


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

Chapter - 09
Lecture - 30
Multicore Systems Part - VI


(Refer Slide Time: 00:38)



Enhancements to the Directory Protocol

Let us list some of the common **problems** associated with directories

- We need an entry for each block in a program's **working set** (lot of storage)
- In each directory entry, we need an **entry** for each constituent **cache** (storage overheads)
- The **directory** itself can become a point of **contention**
- Let us look at →



McGraw-Hill | Advanced Computer Architecture

119

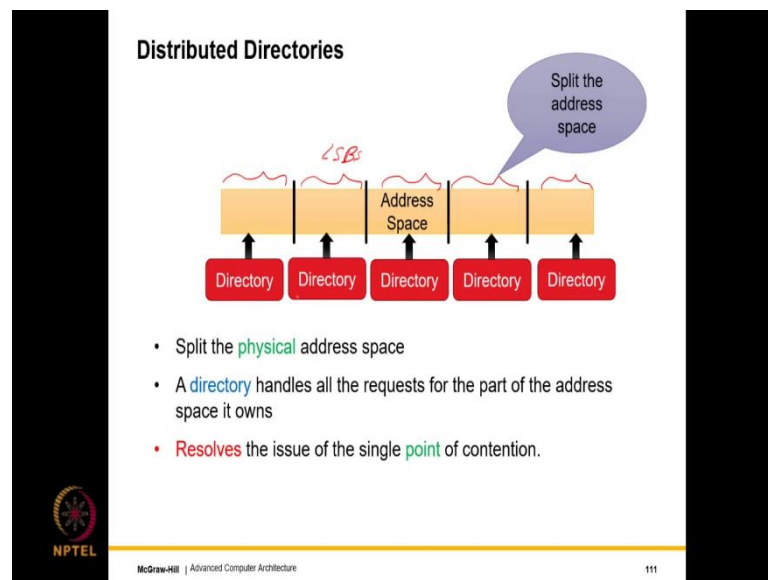
So, let us now discuss enhancements to the directory protocol. So, the directory protocol as was described in the previous lecture is correct, is complete, but it is not efficient. So, let us first list some of the common problems associated with directories and how we will fix them.

So, first is that we need an entry for each block in a programs working set. So, recall that the working set is defined as the amount of memory that the program uses in a short period of time. So, of course, there is some subjectivity in this definition, but there is a somewhat better definition in the book that talks about the short period of time in terms of the phases that a program goes through.

So, a program typically has a settled phase of execution then another settled phase of execution. So, in these settled phases roughly it uses in a constant amount of memory and it works on that, so that is the working set. So, each block over here we need an entry in the directory.

And so, this is required for all the core. So, the directory per say has to be huge in each directory we need an entry for each constituent cache and that adds to the storage of overheads. And given the fact that we have a single directory that itself can become a point of contention, hence we have to look at solutions to make this simple idea an efficient idea.

(Refer Slide Time: 02:12)



So, what we can do is that we can take care of the size and contention problems like this; that we take the physical address space and we partition the physical address space into different disjoint sets. So, this is set 1 set 2 and so on. So, we just partition the physical address space.

So, one nice way of partitioning is instead of partitioning by the MSPs we partition them by the LSPs of the block address that will be far more uniform that way and for each partition we assign a directory. So, we can have multiple directories that is not an issue. And given in the fact that a single block will always be mapped to a directory the same directory correctness is not an issue. So, what we do is that we split the address space and each partition gets mapped to a directory.

A directory will then handle all the requests for the part of the address space that it owns. So, this also issues resolves the issue of the single point of contention in the sense that we have multiple directories that look at different addresses. So, based on the address message can be routed to the correct directory on the n o c and that directory will have a smaller size as compared to a large big combined one.

(Refer Slide Time: 03:39)

List of Sharers

- How to **maintain** the list of sharers?
- Solution 1 [Fully mapped scheme]:
 - If there are N **processors**, have a bit vector of N **processors**.
 - Each **block** is associated with a bit vector of **sharers**

block address 11000000 10001011

Space-efficient Solutions

- **Maintain** a bit for a set of caches. Run a **snoopy** protocol inside the set.
- **Store** the ids of only k sharers. Have an **overflow bit** to indicate that there are more than k **sharers**. In this case, every message needs to be **broadcasted**.

[Partially mapped scheme]

64
 $4 \times 6 = 24 + 1$ (25)

NPTEL

McGraw-Hill | Advanced Computer Architecture

112

So, now let us come to the list of sharers, how do we maintain the list of sharers? So, one is the fully mapped scheme. So, if there are n processors then assume I have one sister cache per processor, I can have a bit vector of these N sharers, capital N sharers where each block is associated with a bit vector of sharers. So, if I have 64 sharers I need to have 64 bits or 8 bytes per block.

So, I can have a slightly more space efficient solution what I can do is that I can have a single bit for a set of caches. So, instead of, 64 I can break it into maybe sets of 4 and have 16 such sets. So, this could be set 1, this could be 1, 2, 3, 4 first 4 bytes this could be set 2, set 3, set 4 and so on and so forth.

And so then if I have a 1 over here for a set it will indicate that at least one of the sister caches within the set has a copy of the block. Then what we can do is; we can run a small snoopy protocol inside the set or we can essentially query each of the caches within the set by sending a multicast message.

Something of this nature is possible. So, there are two advantages; one is that this reduces the storage space at the directory that is point 1 and the second is that this is naturally I mean able to the structure of a modern multicore system. Because the modern multicore system is typically broken down into tiles, where we have like these cores and cache bank combines.

So, where we have a single tile will have a core and a cache bank. So, in a certain sense within a tile they share a bus. So, nothing stops us from having like a local snoopy protocol running here and overall directory protocol. So, that will ensure that in each set we have an order of rights and we also have a global order of rights.

The other is that we store the ids of only k sharers. So, this is like a partially mapped scheme that only for k sharers we stored the id. So, we just say that to we will only store the ideas of k sharers. That is because typically most blocks are not associated with that many sharers. So, maybe we will have like 2, 4, 6 maximum sharers.

So, let us say if we if we do for 4 and so assume we have 64 blocks and we store only for 4 sharers. So, what we need is we need 4 into 6 bits which is 24 bits and after that we can have an overflow bit that stores the fact that there are more than k sharers. So, the overflow bit which will be one additional bit over 24 will be 25 this will essentially indicate that look we have more than k sharers.

