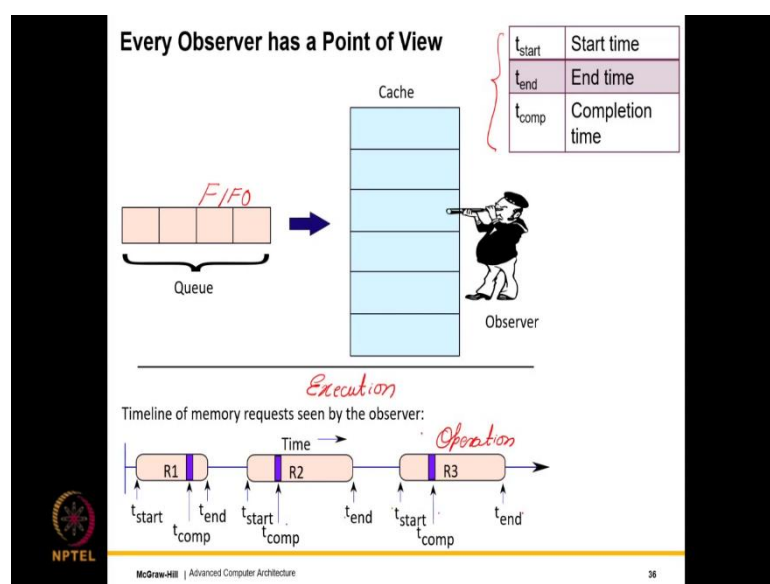
replicas of a variable within a distributed cache all those disparate physical locations should appear to be a single location to an external observer.

So, this is cache coherence when we are restricting coherence to caches and as I said most of the time we use the term cache coherence, this coherence is not really defined otherwise even though nowadays with other memory structures coherence is being defined in terms of them, but traditionally we have only talked about cache coherence.

(Refer Slide Time: 35:14)



(Refer Slide Time: 35:20)



So, now we will get deeper into theory and do more of its like theory within theory. So, now I will discuss some theoretical terms I would ask you to go slow on this part, but if you think about its kind of easy, but as long as if you get a hang of it its even easier. So, look at this as a cache. So, let us say I place an observer.

So, this notion of observer is very important who is sitting where and observing and recording what. So, what will happen is, we will have a set of requests that come to a given location within a cache and they can be put in a first in first out cube a FIFO cube and we will define 3 terms. So, starting time which means the time in which the request starts.

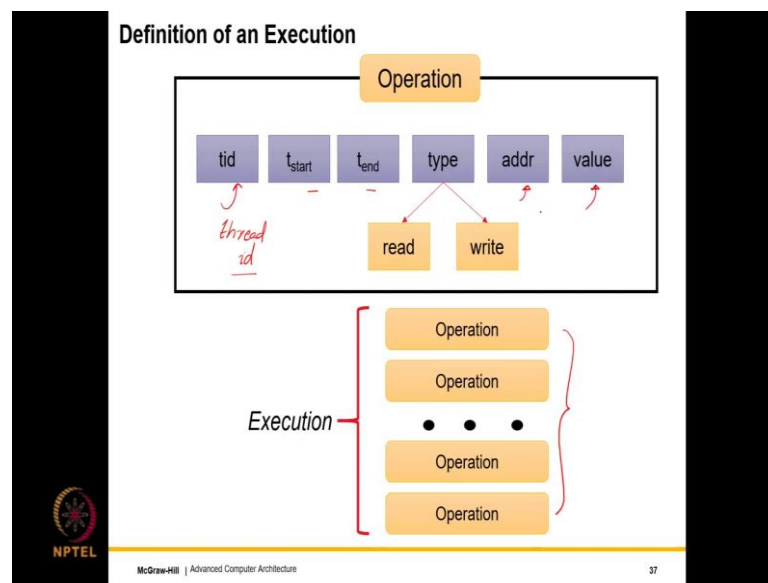
The ending time the time at which it ends right and the completion time which means the time at which it appears to execute appears to fully execute. So, read would be when pretty much the value has been latched value has been sensed and it has been lashed into the buffers of the cache, write is when we actually the values in the s ram cells change.

