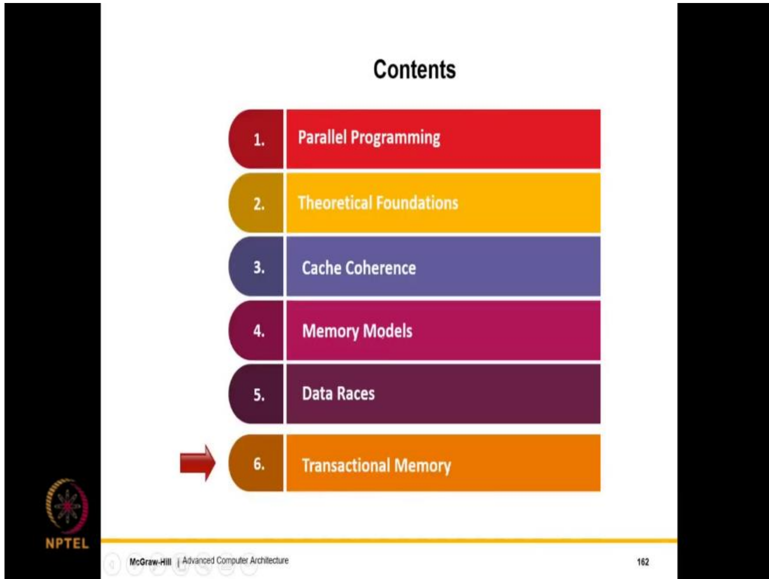
Advanced Computer Architecture
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture - 33
Multicore Systems Part - IX

(Refer Slide Time: 00:17)

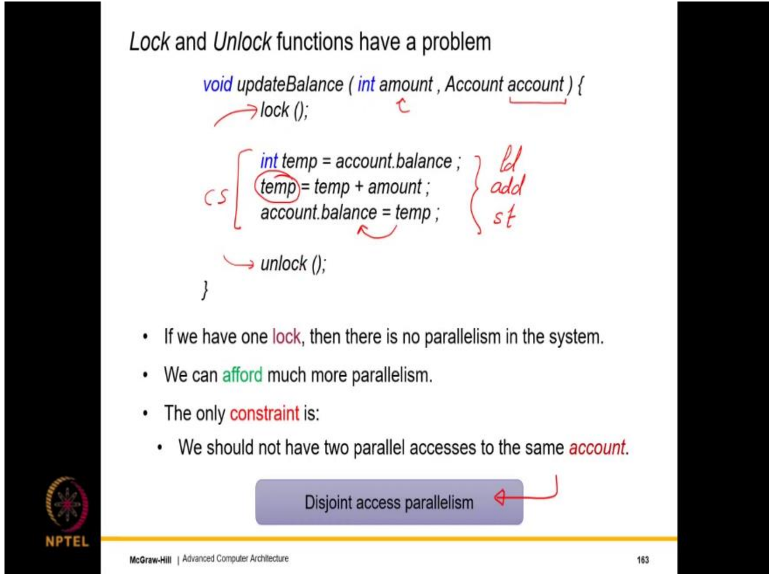
(Refer Slide Time: 00:23)

(Refer Slide Time: 00:39)



Welcome to the 9th lecture in this chapter, in this lecture we will discuss transactional memory.

(Refer Slide Time: 00:45)



So, lock and unlock functions the way that we have described have a problem. So, let us first understand what we have described and what is the problem? So, let us take a look at this function update balance. So, what we do is if this is the bank account of class account and we want to let us say add a certain amount which is amount. So, then we acquire a lock and let us assume that we have a single lock. So, we just acquire a lock and then we

read in the account balance. So, the code has been written in such a manner that one line of C code translates to one line of assembly code.