So, the overflow bit will the fact that it is being set will indicate this bit but this will be very rare event in the sense we expect very very few blocks to have more than k sharers. So, in this case, we can revert to a mechanism where every message is broadcasted to all the sister caches and this of course, will be much slower far slower, but given in the rarity of this event this is perfectly doable and it is done also it is known as the partial mapped scheme which is reasonably popular.

(Refer Slide Time: 07:31)

Size of the Directory


The **directory** should ideally be as **large** as the number of blocks in the programs' **working sets**

Having an **entry** for every block in the physical address space is **impractical**

Practical **Solution**:

- Design a **directory** as a cache (*set-associative cache*)
- Keep the state of a **limited** number of blocks
- If an entry is **evicted** from the directory, invalidate it in all the caches

Overhead



NPTEL

McGraw-Hill | Advanced Computer Architecture

113

Now, let us come to the size of the directory. So, the directory should ideally be as large as a number of blocks in the programs working sets, you might not have one program you might have many programs. So, the directory should ideally be as large as the sum of the working sets since you have multiple directories of course, the blocks will get partitioned, but at least the directory as a whole should contain all the blocks.


So, of course, having an entry for every block in the physical address is impractical. So, the way directory is designed is that is designed as a cache. Similar to a set associative cache with the same thing that we use certain bits of the block to address the off set index and all of that.

So, it is designed the same way as a set associative cache and it does keep the state of a limited number of blocks because after all it is a finite sized cache and it is organized. So, its organizes exactly the same way as a set associative cache with the tag array and the data array and the tag array in this case is some bits on the block address and the data array in this case will contain the list of sharers and the state.

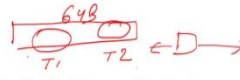
If an entry is evicted from the directory what do we do? So, this is why the size of the directory is important. So, if it is a shared block, so further more we can say that look certain parts of the address space are either read only or private. So, they need not enter the directory in the first place itself. But if let us say it is a shared block then it has to be it has to enter the directory and if it is evicted we need to invalidate it in all the caches.

This is the overhead this is the problem and this is why we need large directories in the first place that we do not need it and this is exactly why we did all of those optimizations. Such that we can store as many blocks as needed in the directories because evicting a block from the directory is expensive in the sense that all the caches that needed they also need to discard it.


(Refer Slide Time: 09:47)



A Few More Issues



- **False sharing** → Consider a 64-byte block. It is possible that different threads access different memory words within the block.
 - The block will keep **bouncing** between caches.
 - These are **false sharing** misses.
 - Use a smart compiler to place data more **intelligently**.
- **Race conditions** 7
 - In real hardware, there are a lot of **interactions**. It is possible that multiple messages of different types for the **same block** might arrive at the same time.
 - Such concurrent events (**race conditions**) need to be handled. Hence, a practical cache coherence protocol has close to a 100 states.



McGraw-Hill | Advanced Computer Architecture 114

A few more issue so we can have a new kind of a miss in a multiprocessor system which we did not have in a uniprocessor system consider a 64 byte block it is possible that different threads accessed in different memory words within the block. So, consider a block like this which is let us say 64 bytes possible that thread 1 accesses these set of bytes and thread 2 accesses these set of bytes.

If they are accessing these bytes concurrently what will happen is that one thread will invalidate the block, then the other thread will invalidate the block. So, the block will just keep bouncing between the threads.

But in effect, so this bouncing will reduce the performance, but there is no true sharing in the sense same set of bytes are not being shared this is false sharing, why is it false sharing? Because we are accessing different sets of bytes and these bytes are disjoint they are not the same, but they happen to be located in the same 64 byte block which is causing the block to bounce because blocks for us are atomic units.

So, to reduce such false sharing misses we need to use a smart compiler that places data far more intelligently and smartly and the smart compiler essentially with an intelligent placement of data can reduce the amount of false sharing. Also when we design a real cache coherence protocol, as we saw in the MOESI protocol. So, in the MOESI protocol actually we did not have 5 states if we include the temporary states we had actually 7 states.

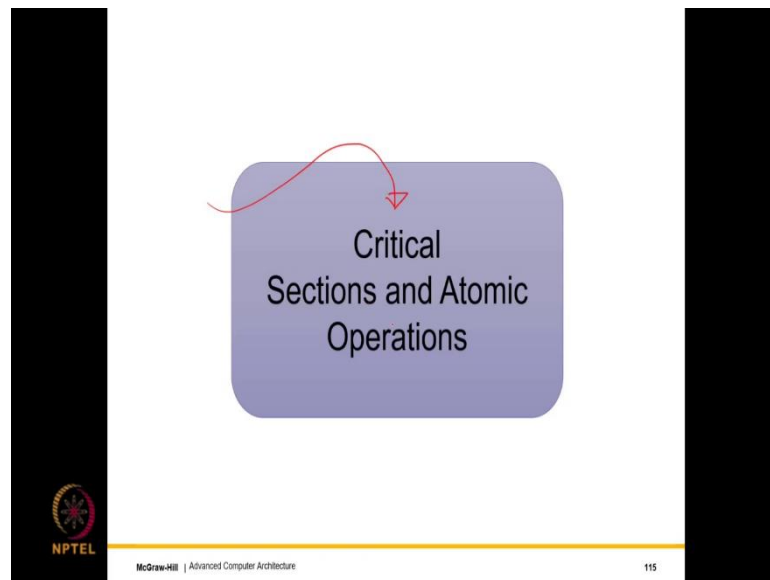
So, there are a lot of interactions in real hardware and lot of events can happen. So, it is possible that multiple messages or different types for the same block might arrive at the same time, along with that we might have context which has we might have interrupts we can have a lot of things. So, to capture all of these interactions and to have a state diagram that captures all these combinations of events which is the concurrent events we need to do a lot.

So, whenever we have such concurrent events. So, let us say one event could be that I want to transition from the m to s state and in the same point of time another. So, I receive a read miss and the write miss at the same time. So, do I transition to the s state or do I transition to the invalid state? And who sends the data to whom?

So, all of these concurrent events to the same set of to the same block which can cause problems these are known as race conditions and all such race conditions need to be handled. Because we do not want a single corner case to kind of derail the entire cache coherence protocol. Hence if we consider all of these things and we design a practical cache coherence protocol you can very well have a 100 plus states.