See if I were to draw a timeline at the bottom the timeline of memory request seen by the observer, we would see that every request has a starting time which is when maybe we can say it entered the queue, it has the completion time when the actual read or write happened on the array and it is an ending time when finally, it is removed from the queue and we clean up its state.

So, similarly we have a starting time a completion time ending time completion time being in between starting time completion time ending time and clearly for the same location when we are when we have a single port, there is no overlap between these. So, the overlap between these is actually not there fine. So, start comp end; start comp end; start comp end and so on in a non-overlapping manner.

So, this is what somebody sitting on this memory location in a single cache would actually observe. So, this is the point of views define as the point of view of this observer as you can see.

(Refer Slide Time: 37:35)

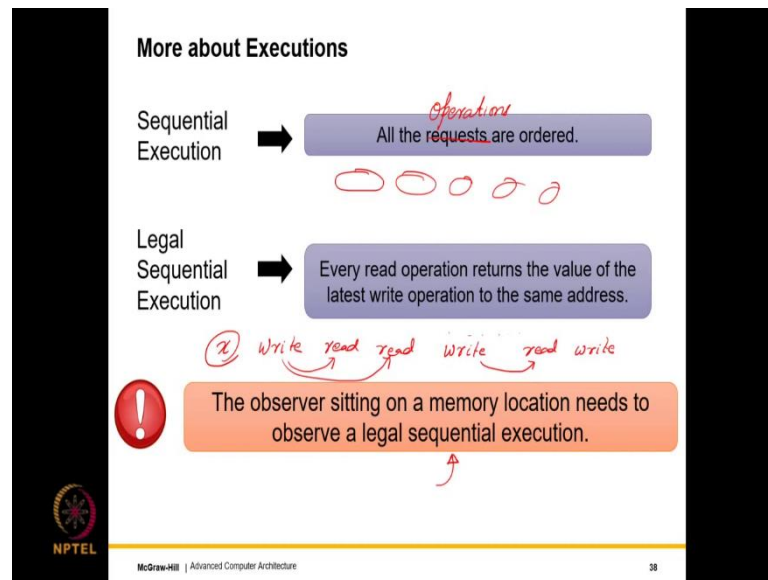


So, let us now define formally based on what we have just seen what is an execution? So, the execution is essentially a set of operations just a list of operations where an operation contains 6 fields. So, one `tid` we can say this is the thread id the thread that is executing the operation the start time end time the type of the memory requests whether it's a read or write what is the address?

And in the case of a store what is the store value in the case of a load what is the value that is read, in the case of a store what is the value that is written. So, each of these. So, this we can think of this timeline we can think of it as an execution, where each of these circular rectangles can be thought of as an operation.

So, in execution as you see is a list of operations and each operation has a thread id a start and end time a read write type and address in a value and we have correlated in an operation sorry correlated an execution with the figure over here.

(Refer Slide Time: 39:03)



So, let us now talk a little bit more about executions. So, let us talk of a sequential execution. So, here all the operations they are ordered. So, ordered means that this comes first and first second third fourth fifth sixth seventh eighth.

So, there is some theoretical terminology, but I do not want to introduce that and its actually called a bijection with the set of natural numbers, but let us not get into that. So, its all the operations are ordered. So, it is a sequential execution where I can say it is an operation 1, 2, 3, 4, 5, 6 just ordered what is a legal sequential execution? So, in a legal sequential execution. So, what we will have? We will have may be read.

So, they are ordered. So, in a sequential execution they are ordered. So, we will have something like this. In a legal sequential execution every read operation returns the value of the latest write operations the same address. In the sense if I read I will get the value of this write if I read over here I will get the value of this write.

So, what I do is that I kind of walk back in a sequential execution, I find the latest write to the same address and then. So, let us assume that all of this systems same address x. So, I just walk back find the latest write to the same address and I forward the value, this would be a legal sequential execution. So, you would expect that the observer sitting on a memory location. Let us assume for the time being that there is a single memory location for every variable a single location that observer needs to observe a legal sequential execution.