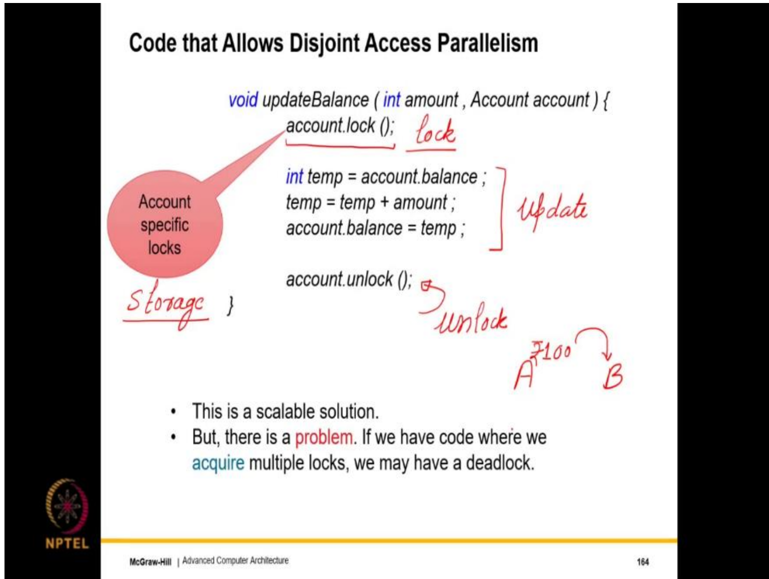
So, what we do is we first read in the balance which is the memory read operation. Then we add the amount to it and finally, the updated amount is return back to the account balance. So, this requires a load instruction, this requires an add instruction and the last statement requires a store instruction. And finally, we relinquish the lock we get rid of the lock.

So, we do it by an unlock call. So, these are the two calls lock here and unlock here we enter the critical section which in this we update the account. See if you have one lock then there is no parallelism in the system because let us say that this is for a huge bank and there are 10000 customers. So, then they cannot operate their accounts in parallel even though their accounts may not have any dependencies between them, but still the accounts cannot be updated in parallel and that is an issue.

So, we need to have much more parallelism the only constraint is that we should not have two parallel accesses to the same account. Or, let us say that in a more complex scenario if two accounts are linked then it may be necessary to lock all the accounts before a single transaction is allowed.

But most often this is not the case that is the reason when two accounts are disjoint which means that there is no relationship between them. We should be able to update the accounts in parallel this is known as disjoint access parallelism. So, disjoint access parallelism will give us the parallelism that we seek in this case it is just that with a single lock and unlock function we are not getting it.

(Refer Slide Time: 03:23)



So, now let me show a piece of code which is far more scalable in the sense that disjoint access parallelism is allowed. This is the same code, but instead of having one global lock address we have a lock address associated with each account. So, we lock that then we proceed to do the account update and then we unlock. So, we go lock we proceed with the account update and then we unlock.

And furthermore, we have account specific lock, so this does increase the memory foot print. So, the storage area of the program the memory foot print increases the reason being that previously we had 1 lock and now if we have 10000 accounts we have 10000 locks.

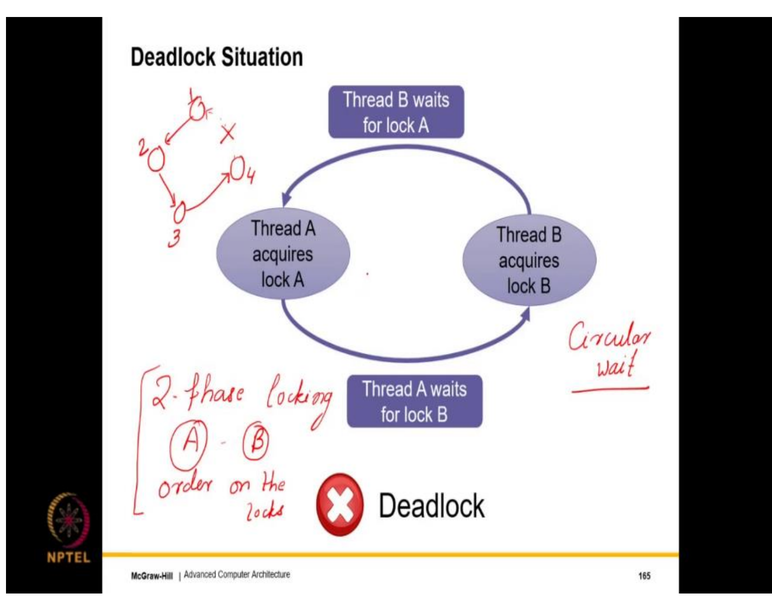
Nonetheless, this is a very scalable solution in fact, this is something that is used, but again there are problems. So, the problem is that assume that there are linked accounts. So, let us say that in different backing systems we have these joint accounts and sometimes it is necessary to check one account before actually updating it.