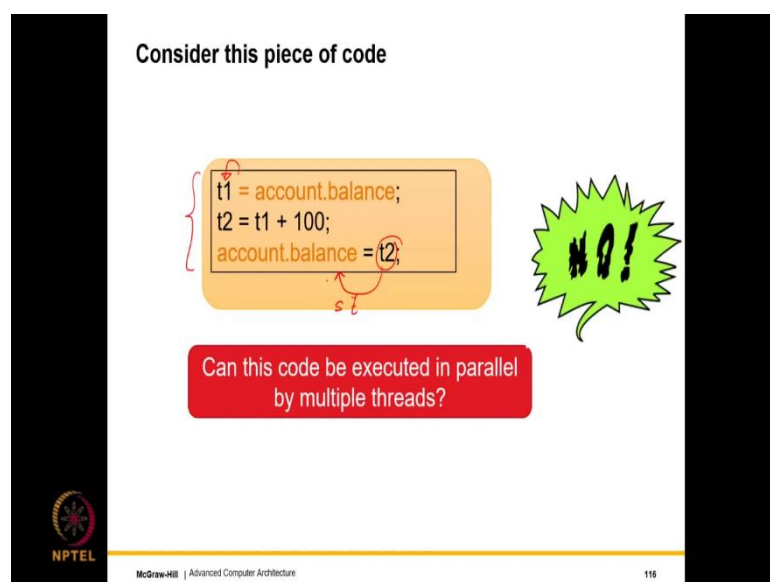
So, it is not unknown and it is not uncommon or unnatural to have 200 plus states in a cache coherence protocol and of course, it is not possible to verify that the protocol is correct. But we can have an automated verify which can take the state diagram and verify if the protocol is indeed correct, in the sense that whether it captures all the corner cases or not.

(Refer Slide Time: 13:25)



So, now we will extend the basic cache coherence protocol to implement something which is very important called a critical section and implement atomic operations which are also sync synchronization operations.

(Refer Slide Time: 13:39)



So, this is a quintessential example that is given that where we want to update the balance of a bank account. So, in this case we just want to add I am using the Indian rupees over here. So, we want to add 100 rupees to a bank account. So, in international viewers replace rupees with dollars or your local currency does not matter.

So, rupee is the Indian national currency it is also said as INR Indian National Rupee and this is the symbol. So, we add two lines like this. So, let us say I want to add 100 rupees to my bank account. So, what we do is we read the balance in the account which is a load operation put it in a register t1 we add 100 to it and then we put the register t2 again back with a store operation.

We want all of these to happen atomically in the sense either all three happen or none happens. So, unfortunately this code cannot be executed in parallel by multiple threads in a sense the multiple threads try to update the bank balance. The same account balance at the same time the execution can sadly be incorrect. Let us see why?

(Refer Slide Time: 15:06)

What is the problem?

100 t1 = account.balance;
200 t2 = t1 + 100;
200 account.balance = t2;

100 t1 = account.balance;
200 t2 = t1 + 100;
200 account.balance = t2;

100 + 100 + 100

- Each line corresponds to a line of assembly code
- Assume corresponding lines execute in parallel.
- The final answer is wrong. It should be 300, it is 200.

NPTEL

McGraw-Hill | Advanced Computer Architecture

117

So, let us look at this. So, I am already putting an incorrect symbol over here to draw your attention to the fact that look a mistake is going to happen. So, let us assume that the original balance was 100 and then I add 100 more it becomes 200 and then the final balance is 200. So, then here each line corresponds to a line of assembly code as you can see a load a register r and then the store.

So, let us assume that we have two threads running a same piece of code and the two lines two corresponding lines execute in parallel, a lot of problems can happen. So, the final answer will be wrong final answer should be 300 because we are adding 100 twice, but we will see that that will not happen.

So, what will happen? The problem is that if both these lines execute roughly at the same time both will read the original account balance to be 100. Both will then locally add 100, 200 and make it 200 and finally, both will store 200.

So, the point is that if this happens the final answer will be wrong, the final answer should be 300 because we are adding $100 + 100 + 100$, but what we see over here is that because there was no synchronization between these two pieces of code the final answer is 200 and that is not correct.

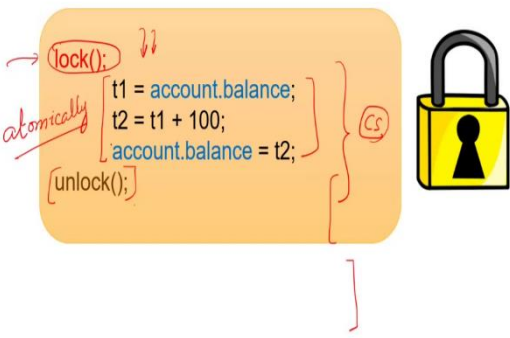
And the reason basically is that we have a race what is called a race condition that the same address we are simply reading both, both the threads are reading it 200 they are computing the new balance and writing it back and the final answer is not correct, its wrong.

And what would have avoided this well what would have avoided this entire code block would have executed before this entire code block or vice versa. So, in this case the starting balance would have been 200 we would have set this to 300 and finally, written 300.

(Refer Slide Time: 17:28)

Solution: Use Locks

Only **one thread** can execute this piece of **code** at any single point in time.



```
lock();
t1 = account.balance;
t2 = t1 + 100;
account.balance = t2;
unlock();
```

atomically

CS

NPTEL

McGraw-Hill | Advanced Computer Architecture

118

So, one solution is that we restrict this piece of code. So, we say that before we enter it we somehow lock it we will see how. And this ensures that two threads cannot be executing this piece of code at the same time only one thread can and then once it is done executing they unlock it.

So, this ensures that only one thread can execute this piece of code, then another thread, then another thread. So, this is said to execute atomically, what does atomically mean in this case? It means that either all three lines appear to execute instantaneously is the same as the definition of atomicity in the case of a right that either all three lines appear to execute instantaneously or they appear to execute not at all or they or it appears that they have not executed at all.

So, in this case, if these three lines execute atomically then we will find that first the balance will be updated from 100 to 200 and then from 200 to 300. The way this is done is that we create a critical section. So, this piece of code with a lock and unlock is a critical section also known as CS. In a critical section this lock function returns when the thread has exclusive access to this piece of code no other thread can simultaneously execute and unlock kind of relinquishes the lock.

(Refer Slide Time: 19:02)

The slide is titled "Use Atomic Instructions to Implement Locks". It contains the following text and annotations:

- For implementing lock and unlock functions.
- We need atomic instructions
- Either execute **completely** or not at all. Nobody observes a **partial state**. (Handwritten: "completely" is underlined, "partial state" is circled in blue)
- Most **atomic operations** also act as a **fence**. (Handwritten: "atomic operations" is underlined, "fence" is circled in red)

Below the list, there is a diagram illustrating an atomic exchange operation:

- A yellow box labeled "Atomic exchange operation" is shown.
- Below it, a blue box contains the instruction: "Atomically exchanges their contents".
- Handwritten annotations around the instruction box include: "Lock" (in red), "Coherence" (in red), "expensive" (in red), and "SO" (in red).
- A red arrow points from the "Atomic exchange operation" box to the "fence" annotation.