(Refer Slide Time: 41:35)

So, in the case of a load what we see is that this relationship still holds if the observer is sitting on a core. So, now, what we do is that we change the location of the observer instead of the observer sitting on the memory location within the memory system, let the observer sit on the core and the core has an out of order pipeline needless to say.

So, complete in this case means, that it will write its value such that all the other cores can see. So, that is the most important point what will happen is that it will write its value to the memory system such that all the rest of the course can see the value. So, we really do not know when that will happen.

So, what will happen is that we might commit the store, but the store might be stuck in a write buffer or the store might be stuck in an NSHR or in some message on the NoC and it might not have reached the other cores. So, unless we have some way of getting an acknowledgement which in most cases we do not. The store will I mean the store in a might reach its final destination a long time later.

But as far as we are concerned the operation ends when we when the store leaves ROB. So, we can very well have  $t_{start} < t_{end} < t_{comp}$  let me go back a few slides. So, this is exactly what was happening is that, the completion time of a store was happening later, but a later load was completing earlier.

So, this was again an artefact of our pipeline and what you get to see in this later slide is that things can actually be slightly worse in the sense even after a store leaves, it would not have fully completed what that basically means is that, the even if at that point of time other cores try to read the value they might still get the previous value because even if one core writes and let us say each core has a private L1 cache until the store reaches all of them we are not deemed to have completed.

So, this will happen much later after it leaves the pipeline. Nevertheless, the observer on the core right the observer on the core expects to see a legal sequential execution in the sense; in the sense that if let us say the core has written some value and later on the core reads it, it expects to either read that value or a later value, but not something earlier. So, let us say that I am sitting on a core over here and I am the observer and let us say that I set  $x$  to 1 and a long time later I read  $x$ .

For me given the fact that I set  $x = 1$ , I cannot read  $x = 0$  I have to read either  $x = 1$  or something else that some other core wrote in the time being. So, I am fine with that, but I can definitely not read  $x = 0$  see in this sense I do expect to observe a legal sequential execution otherwise the execution will not make any sense. So, what is the idea? The idea is that if let us say I am looking at my own execution of course we will be in sequence and I expect every read to fetch me the value of the latest write.

So, let me for the time being ignore the effect of other threads, if I am the single thread running then I expect to see a legal sequential execution in a unit threaded core that I expect that the way that the assembly instructions the programmer has written the core executes

this then this, then this, then this, then this regardless of the underlying hardware whenever I read something, I get the value of the latest write.

At least if there is a single thread this is what I would expect otherwise the execution will stop making any sense to me and the typical assembly programs that we write they will just stop working because there are two important if you think about it theoretically for a program, there are two assumptions that we make.

If these are the assembly instructions we execute the first one then the next, then the next, then the next, then the next and so on whenever we read we get the value of the latest write. These are the only two assumptions that we make for a single threaded program even if you have other cores this abstraction still has to hold true.

(Refer Slide Time: 47:14)

**Parallel Executions**

Consider a parallel execution.

Term	Meaning
Rx1	Read the value of x as 1
Wy2	Set y = 2

Multiple threads: one observer per thread. Each observer records the local execution history of a thread.

NPTEL

McGraw-Hill | Advanced Computer Architecture

40

So, let us keep moving. So, now, let us consider a parallel execution where we have multiple cores. So, in this case we could have had multiple cores, but we were still looking at a single thread, but now we will look at multiple cores each running a thread. So, multiple cores and multiple threads.