So, I will give a simple example let us say that there are two accounts A and B and we want to withdraw some amount from here, let us say we want to withdraw 100 rupees from here and we want to transfer it over here. So, one thing we need to check is that whether the first account has 100 rupees or not in its balance and second we need to check if account is frozen or not.

So, many times what happens is because of different law and order reasons account B may be frozen if it is frozen we cannot transfer any money to it. So, what we would need to do is we would need to lock both accounts A and B and this is where we can have a potential deadlock situation.

So, the key idea is that whenever we are dealing with multiple locks we may be looking at a potential deadlock situation. In such a deadlock situation and such deadlock situations are difficult to handle, but I will provide a solution. So, the deadlock is mentioned is shown in the next slide.

(Refer Slide Time: 05:51)



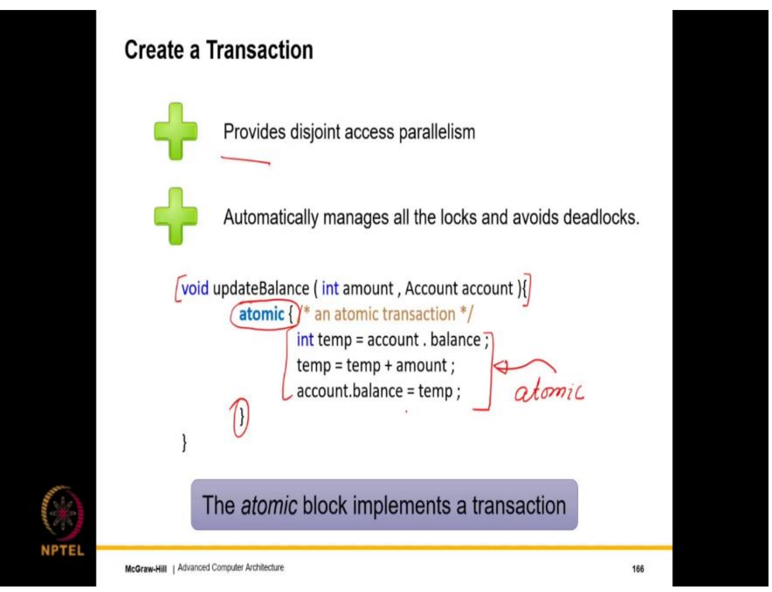
So, let us assume that there are two threads A and B. So, then thread A will acquire lock A and thread A will wait for thread B to relinquish lock B. And thread B will acquire lock B it will wait on thread A to relinquish lock A. So, as you can see we have a circular wait here with no thread making any progress and then what will happen is that we have a deadlock situation.

So, one easy way of actually solving this is known as two phase locking. So, what we do is that we impose an order on the locks. So, we say that look you will first acquire let us say lock 1 and then lock 2 or first you will acquire lock A and then lock B. So, then what will happen is that the moment you impose an order on the locks then the deadlock is not possible. Because if you see a cyclic wait is not possible the reason being that we can always draw a kind of dependencies.

That I am holding on to let us say one lock 1 and I am interested in lock 2, another thread is holding on to lock 2 it is interested in lock 3. So, let us assume that the lock numbers also correspond to the order in which they are locked. So, you will finally, see that if I try to acquire the locks in a certain order which in this case is 1, 2, 3, 4; 4 can never wait on lock 1 which basically means that after acquiring lock 4 I will never wait to acquire lock 1.

The reason being that I am acquiring locks in order even the fact that a circular wait is not possible the deadlock is not possible, otherwise it is. So, of course, two phase locking can be used to solve this issue, but here again the assumption is that before a program starts to execute I am aware of all the locks that it will acquire which in many cases many times I am not.

(Refer Slide Time: 07:55)



So, this is where the modern paradigm of transactional memory comes into play. So, it is not really that modern it was introduced roughly in the 2010-time frame, after that there has been a lot of work. But fortunately, now we have a lot of sophisticated hardware and software solutions that implement transactional memory.