The slide footer includes the NPTEL logo, "McGraw-Hill | Advanced Computer Architecture", and the page number "119".

So, how do we use atomic instructions to implement locks that are a vital aspect of the critical section? For implementing lock and unlock functions we need atomic instructions we will see in a second what they are. So, an atomic instruction is similar to an atomic critical section in the sense either executes completely or not at all nobody observes a partial state. In addition, most atomic operations also act as a fence.

And so, that fence properties there which basically says that all instructions before it complete then the fence executes and then all instructions begin. So, let us look at one

atomic exchange operation which we will also call the synchronizing operation in a second we will see how that works and in atomic exchange operation will look like this and we will use the coherence protocol to implement it. So, we will see how?

But the way that it looks like is it takes two arguments a register and a memory. Instantaneously or let us say it appears to execute instantaneously it atomically exchanges their contents. So, registers contents go to the memory the memory's contents go to the register.

So, this is an atomic exchange instruction and almost well not almost all instruction sets that are commercially used have such atomic instructions because they are required without them we will not be able to implement locks. So, in excitation instruction set if we just add the lock prefix to an otherwise non atomic exchange it becomes an atomic exchange and in atomic instruction is expensive well why is it expensive well number one it does a write in the sense you know you are reading it and then you are writing it.

So, in that sense, but that is not that expensive it is more expensive because it is a fence. So, fence essentially prohibits out of order execution between instructions of both sides of the fence this is why these instructions are very expensive, but they will be used to implement what we call synchronization edges in our execution witness we will see in a second how?

(Refer Slide Time: 21:31)

Use Atomic Instructions to Implement Locks

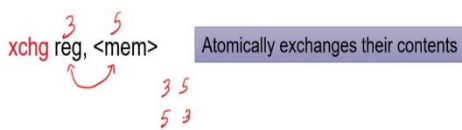
For implementing lock and unlock functions.


- We need atomic instructions
- Either execute **completely** or not at all. Nobody observes a **partial state**.
- Most **atomic operations** also act as a **fence**.

Atomic exchange operation

`xchg reg, <mem>`

Atomically exchanges their contents





McGraw-Hill | Advanced Computer Architecture

119

So, let us again look at an atomic exchange. So, in atomic exchange what it does is that it exchanges the contents of a register and memory and that too atomically. So, nobody sees a partial state in the sense that let's say this contains 3 this contains 5. So, you will either see 3 5 or you will see 5 3, but no other state.

(Refer Slide Time: 21:47)

Assembly Level Implementation of the Lock and Unlock Functions

```

lock:
    mov r1, 1
    xchg r1, 0[r0]
    cmp r1, 0
    bne .lock
    ret
        
```

Handwritten notes:
 - Red circle around `0[r0]`
 - Red arrow from `0[r0]` to `0[r0] ← 1`
 - Red arrow from `0[r0]` to `0`
 - Red arrow from `0` to `0 → 1`


- `r0` contains the lock address
- The `xchg` instruction contains a fence
- Contains 0 if the lock is not acquired
- 1 if acquired

```

.unlock:
    mov r1, 0
    xchg r1, 0[r0]
    ret
        
```

Handwritten notes:
 - Red circle around `0`
 - Red arrow from `0` to `0[r0]`

- We store 0 at the lock address



McGraw-Hill | Advanced Computer Architecture

129

Let us now look at the implementation of a lock function in assembly. So, the lock function uses an atomic exchange instruction as is visible here and. So, we assume that `xchg` in this case refers to atomic exchange.

So, what we do is that every log function needs to have an associated lock address. So, let us assume that the register `r0` contains the lock address and the furthermore exchange instruction also contains a fence. So, the exchange instruction is a synchronizing instruction. So, along with doing an atomic exchange it contains a fence as well.

And furthermore, the semantics of the lock is such that if the memory address contain 0 it means that the lock is free it is not acquired. And if the memory address contains one it means the lock has been acquired by some thread. We do not know which is thread, but some thread has acquired the lock.

So, now, the way to acquire the lock is these two pieces of code. So, we move one to register `r1` and then we exchange the contents of `r1` with the lock address. So, at the end of this atomic operation the lock address contains 1 basically, but that is not important. So,

this anyway we are doing, but what we care about is the value of $r1$ because that will contain the old value of the lock address. So, the value of $r1 = 0$ this means that this instruction essentially acquired the lock. So, it set the starters from 0 it set it to 1 which means that the lock was free and the now it set it to acquire.

So, this means that the current thread has acquired the lock. So, we compare $r1$ with 0 if it is not equal which means if $r1 = 1$ this means that the lock was acquired by some other thread prior to this exchange instruction. Hence we have not acquired the lock. So, we go over here. But if $r1 = 0$ it means that the previous value was 0 and of course, the current value is 1. So, we have acquired the lock, so it the function just returns. So, pretty much it keeps on trying to acquire a lock in a loop over and over and over again until it gets the lock.

And the unlock function what we do is we do the same thing we store 0 at the lock address again we use an atomic instruction for this purpose and the reasons will be clear very soon. So, we put 0 in $r1$ and then we do an atomic exchange. So, in this case what we do is that the contents of $r0$ come to $r1$, but that is not important. What is important is the contents of $r1$ go here and the contents are 0 this means the lock is free and after that we return.

And given the fact that this instruction also there is no need to loop and the there is the reason there is no need to loop is because this instruction will be successful at some point in time. So, that is essentially a requirement of coherence. So, we will see how it extends to such atomic instructions, but pretty much the idea is that we just need to execute it once we need to set this to 0 and move out.

In both cases we are using this atomic instruction which also contains a fence. So, this is the lock function and this is the unlock function. So, the unlock function does not have a loop, but in principle the lock function might keep on looping till infinity because it will never get the lock because another thread is taking the lock.