So, we are slightly broadening our scope. So, let me just say from where we are coming. So, we looked at a single threaded execution and we looked at an observer on a cache location, it needs to see a legal sequential execution then we considered a setup where we have a single thread the observer sets on the core, regardless of the number of cores in the



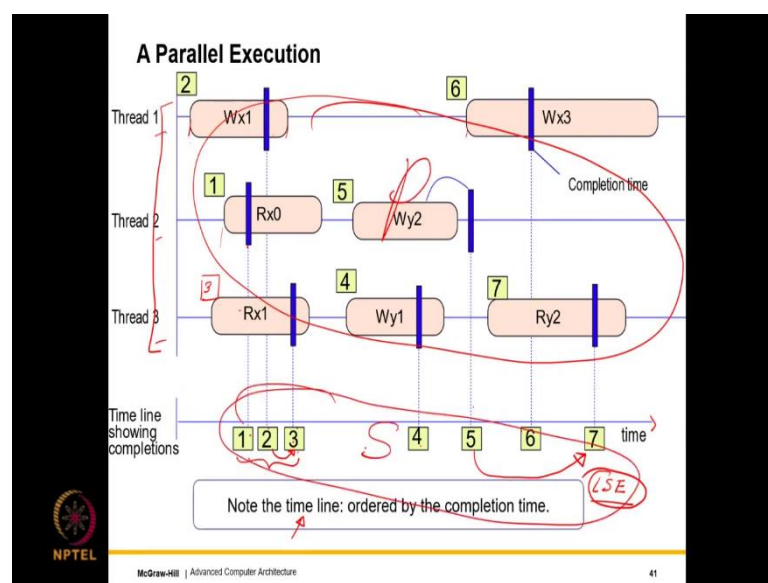
system for a single threaded program I have to see a legal sequential execution because for me this make sense.

In this case, an instruction is an operation. See if I have 20 assembly instructions, I observe that one instruction first instruction is fetched second is fetched third is fetched fourth is fetched. So, I can order them by their start times which are their fetch times and I would expect them to finish in that order and then whenever I read something I want the value of the latest write that is all that I expect. If that holds my execution from the memory point of view is correct.

But if I consider a parallel execution things become trickier. So, that is why I asked you to drink a cup of coffee, where we will have a multiple cores and we will have. So, let us say we have  $n$  cores and we have  $n$  threads  $i$ th thread running on the  $i$ th core. So, let us introduce a little bit more terminology let us say  $R_x 1$  means that I read the value of  $x = 1$ ,  $W_y 2$  means that we set  $y = 2$ . R for read, W for write  $x$  is the name of the variable,  $y$  is the name of the variable.

In this case, I am reading 1, in this case I am writing 2. So, in multiple threads I have one observer per thread these are software threads each observer records the local execution history of a thread. So, I will have a multiple observer each observer just records what it observes.

(Refer Slide Time: 49:42)



So, let us look at a parallel execution. So, what can happen is that, I might write one to x I will have if I have a multiple variable I might doing a lot of stuff reading writing reading writing and so on. So, let us not worry about that let us say the start of a rectangle is the start time the end of a rectangle is the end time start time end time start end and the blue is the completion time.

So, the completion time can be before the ending time of a write it is possible in an out of order pipeline also if the write happens instantaneously and slightly later we remove it from the pipeline may be like a fraction of a cycle or 1 cycle later. So, the commit process itself can take some time. So, does not matter whatever is the completion time we have individual observer sitting at the threads which is fine.

So, what we can do is that let us look at the outcomes and let us see that can be order them. So, here what we are assuming is that, every operation as you can see over here has the completion time and the completion time is known. So, this is the big assumption that we are doing that we somehow our observer knows when an operation is completing.

And we are assuming that it at the point of completion, it appears to execute instantaneously which means that it appears to read exactly at this point of time, it appears to write exactly at this point of time and if you consider a practical system this is more or less the case that when you read, reading is basically a fraction of a cycle affair in the sense you do everything set the decoder and all of that, but when you read you basically latch the values from the sense amplifier to the buffer.

So, it's just capturing that moment and writing is also when you set the values of the SRAM cells its capturing that moment that is the completion time. So, what we can do is that if all the observers see what the course executed, then they all have a conference at the end and they order everything as per the completion time. So, then they would order Rx 0 over here fine then Wx 1 completed then Rx 1 completed.