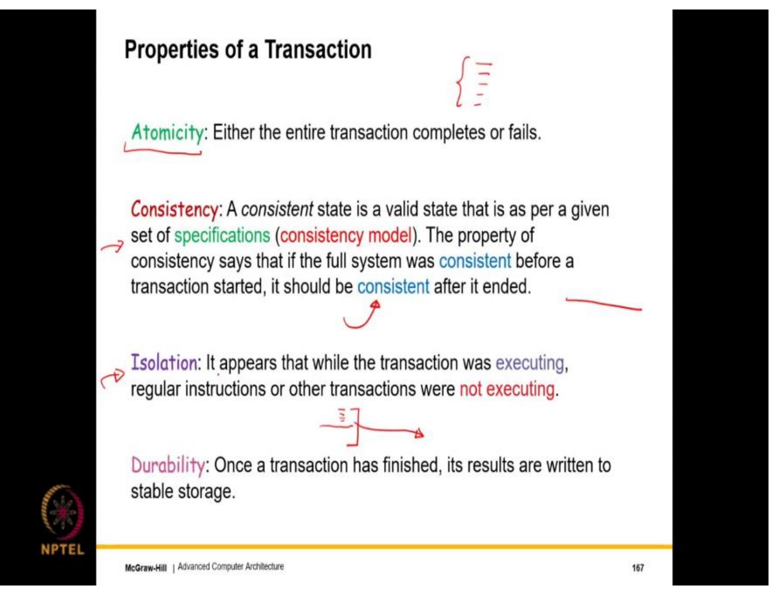
So, the two great plus points of transactional memory a are that they provide disjoint access parallelism. They automatically manage the issue of locks and deadlocks and everything. So, this entire lock and synchronization business is managed by transactional memory. So,

if I were to implement the same code using transactional memory. So, the signature of the function would remain the same, the basic code would remain the same.

But here is where the magic happens I will just enclose the critical section within an atomic block and the transactional memory subsystem would magically, automatically and magically auto magically do the rest. So, what would it do? It would automatically ensure that this piece of code executes in an atomic fashion further more whatever locks need to be acquired are acquired, the programmer need not bother and there will be no deadlocks.

So, all of this is being achieved by just wrapping this piece of code within an atomic lock as you can see starting with a curly brackets here, ending with the curly brackets. So, the atomic lock implements a transaction. So, in this lecture we will talk about different kinds of transactional memory memories they can be implemented either in software or in hardware.

(Refer Slide Time: 09:49)



Let us now look at the properties of a transaction. So, these are the same properties that are used in the world of data basis and bit they have been slightly modified to suit our requirements of transactional memory. So, these are the famous asset properties which are extremely commonly used, they are extremely popular in the world of data basis. So, we have looked at atomicity earlier also.

So, in this case of course, in the context of transactions it varies slightly. Say the context of transactions it basically means that either the entire transaction completes or fails. So, which means either it completes or aborts. So, failing a transaction basically means abort which means that the rest of the world either perceives to the set of instructions have either executed completely or they have not started at all.

So, it is kind of the same thing as what atomicity is in the case of reads and writes. So, basically we can turn off merge both by saying that the entire transaction appears to complete at some instance of time. So, at some instant of time it appears to complete instantaneously or it appears to have not started at all consistency.

So, this is slightly different from a memory consistence a memory consistency model well let us go through it. So, a consistent state as we have described before is a valid state that is as per a given set of specifications. Say this is known as the consistency model where we say that a consistent state is a state which follows a certain set of specifications whatever it may be it was the same case in memory consistency as well.

The property of consistency in this case says that in the full system was consistent before a transaction started it will also remain consistent after it ended, which basically means that let us say either configuration of the entire system satisfied some property before a certain transaction started. When the transaction ends it will still continue to satisfy the same property.

So, that is what consistency means in this case, where as in the case of memory consistency meant that at all points of time the state is as per specifications. But in this case it says that look the state is as per specifications no doubt I if we start at the beginning of a transaction it needs to be consistent and at the end also it needs to appear to be consistent.