(Refer Slide Time: 25:46)

Implementation of Atomic Exchange

$xchg(r1, [r2])$

- temp = r1, r1 = [r2], [r2] = temp

It involves 3 steps

- 1 memory read + 1 memory write + register move
- All the operations need to happen atomically
- This is called a read-modify-write instruction (RMW)

Method

- Get exclusive access (M state) with write permissions for the memory address in r2
- Perform the read-modify-write operation
- Do not respond to any other requests from the local cache, or other caches, or the directory when the operation is in progress
- Respond to the directory or other caches only when the operation is over

Handwritten notes: $xchg$ mechanism (completion of a write), $xchg$ (fence)

NPTEL
McGraw-Hill | Advanced Computer Architecture
121

So, implementation of atomic exchange, so how did we implement atomic exchange?. So, what we do is that, so the general format is you have a register and the memory address. So, r1 needs to be put into some temporary storage location we will read the value of r2. So, r2 is a memory address. So, the value of the address who's the value stored at address r2 that we put into r1. And then the final is the last step is that we will do a store operation put temp over there.

So, this is essentially what is called a read modify write operation in the sense that we are reading this memory location there is a read and clearly we are modifying. So, in this case we are not modifying r1 is still being kept, but we maybe we can have an increment operation also where we change it. So, that would be a modified and then we have a write.

So, it involves 3 steps a memory read a memory write and a register move, all three of them need to appear to happen instantaneously or need to happen atomically and such kind of a read modify write instruction that combines a read some sort of a register movement. And then write is what we want to implement using an extension of our cache coherence protocol that is the challenge that lies in front of us.

So, the method for this is like this that we get exclusive access with write permissions for the memory address in r2. So, that is the first thing that we get exclusive access for this address. So, this is the exclusive access that we get the way we get it is that we essentially

get the line in the M state. So, basically line is set to the M state. So, we have right permissions for the memory address in r2.

So, this can be caught and by getting a right miss on the by placing a right miss on the bus. So, we get exclusive access then we perform the operation in the sense we read the register and then we read the memory location into the register. And finally, we transfer the old value of the register into the memory address. So, we perform a swap.

So, the interesting thing is that the way we guarantee atomicity. So, the way that we guarantee atomicity is that after getting exclusive access we perform these three operations quickly. So, question is how? So, first is that in an exchange operation we need to understand that there is a fence right there is a fence within it.

So, we wait for the exchange operation to reach the head of the r o b. So, we just wait for the exchange operation to reach the top and so this will guarantee that all the instructions before it have completed and gone away. And after that it is possible that some writes might have been delivered to the memory system, but they would not have reach their final location.

So, at this point we were flush the write buffers and m s h rs and so on. And get pretty much get an acknowledgement from the memory system that all the instructions which were fetched before the atomic exchange have all completed totally. And no other instruction after the fence has will be allowed to perform any read or write. So, we will pretty much stall that part of the execution. So, that is the fence part.

And after this once exchange reaches the head of the r o b that is when this is sent to the memory system. So, the memory system the first thing that it does is it gets exclusive access and till the time that the exchange operation is not completed. So, interrupts are not delivered and I mean think of a that way that interrupts are kind of kept on hold and interrupts is only being delivered to the next instruction not to this.

And there are variants of this instruction that additionally provides the status of whether this could execute successfully or not in that case interrupts are delivered. So, this instruction can be executed partially. So, then of course, the user will know that this instruction did not execute successfully.

But let us for the time being assume that this executed successfully. So, what we do is we get access to the line in the M state we quickly perform the swap and then we perform the write and we wait for the write to fully complete which means that we need to flush off the write buffers and m s h rs and all intermediate buffers.

So, pretty much the architecture has to provide some kind of a mechanism to ensure that a write has fully completed. So, some sort of a mechanism for complete completion of writes needs to be there most of the time flushing the buffers is good enough because the writes ultimately then head towards their final destination.

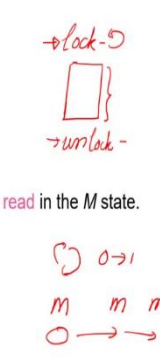
So, while this instruction is being done it is possible that we might get other requests on the local cache or other caches or the directory when the operation is in progress. So, it is possible we have read, but we have not written and then suddenly a write miss comes. So, the question is what do we do? Well we do not respond to any of these requests until the x atomic exchange is complete and only when the operation is over doing we respond to the directory or other cache.


So, for us executing an atomic instruction is kind of heavy that the overheads are large in terms of performance. And the reason that the overheads are larger like this because we have a fence that is number 1. And number 2 when we are we need to also execute this instruction very carefully in the senses after we start we simply do not respond to any message and we quickly perform the memory read write register move and then we respond to other messages. So, this ensures that this instruction cannot be interrupted in the middle.

(Refer Slide Time: 32:33)

Spin Locks

- We need to repeatedly try to **acquire** the lock
- This is a spin lock
- There are a lot of **overheads**:
 - Every time we **access** the lock, it needs to be **read** in the *M* state.
 - Too many **invalidation** messages
- Test and Exchange Lock





McGraw-Hill | Advanced Computer Architecture

122

So, now the question is that if you have not understood up till now why did we design the lock in this manner and why do we need a fence? well. So, the idea basically was that we will have a piece of code a lock. So, we keep on spinning on the lock address that is why it is also called a spin lock this process has a lot of overheads in addition to the fence? So, we will see what?

But pretty much the idea is that the reason it acts like a fence is because none of the instructions in the critical section should actually execute before the lock because if they actually do then the notion of having a lock that kind of breaks down. So, that is the reason we need to have a fence here that we acquire the lock first and then the instructions in the critical section execute they completely finish and after that an unlock corporation happens that is the reason we need one fence in the lock part and one fence in the unlock part.

(Refer Slide Time: 33:41)

Assembly Level Implementation of the Lock and Unlock Functions

Handwritten notes:
L → xchg
U → xchg
Sc = program order + atomicity

```
.lock:
    mov r1, 1
    xchg r1, 0[r0]
    cmp r1, 0
    bne .lock
    ret

.unlock:
    mov r1, 0
    xchg r1, 0[r0]
    ret
```

- r0 contains the lock address
- The xchg instruction contains a fence
- Contains 0 if the lock is not acquired
- 1 if acquired

- We store 0 at the lock address

NPTEL
McGraw-Hill | Advanced Computer Architecture
120

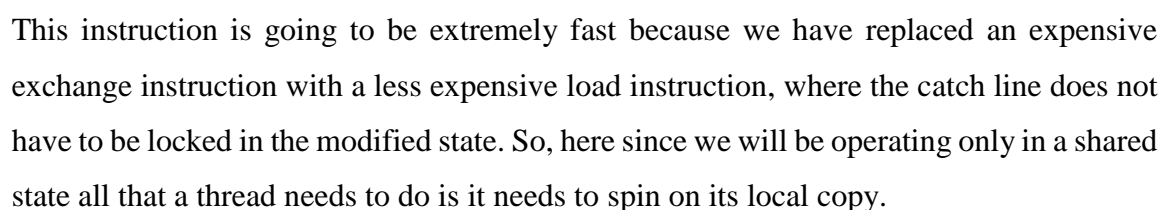
So, now if we also think about this piece of code I want to raise one more point before actually move ahead. So, if we actually look at the structure of a critical section what here is what you will observe? What you will observe is it starts with a lock then we have the code and then an unlock again a lock and then an unlock and so on.