Till this point the execution is sequential because we can order the operations one after the other as long as we can order them one after the other which we have done here fine its sequential is it legal yes still now its legal because 3 is reading the value written by operation 3 is reading the value written by operation 3.

Then we will have instruction 4 where we write  $1 = y$ , then we write  $2 = y$ . So, in this case, the completion time is outside the start and end that is ok then we again write  $3 = x$  over here and then we read  $y = 2$ , which means it gets the value of the latest.

So, this is a legal sequential execution as we see its an LSE why because we were able to order all the operations even though they were issued by different threads that is important keep that in mind bear that in mind that even though these were operations that were issued by different threads we were able to order them by their completion time. so, that made it sequential.

And further more when we ordered them on the same timeline, we see that for any variable the read is getting the value of the latest write. So, this makes this a legal sequential execution. So, we have a parallel execution at the top we ordered them by their completion time. So, we are assuming that we somehow know what the completion time is. So, it becomes a sequential execution and we are very happy with it because it is legal also.

So, let us not look too much into it let us keep going. So, what you need to note is the timeline, where it is ordered by the completion time. So, this is an important point to note we will keep on revisiting this, this is a simplistic scenario, but nevertheless this has been our first parallel execution where we have placed multiple observers at each of the cores observer are the core and observer are the thread let us assume as the same thing and then what we do is that we just see what they read and write we arrange them by their completion time we find it to be legal and sequential.

(Refer Slide Time: 55:01)

**Issues with Parallel Executions**

- **Atomicity** → Every operation has a single **completion time**. It **appears** to execute **instantaneously** at that point of time.
- Have **multiple observers** (one per each thread) with their **points of view** does not help.
- It is better if we can **think** of parallel executions in terms of sequential executions → It is very intuitive.
- For this, we need to introduce the notion of the equivalence of **executions**:

Handwritten notes and diagrams on the slide:

- A red circle around the text "P and S" with an arrow pointing to the "S" in the list item.
- A red arrow pointing from the "S" in the list item down to a handwritten "S".
- A red arrow pointing from the "P" in the list item down to a handwritten "P".
- A red arrow pointing from the "P" down to the "S".

NPTEL logo and footer text: McGraw-Hill | Advanced Computer Architecture 42

So, what did we learn from the execution shown in the previous slide? Well, what we learnt is that we have a general property it's called atomicity which means that if an operation appears to execute instantaneously at one point in time, it is said to be atomic.

So, we observe this property whether it holds for all architectures or not we will look at that later, we also found that we have multiple observers in the sense we have multiple observers. So, having multiple observers really does not help because one observer does not know what the other guy is observing.

Of course, if all of them sit together and create some kind of a timeline like this that helps because we can at least see how the system proceeded. So, that definitely helps. So, that is why at the end they can have a conference and they can reduce or equate the parallel execution to a sequential execution makes things extremely intuitive as you can see do not you think that this is very intuitive? It is right.

And for this we need to introduce the notion of the equivalence of executions. So, particularly what we did over here let us formalize this. So, we have a parallel execution. So, this entire thing is a parallel execution this is a sequential execution. So, a parallel execution as such we did not find a lot of value, but once we equated it to a sequential execution somehow as we did in the previous case by ordering them by the completion time, we were able to show that look the sequential execution is legal and sequential.

So, it makes sense. So, what does this mean? What this means is that this gives the most likely order well in this cases are most likely, but it's the exact order in which these operations would have actually executed on a multi-threaded multi core system.

On the multi core system this is how they would have actually executed and given the fact that it's a legal and sequential execution, this execution is indeed possible and feasible and it is also correct howsoever we think about it is also correct because well I mean for every memory location as you can see it is getting requests ordered by these blue completion timelines over here, it is servicing them we are reading what we should be reading.

So, it is correct. So, let us take this further and just equate see when we can establish an equivalence between a parallel execution and a sequential execution, it will allow us to think about parallel executions in a much better and more intuitive manner.