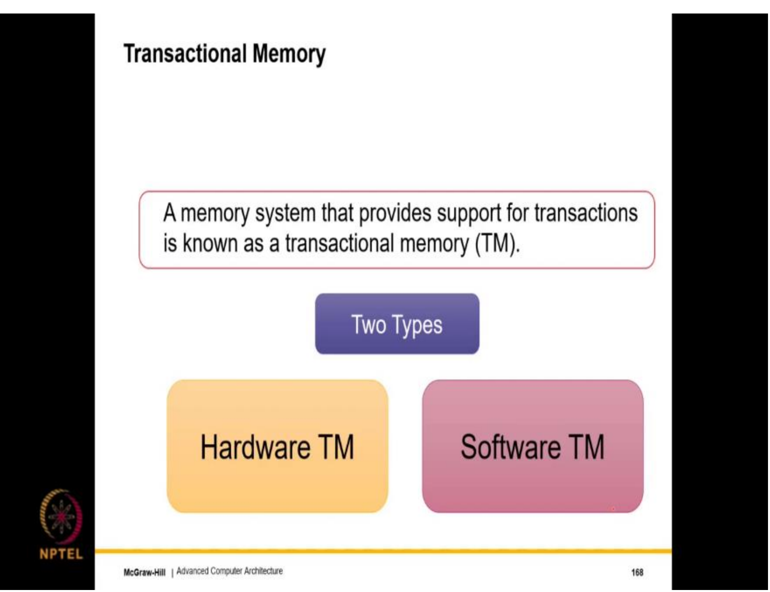
Isolation this basically means that why the transaction is executing, it appears that regular instructions and other transactions are not executive. Which means that, if a transaction is executing it appears on the transaction is executing isolated. So, it is not at all its execution is not at all being affected by other instructions or other transactions and it is also not affecting other instructions and other transactions.

So, one direct side effect of isolation is that let us say no other transaction or memory access can see an intermediate write of a transaction. So, let us say in the transaction is

continuing and in the middle you write something, but you have still not committed the transaction what does committing mean, committing means you have not finished it. So, we will discuss more of this, but the key idea is that the group of instructions the entire group as a whole has not finished.

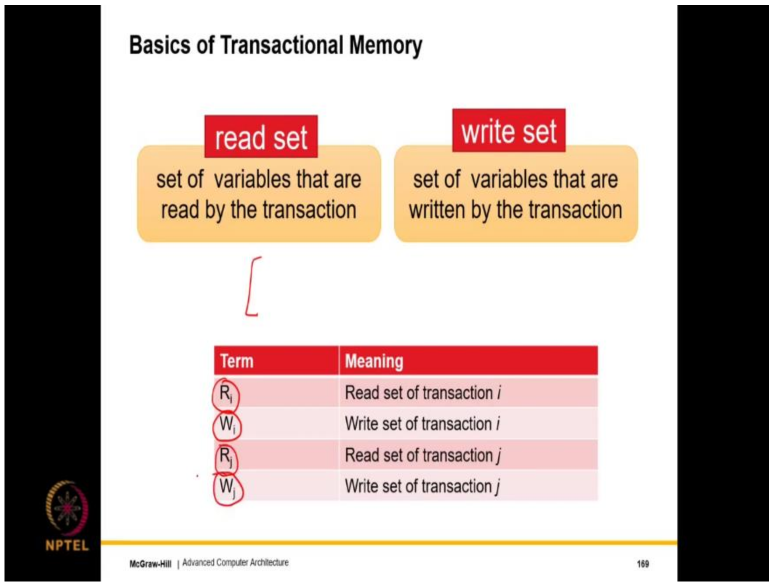
So, when it finishes successfully you say that it has committed. So, isolation basically means that in the middle of a transaction nobody else can see any intermediate values. And finally, we come to durability, durability says once a transaction has finished each results are returned to stable storage. This basically means that, once the transaction has finished its results cannot be erased they are written into a stable storage and they remain there.