So, the key instruction over here is atomic exchange instruction see if I just look at the program snippet of all of these atomic exchange instructions across all the threads I claim it will be in sc well why because Sc is program order plus atomicity program order is clearly being maintained because there is a fence within these instructions. So, program order is clearly being maintained.

And given that these are atomic instructions atomicity's also being maintained. So, the synchronizing instructions the synchronization instructions, in this case these are all the exchange instructions if I were to just consider them then they will be in sc and this is an important property which we will use later. So, bear this in mind keep this in mind.

So, now, coming to the lock process what we do is we do an atomic exchange we see if we set the lock from 0 to 1 if not we try again if not we try again. So, we keep trying say we needs try what we are essentially doing is that we are locking it in M state and we are trying to perform the swap.

(Refer Slide Time: 35:42)



Once it finds its local copy to actually be 0, well then it will attempt and exchange only when you think. So, the lock is free when it reads a 0 in that case what it will do is it will exchange $r1$ with the value of the lock it will compare $r1$ with 0 if it is equal it means we have gotten the lock that is great.

Otherwise, if it is not equal it will again jump back to test and again we will keep on spinning, but we will in the spin we will not do atomic exchanges we will just do a read. And when we see some hope when we see the lock to free lock to be free we will again try to attempt it.

So, there are power efficient versions of this in the sense we can add a random delay or we can add what is called exponential back off. And initially we wait for some time which is let us say t units or a maximum of t units between 0 and t units then another if let us say we are not successful we again wait for some time which is which is between 0 and $2t$ units again we wait for the random duration which is between 0 and $4t$ units and so on.

So, ultimately this works and with extremely high probability we actually acquired the lock and given the fact that we acquire the lock well what's left then we return and we execute the critical section. Unlocking is simple, so this is the same as the basic exchange lock.

So, we will call this the test and exchange lock as compared to the basic version. So, in this case, we just exchange the contents of $r1$ and since $r1$ contains 0; 0 goes over here. So, the lock becomes defect or free and then we return. So, this is the basic structure of often optimized version of the lock acquire algorithm where instead of the exchange function we use the test and exchange function.

(Refer Slide Time: 39:12)

Test and Exchange Lock

```

.lock:
    mov r1, 1

.test
    /* test if the lock is free */
    ld r2, 0[r0]
    cmp r2, 0
    bne .test

    /* attempt an exchange only when the
    lock is free */
    xchg r1, 0[r0]
    cmp r1, 0
    bne .test
    ret
        
```

```

.unlock:
    mov r1, 0
    xchg r1, 0[r0]
    ret
        
```

McGraw-Hill | Advanced Computer Architecture

123

Where the spin loop is on testing and in the spin loop we have a much cheaper load instruction as oppose to the expensive exchange instruction which had a fence. It had a fence it had a write it at all of that stuff, but in this case because we have a load it is possible for multiple caches to have the lock in the shared state. So, we only spin on the local copy or in the local cache.

(Refer Slide Time: 39:30)

| Atomic Operation | Example | Explanation |
|--|---|--|
| Test and Set | tas r1, 8[r0] | if (8[r0] == 0) { 8[r0] = 1; r1 = 1; <i>Success</i> } else r1 = 0; <i>Failure</i> |
| Fetch and Increment | fai r1, 8[r0] | r1 = 8[r0]; 8[r0] = r1 + 1; |
| Fetch and Add | faa r1, r2, 8[r0] | r1 = 8[r0]; 8[r0] = r1 + r2; |
| Compare and Set | cas r1, r2, 8[r0] | if (8[r0] == r3) { 8[r0] = r2; r1 = 1; <i>Success</i> } else r1 = 0; <i>Failure</i> |
| Load linked (ll) Store conditional (sc) | ll r1, 8[r0] mov r2, 1 sc r3, r2, 8[r0] | r1 = 8[r0] /* Use the ll instruction */ /* sc */ if (8[r0] is not written to since the last ll){ 8[r0] = r2; r3 = 1; } else r3 = 0; |

McGraw-Hill | Advanced Computer Architecture

124

So, it turns out that there are a large number of additional atomic operations not just an atomic exchange. So, one of the most common ones is a test and set. So, here let us say we

have $r1 = 8r0$, let $8r0$ be the address of the lock (Refer Time: 39:51) generalized it. So, this is the pseudo code of it and entire pseudo code executes atomically.

So, here if the lock address = 0 we set it to 1, we set the value of $r1 = 1$ which indicate success, else we set the value of $r1 = 0$ which indicates failure. Then we have fetch and increment since fetch and increment what we do is we again take a register as a memory address. The contents of the memory address are transferred to the register and we increment the contents of the memory address. So, $8r0 = r1 + 1$ which is previous value +1 and this happens atomically. So, we can extend this to fetch and add there you have two registers in the memory address assuming the isas supports this.

So, in this case, we read the old value of the memory address and the new value of the memory address is $r1$ which is old value plus an offset $r2$. Now we will discuss two more synchronization operations which are regarded as extremely powerful what makes an synchronization operation powerful I will discuss that a few slides later. So, the first one is compare and set.

So, in compare and set we compare the memory address $8r0$ with the value of a register $r3$, if it is equal we set the value of $8r0 = r2$ and we set $r1$ which is the flag saying success. Otherwise, if you are not able to do this we set $r1 = 0$ which indicate failure.

So, what do we do? what we do is that we read the memory address and we compare it with the value if it matches then we set it to the value of this register and we set $r1 = 1$ indicating success otherwise we set $r1 = 0$ which indicates failure then let us look at another pair of synchronization instruction.

This is a pair. So, unlike the other one it is a pair it is always meant to be used in pairs is called load linked in stored conditional. So, the first load link looks like a regular load, but it is just a synchronizing load. So, $r1 = 8r0$ means the contents of the address are sent to register $r1$.