(Refer Slide Time: 58:20)

### Equivalence of Two Executions

Equivalence of two executions

Expression	Meaning
$P \mid T$ <i>thread</i>	All the operations issued by thread T (in the same order). This is an ordered sequence.
$P \mid T \equiv S \mid T$	There is a one-to-one mapping between the two sequences of operations.
$P \equiv S$	For all T, $P \mid T \equiv S \mid T$

*Handwritten notes:*

$P \mid T$        $S \mid T$        $O \rightarrow O \rightarrow O \rightarrow O$  (with a circled 1 above the first O)

$P \mid T: 1 \rightarrow 2 \rightarrow 3$        $S \mid T: 1' \rightarrow 2' \rightarrow 3'$

$S \mid T: 1' \rightarrow 2' \rightarrow 3'$

McGraw-Hill | Advanced Computer Architecture

43

So, let us now theoretically describe how to establish the equivalence of two executions. So, this is a standard result in the area of concurrency systems it's very easy to understand. So, whatever we trying to do? We are trying to say that a parallel execution with multiple threads and observer per thread is equivalent to the sequential execution where of course, the operations are arranged in a linear sequence.

So, we want to create some notion of mathematical equivalence. So, for this we need to introduce some terms. So, first is P the straight line T. So, you can say P conditioned upon

T or the set of T is a thread and P is a parallel execution. So, this basically refers to all the operations issued by thread T in the same order. So, what would happen? Let us go back to the parallel execution. So, let us say thread 1, we can see all the operations issued by it, there is a sequential order of at least of all the operations issued by single thread.

So, for thread 1 all the requests all the memory requests or let us use the generic term operation they are ordered in a linear sequence. So, if we consider only those operations issued by a given thread in the same sequence in which the thread had issued them this ordered sequence is T straight line T. So, it is given by this expression.

So, we say that this expression which is all the ordered this is the sequence of all the operations issued by thread T this is equivalent to. So, we will use the 3 lines to indicate equivalence of S T; where S T S is again the sequential execution, but out of the sequential execution we take out all the executions all the operations that have been issued by thread T. So, let us say that we have a set of operations in the sequential execution and they are organized as a linked list.

So, out of this let us say this and this were issued by a thread 1 and we want to compute this and what we do is, we just take these operations out and we organize them as a linear sequence in the same sequence in which they were in the original. So, they are just extracted.

So, this is just a projection operation we can see P projected upon thread T and S projected upon thread T. So, all that we do is that we just extract out all the operations issued by thread T and given the fact that they were in any case in an ordered sequence they are extracted out in an ordered sequence.

So, now the thing is that the operations of course, need not be the same in the sense when they are when we are seeing their equivalent, let us treat them as a different set of operations, but what we can do is, we can establish a one to one mapping between the two sequences. So, let us say one sequence of operations 1, 2, 3, 4 and so on in other case we can say it is 1', 2' and 3' where there is a one to one mapping between 1 and 1', 2 and 2', 3 and 3'.

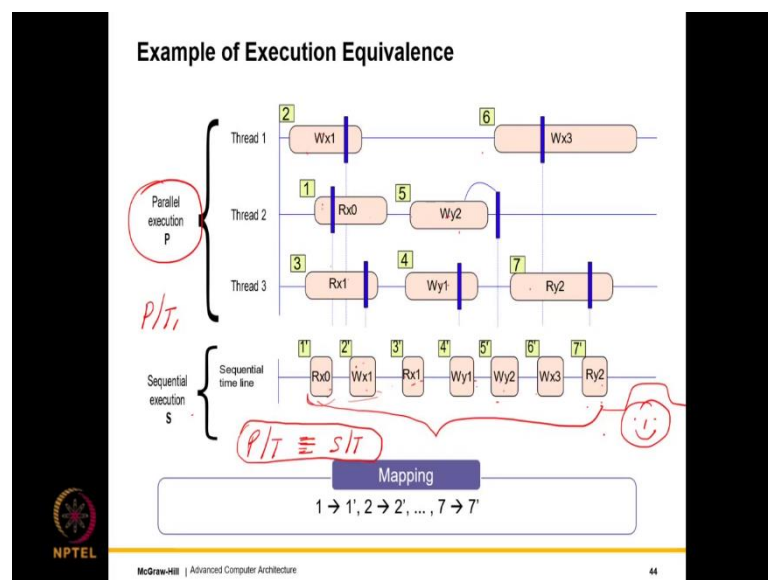
So, this what we are doing? Well what we are doing is we are taking the parallel execution extracting all the operations by a thread T again taking the sequential execution extracting