(Refer Slide Time: 14:07)



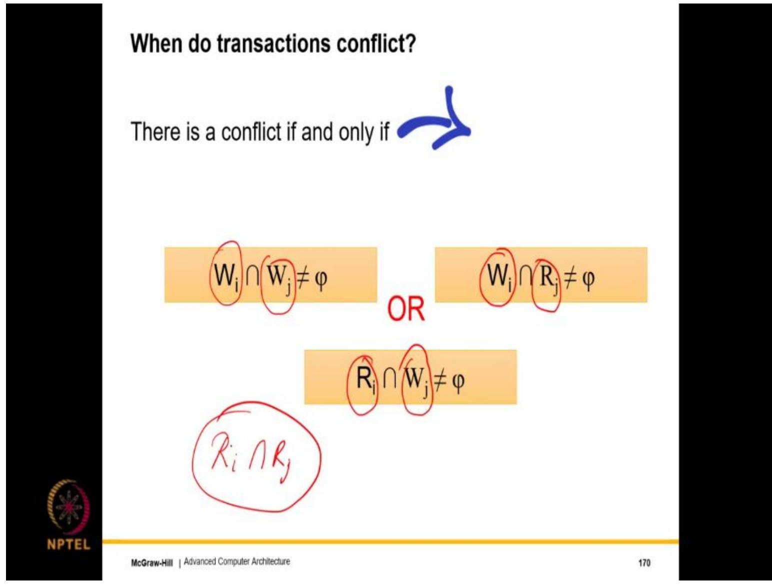
So, a memory system that provides support for transactions is known as the transactional memory. And there are two types of transactional memory hardware transaction memory and software transactional memory as we have discussed. So, hardware transactional memory provides transactional support in hardware and a software transactional memory provides transactional support in software.

(Refer Slide Time: 14:33)



So, let us now look at the basics of transactional memory. So, we have a read set and a write set for a transaction or a set of transactional statements. The read set basically says that the set of variables that are read by the transaction that is the read set. Similarly, the write set is the set of variables that are written by the transaction that is the write set. So, now let us introduce some terminology. So, let the term R i mean the read set of transaction i W i mean the write set of transaction i. Likewise, is the case for transaction j which is the read set and write sets are R j and W j respected.

(Refer Slide Time: 15:21)



When do two transactions conflict? so, go back to our definition of disjoint access parallelism and then try to come back. So, they conflict only when there is an overlap in their memory foot prints, if I were to formalize that here is what it would mean. Either the write sets overlap in the sense the write to the same variable. If the write to the same variable intersection of the write sets will be non null then of course, there is a conflict.

So, there is a conflict no doubt between them because it is like a write after write or a hazard. If there is a conflict between the read and write sets which can happen in two ways either W i intersection R j is non null or R i intersection W j is non null.

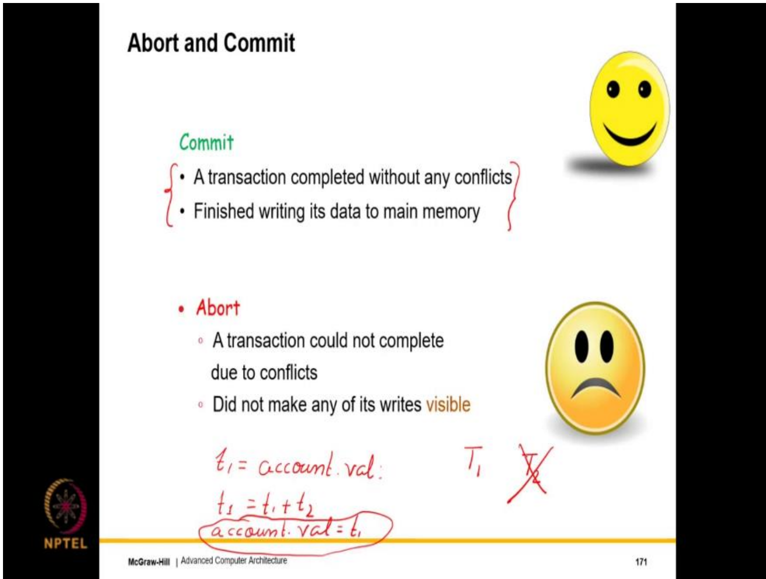
In this case what will happen is that the read and write sets will have a non null intersection. And because they will have a non null intersection there is a conflict mainly because what will happen is that we can have a write after read or a read after write problem.