And so, you will actually look at a cache instruction I will tell you the quick difference if you have seen it up till now. The cache instruction actually requires 4 arguments to 3 registers and one memory most i s a s will not be able to support it unless we do a little bit of trickery here and there, but the ll and sc pair is nicely supportable by most risk instruction sets.

So, if you see the first instruction it is a regular node, but its slightly special in the sense it is a synchronizing load and it has it has its it is also a fence and then what we do is that we call sc and in sc what we do is we store 1 in r2 and we try to store one in the lock address.

So, here is what we are actually trying to do. So, the connotation of lock address is slightly different over here. So, if 8 r0 is not written to since the last load linked. So, here it does not really matter what is the value of the lock address what we really care about is that when we do store conditional since the last time that we did a load linked has there has there been any writes to this address if there have been no rights to this address we set 8 $r0 = r2$. And we set the status = 1 otherwise we set the status equal to 0.

So, that is the broad idea over here that what we do is that when we acquire a lock over here we basically execute this instruction in pairs first we do a load and then we do a store conditional and let us see in between them if no other thread has accessed the lock address then the lock address is set to 1. I mean in any addresses set to 1, but it can be another value it can be set to 0 also it does not matter and, but the most important thing is we set the status to 1 we set $r3 = 1$ and if let us say that there has been an intervening we set $r3 = 0$.

So, what this basically does is on the time line it just indicates that between the l l and the sc this is an l l l l and the sc no other thread has written to the lock address. So, this also can be used as a synchronizing operation. So, this definitely can be used to achieve something similar to what compare and set does. So, you will find details in the book.

(Refer Slide Time: 45:36)

Lock-free Algorithms

- Let us write the same code without **locks**.
- If the thread goes to **sleep** after acquiring the lock, all the *Other* threads **wait**.

```

lock while (1) {
    t1 = account.balance;
    t2 = t1 + 100;
    if (CAS (account.balance, t1, t2))
        break;
}
unlock
  
```

Request → update acc. balance.
LL/SC
atomic
account balance

⚠ It is possible for a thread to starve – never get the lock.

NPTEL | McGraw-Hill | Advanced Computer Architecture | 125

So, now given all of these different synchronizing operations what else can we do with them. So, what we can do is that we can write a set of lock free algorithms that do not actually require locks. So, what is the problem with locks if the thread goes to sleep after acquiring the lock all the other threads wait, it is a thread, all the other threads wait they there is nothing that they can do. In this case; however, you take a look at this piece of code I am trying to update the account balance and I am not using any locks.

(Refer Slide Time: 46:16)

| Atomic Operation | Example | Explanation |
|--|---|---|
| Test and Set | tas r1, 8[r0] | if (8[r0] == 0) { 8[r0] = 1; r1 = 1; } else r1 = 0; |
| Fetch and Increment | fai r1, 8[r0] | r1 = 8[r0]; 8[r0] = r1 + 1; |
| Fetch and Add | faa r1, r2, 8[r0] | r1 = 8[r0]; 8[r0] = r1 + r2; |
| Compare and Set | cas r1, r2, 8[r0] | if (8[r0] == r3) { 8[r0] = r2; r1 = 1; } else r1 = 0; |
| Load linked (ll) Store conditional (sc) | ll r1, 8[r0] mov r2, 1 sc r3, r2, 8[r0] | r1 = 8[r0] /* Use the ll instruction */ /* sc */ if (8[r0] is not written to since the last ll){ 8[r0] = r2; r3 = 1; } else r3 = 0; |

NPTEL | McGraw-Hill | Advanced Computer Architecture | 124

Instead what I am using these atomic operations and all of these atomic operations all of them can be implemented with basic changes to the coherence protocol which lets say let us consider last example. We load this address in the M state and then we simply wait till the subsequent store and in the middle if anybody else has invalidated it then we say that this address has been written to otherwise we see it has not been written to that is all that we need to bother about.

Now, coming to this I will use the CAS the compare and set primitive to implement this. So, first is that we have an infinite loop, in the infinite loop first we read the account balance which was the same earlier also then we set $t2 = t1 + 100$ and then what we do is we do a compare and set.

So, in the compare and set essentially what we check is if the account balance is still $= t1$. So, the account balance is still $= t1$ then we set the account balance $= t2$. So, you can run two thread simultaneously.

So, what will happen is that if both do the CAS together one of them will succeed and one of them will fail, why is that? Well, the reason is that one of them will atomically change the account balance from 100 to 200 and when the other thread performs the CAS its account balance will not match what it read. So, the so that one will fail.

So, in this case what we are essentially doing is that we have replaced a code with a lock and an unlock which had high overheads lock and unlock as saw it you need. So, many functions and further more you need. So, many calls for just getting the lock and then doing the job.

So, we have kind of folded all of that into the simple piece of code that does not require locks all that we need to do is we need to repeatedly execute these CAS instructions until we are successful. When we will be not being successful? Well when there are concurrent requests to update the account or balance variable that is when we will not be successful.

Otherwise, this also works similar to an LL sc similar to a load linked and store conditional something that we saw over here. So, it is a similar philosophy. So, we read the account balance once we update it and before writing we check whether the account balance is the same as the old value is something that we read, if it is equal we update it.

And this entire CAS operation happens atomically which is the magic of atomic instructions. So, as you can see that the CAS might fail for a few threads because some thread would have had a successful CAS and it updated the account balance. So, threads will continue to iterate until the compare and set operation is successful, but once it is they will get out.

So, we still if we let us say consider one of the consider the best case we do not have to execute all the extra instructions of the lock and unlock code, we directly execute this and we are done. Of course, it is possible for a thread to starve when the sense it never gets the lock that is because the status of the CAS operation for it never returns as successful.

Then of course, there is a problem, but in practice this actually does not happen. So, very rarely it does happen in practice. So, it is not really something that most people should worry about and lock free algorithms consequently are very popular.

(Refer Slide Time: 50:16)

How do we eliminate starvation?

Answer: Wait free algorithms ←

Basic Idea

- A request, T , first finds another request, R , that is waiting for a long time
- T decides to help R
- This strategy ensures that no request is left behind
- Also known as an altruistic algorithm

wait-free

NPTEL

McGraw-Hill | Advanced Computer Architecture

128

Now, but let us say that we still want to eliminate starvation. So, we do not want to entertain the possibility that one thread continuously tries to tries to update its account balance. But it is not able to do because some other thread is getting in and updating the account balance in parallel the CAS is failing. So, we want to totally eliminate starvation we go for wait free algorithms.