all the operations by thread T and then we are establishing a one to one correspondence between the two sequences. Say if we can do this for all threads for all threads T if we can establish an equivalence of this type, then we say that the executions are equivalent.

So, there is a lot of maths over here this is simple stuff, but nevertheless for many of you who would have seen this kind of algebra kind of math modern algebra kind of math for the first time, this would appear a bit weird and a bit strange, but that is the reason I would ask you to take this thing slow read this several times and definitely read this part in the book. So, that is important.

So, this will be clear when we consider an example which we will do in the next slide which will be the last slide of this lecture.

(Refer Slide Time: 63:24)



So, let us look at the same parallel execution as we have done just to verify it's the same just take a quick snapshot of this, we have seven 1, 2, 3, 4, 5, 6, 7 and just take a quick snapshot of this 1, 2, 3, 4, 5, 6, 7. So, here what we essentially did is that, we had a set of operations of a parallel execution  $P$ .

So, what we do is that we create a sequential execution. So, mind you the sequential execution operations can be different, but they have to read and write the same values. So, there is a one to one correspondence. So, the one to one correspondence means that is the

same operation reading and writing the same value. So, this one to one correspondence does exist.

But of course, their timings and their durations those we are allowed to change. See we consider let us say P and thread 1 it is 2 and 6. As you can see we have 2' over here which is the same Wx1 we have 6 over here Wx3 6 appears after 2. So, will 1 and 5 we have 1' and 5' over here 1e is Rx 0 Rx 0 5 is Wy2 this is Wy2.

Then we have 3, 4 and 7 we have 3, 4 and 7 where we have Rx1 y 1 and Ry2. So, as you can see for all threads P T 1. So, for ever any thread actually 3 lines S T which means I extract the operations from there they are over here in the same linear sequence and there is a one to one mapping between them. In the one to one mapping we again read and write the same values, but of course, the timing is what changes in the sense you just took like take a look at the time of 2.

So, the timing actually does not matter as long as they are arranged in a sequence. So, what matters is the sequence the ordering. So, here 6 appears after 2 here 6' appears after 2' that is all that we care and one is mapped to 1', 2 to 2' 7 to 7'. it's a one to one mapping. So, it's a bijection actually. So, where we will have the same number of elements and it's a one to one mapping and for any thread the operations appear in exactly the same order in both the sequential execution and the parallel execution.

So, this is where we see that the executions are actually equivalent. And so, then what is the advantage that we got out of this? One advantage that we got out of this is that, what we are saying is that we give these parallel executions if let us say we have one global observer who can see all the events everything that is transpiring then of course, this is ordered by completion time.

But let us assume the completion times are not known and in almost 100% of the practical situations we do not know what is the real completion time, but it does not matter it is possible to arrange all of the memory operations in a sequence such that it is a legal sequence. In the sense for any address the read fetches the value of the latest write and it is a possible order in which these operations should have actually completed in a real system as observed by an all-powerful single global observer, it would have observed that globally this is how all of them are completing.



So, this is the feasible execution sequence for the entire system for the entire parallel system the moment I equate it to a sequential execution. So, the moment I equate a parallel execution to a sequential execution of this type where of course, this relationship this equivalence holds this essentially gives a feasible execution order of how these memory operations would have actually executed.

And you can see that it does satisfy at least our criteria of intuitive them in the sense we can say that look first this instruction of thread 2 happened then this happened, then this, then this, then this, then this and look I can arrange them in a linear sequence.

And see this makes complete sense because I read the value of  $y = 2$  because the last write to  $y$  was 2 and then I read the value of let us say  $x = 1$  because the last write to  $x$  was 1 so on and so forth. So, what I can do is that I can reason in this manner in this fashion.