So, in this wait free algorithms what happens is that one request T first finds another request r that is waiting for a long time, so each one of them. So, let us say a request in this case would be a request to update the account balance.

And so what happens is that we can have an array of requests where we store all the pending request. So, once a request T comes it tries to find another request in this array that has been waiting for a long time. So, then R is which is close to starvation because it is somehow not able to execute all of its steps correctly. So, in that case T decides to help r in the sense it tries to do the job that R was trying to do. So, T decides to help R by doing it.

So, then what will ultimately happen is that let's say R becomes the oldest message in the queue all other threads will ultimately try to help R. So, they will one of them will have to be successful because now all the threads are trying to do the same thing and either R will finish its job or some thread will finish the job of R on its behalf.

Regardless of who finishes the job this strategy ensures that no request is left behind and such algorithms where the request of one thread actually first reaches out and helps the requests of other threads is known as an altruistic algorithm. When you do something for somebody else this ensures that our system is overall wait free.

Which basically means that, if I have a set of instructions, if I have a critical section where I want to update something. And so let us say I am trying to acquire a lock or something and say it does not matter any kind of a concurrent algorithm I am trying to update a lock or get implement shared stack or q or whatever, but essentially I am trying to do some parallel operation. So, the lock free algorithm does not give us guaranteed completion as you can see there is an infinite loop.

(Refer Slide Time: 53:20)

How do we eliminate starvation?

Answer:
Wait free algorithms

Basic Idea

- A request, T, first finds another request, R, that is waiting for a long time
- T decides to **help** R *no starvation*
- This strategy ensures that no request is left behind
- Also known as an **altruistic algorithm**

finite

NPTEL

McGraw-Hill | Advanced Computer Architecture

126

But in comparison the wait free algorithm will actually give us guaranteed completion because ultimately all the requests will help R. So, basically there is no starvation in wait free algorithm because one request does help another request and along with no starvation the other nice thing is that requests are guaranteed to complete in finite time if not bounded time.

(Refer Slide Time: 53:49)

Summary of Consensus Numbers

? Are all atomic operations equally powerful?

Consensus number

Consensus problem: each thread proposes a value – one among the **proposed** values is chosen. The consensus number is the **maximum** number of threads that can solve this problem using a **wait-free** algorithm.

| Type of Operation | Consensus Number |
|-----------------------|------------------|
| Atomic exchange | 2 |
| Test and Set | 2 |
| Fetch and add | 2 |
| CAS (Compare and Set) | ∞ |
| LL/SC | ∞ |

lock
unlock
parallel queue

most powerful

! The consensus problem forms the theoretical basis of most concurrent algorithms.

NPTEL

McGraw-Hill | Advanced Computer Architecture

127

So, let us now given that we have seen we have a very very basic idea of lock p and wait free algorithms let us ask a question are all atomic operations equally powerful. So, what does this actually mean what is or what is power? So, let us define the consensus problem.

So, the consensus problem is that each thread proposes a value. So, let us say we have 10 threads in the system. So, thread 1 proposes 3, thread 2 proposes 5, and thread 10 proposes 7 and so on. So, then we run a parallel algorithm similar to the algorithms that I have shown in a similar to this algorithm.

So, this is a parallel algorithm if you think about it they are all trying to do a CAS. So, then in parallel we choose one among the proposed values and everybody agrees. So, let us say that we choose 5. So, that is one among the proposed values and everybody agrees.

So, why is the consensus problem important? because it is a theoretical basis of most concurrent algorithms including the algorithm for example, for deciding that which thread should get a lock. So, assume that we have a single lock for a critical section and five threads are interested in acquiring it ultimately one thread will acquire it and it will go and execute the critical section.

So, this is an example of a consensus problem because ultimately all the threads are agreeing on one thread which is kind of chosen as the winner that goes ahead and executes a critical section and it unlocks and it gets out after that one among the remaining threads will again kind of be chosen as the winner in the sense it will get the lock. It will execute the critical section and get out.

So, the fact that we are choosing one winner and that winner is going to get the lock and furthermore, the winner is being agreed to by all the threads this is an example of a consensus. So, the consensus problem kind of forms the basis of most parallel and concurrent algorithms. Particularly, algorithms that implement these data structures like this parallel lock or something like a parallel queue or a parallel stack or something like that.

So, in this case, the consensus problem is important what is the problem again we have multiple threads all the threads each thread proposes the value one among the proposed values is chosen and accepted by all. What is the consensus number? It is the maximum

number of threads that can solve this problem using a wait free algorithm. So, the consensus number is actually for a certain type of operation.

So, the consensus number let us say for atomic exchange would be the maximum number of threads that can solve the consensus problem using a wait free algorithm. Which means that we are guaranteed to attain consensus in the finite amount of time using only basic register read writes in the atomic exchange primitive.

So, this clearly gives us an idea of the power of an atomic operation the power of a concurrent operation in the sense that it can solve consensus for how many threads. So, there is a proof I am not getting into it, but you can read Herlihy and Shavit's book that the atomic power of atomic exchange test and set fetch and add fetch an increment and. So, on even atomic I mean all of this entire family of instructions exchanged test and set fetch and add fetch and increment and so on. Their consensus number is 2.

So, beyond 2 threads they will actually not be able to solve the consensus problem using the constraints that we have described which is that it should be a wait free algorithm and we will only use atomic operations of this type. So, this basically constrains the kind of parallel code that we can write and also for let us say acquiring a lock without starvation for example, which will also be a consensus problem more than two threads they will actually not be able to manage.

So, that is why every architecture needs to implement either the compare and set primitive compare and set atomic operation or the LL SC the load link store conditional atomic operation both of these have a consensus number of infinity which means that for an infinite number of threads they can solve the consensus problem which is an agreement problem they can solve it.

So, these are by all means the most powerful. So, I should add the word most powerful and most architectures would implement some form of these atomic operations. So, they would also implement either one of compare and set or LL sc because it is very important that we have an infinite consensus number.

(Refer Slide Time: 59:15)

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

NPTEL

McGraw-Hill | Advanced Computer Architecture

128

So, we have discussed a lot of coherence. So, we have spent two full lectures discussing coherence. So, what we will now do is we will kind of wrap up this discussion on memory models orders coherence and so on.

So, we will in the next lecture we will discuss a little bit about memory models you already know a lot we kind of summarize our knowledge and we will discuss about data races which is kind of the cap on the bottle. So, this has all the results that we need to write practical programs on extremely relaxed memory model machines.