And I can argue about the correctness of an execution by essentially reducing a parallel execution to an equivalent sequential execution this being the equivalent and if the sequential execution I think is correct. Then I can at least say that look I think this execution is correct and I think this outcome is valid and if I cannot create a sequential execution as we have seen some examples in the past, then I can say that look I think this outcome is not correct it is not intuitive.

I can always say that, but as we have seen in real processors sometimes they allow executions of this type which do not appear to be intuitive in the sense the parallel execution cannot be reduced to an equivalent sequential execution, but whenever we can do what we just did, we will be happy we will have a big smile on our face and we will say that look I think this is a very intuitive execution which in my opinion is correct.

So, this lecture I will stop over here because we have discussed several things even though the situation is far complete.

(Refer Slide Time: 69:49)

**Issues with Parallel Executions**

- **Atomicity** → Every operation has a single **completion time**. It **appears** to execute **instantaneously** at that point of time.
- Have **multiple observers** (one per each thread) with their **points of view** does not help.
- It is better if we can **think** of parallel executions in terms of sequential executions → It is very intuitive.
- For this, we need to introduce the notion of the equivalence of **executions**: P and S.

*Handwritten notes:*

- **Complex world**
- **All possible outcomes**
- **Specification [Consistent]**
- **Sequential execution**
- **legal**
- $P \equiv S \rightarrow \text{legal}$

NPTEL | McGraw-Hill | Advanced Computer Architecture | 42

So, let me quickly summarize all that we have learnt and how we are going to proceed. So, what the first point that we have learnt is that, it is a very complex world within a multi core processor. So, we have write buffers mshr's on chip networks distributed caches you name it we have it. Given that we can have all possible outcomes including once that make no sense at all. it is possible if we want performance we add these new structures and they give us the strange outcomes for multi-threaded core not for single threaded, but from multi-threaded core we have these strange outcomes.

So, there is a need for some sort of a specification and some sort of an order to a disordered world. So, we that is the reason processor manufactures release specification and if an execution is as per a specification, it is said to be consistent finds fair enough. So, then we looked at this world of executions and we said that look if we have a single threaded program we will observe a sequential execution or observer can be on a memory location or can be on a core does not matter.

The execution will still be sequential and it will also be legal in the sense every read will fetch the value of its latest write. If we have multiple threads the situation becomes very complicated, but if we can establish an equivalence between a parallel and a sequential execution that is going to be great then in that case we can actually reason in terms of the sequential execution.

So, it will be easier to make a specification and further more if this sequential execution is legal, then we can be doubly sure about its intuitiveness and correctness. So, we have not significantly discussed this we are stopping at this point where we are seeing what it means for a parallel and sequential execution to be equivalent and we are not going more than that, but this is not answering many other queries in the sense how do we make a specification, what does it mean to be consistent with this specification all of that.

But we are leaving you at this point where we are saying that look a parallel execution by itself with a complex thing to analyse. If it can be reduced to a sequential execution, then at least it is the sequential execution is something that we can possibly reason about and analyse. So, let us keep this as our aim of converting a parallel to a sequential execution which may be legal might not be legal we will see what that means, over the next few slide sets, but at least let us have this as our aim and let us proceed.

So, what is our aim? Our final aim being to establish some sort of a control some sort of an order over what seems to be a very disordered situation where all possible executions and outcomes and messages are allowed we need some sort of a correctness framework. So, we have taken some baby steps towards it, we have looked at execution, we have looked at a parallel execution and we have looked at what it means for it to be equivalent to a sequential execution. So, over the next few slide sets we will extend this formalism to a full scale memory model.

(Refer Slide Time: 73:47)


### Sequential Consistency

When is a parallel execution equivalent to some sequential execution?

**Sequential Consistency (SC)**

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

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



NPTEL

McGraw-Hill | Advanced Computer Architecture

45

So, this is our last slide. So, next slide set we will start from this point which is sequential consistency. .