So, this characterization is very similar to data races. So, in data races we had said that two accesses are conflicting if at least one of them is the write, which means that there has to be an ordering between them. So, we can extend the same idea to transactions where instead of single accesses we will be looking at sets.

So, we will be looking at read sets and write sets as you can see over here that the read sets and write sets should not have an intersection. Of course, we are not considering one case which is basically the intersection of the read sets of both the transactions. There is no problem as such if they intersect the reason is that even in the case of data races we did not have an issue.

Because we have let us say if two transactions interchange set of variables that their ordering does not matter there is no reason why if the reads have to have an ordering may happens before ordering between them. So, we the notion of conflicts between transactions directly extends what we have studied from data races.

(Refer Slide Time: 17:45)



Now, I will explain two more basic ideas abort and commit they were referred to in an earlier discussion I will be not defined formally. A commit is when a transaction completes without any conflicts means conflicting accesses as described in the previous slide. Which means, it nicely completes and there are no write overlap problems or read write over lap problems in other transactions.

So, then we say that this commits other transactions with other transactions which will ultimately commit. So, if there are no problems of this nature then this happily commits, if there is a problem which means a transaction could not complete due to conflicts. So, what can happen?

Well, what can happen is that if let us say you consider the same bank account example where what we basically do is we first read in the value of the accounts into a temporary. We augment the temporary with the amount that is passed let us say that is in t 2 and then we set the accounts value to t 1. If two transactions two copies of this are running in account is the same.

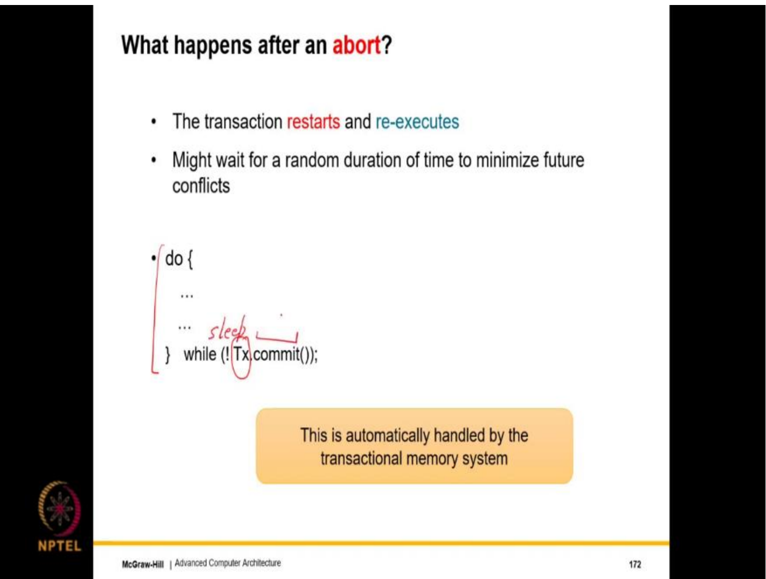
Basically what can happen is that you will have two updates two account .val and there will be no total ordering between them. So, this will be a data race no doubt and also in terms of transactions there will be a conflict because the write sides will overlap. So, both the transactions clearly cannot commit in the sense they cannot write their results to main memory or to the architectural state.

So, one of them has to kind of die and one of them has to be aborted. Aborted basically means that none of its writes or non of its changes will be visible it will look as if this is the other transaction the conflicting transaction of course, there are two transactions t 1 and t 2 that are conflicting. Then it is hard to say which one is conflicting, but let us say both are mutually conflicting.

One of these has to be kind of scrapped, erased, aborted. So, if t 2 is aborted it will appear as if t 1 was the only transaction which was executing in the isolation and t 2 never began, never executed and of course, never completed. So, there will be no record of t 2 or no record of any changes made by instructions within t 2.

So, that is what abort means, commit means successful completion abort means it will appear to the rest of the world that you never started and you never treat anything. Even though, you actually might have done something with like to a conflict, but all of that all of that state will get erased.

(Refer Slide Time: 20:35)

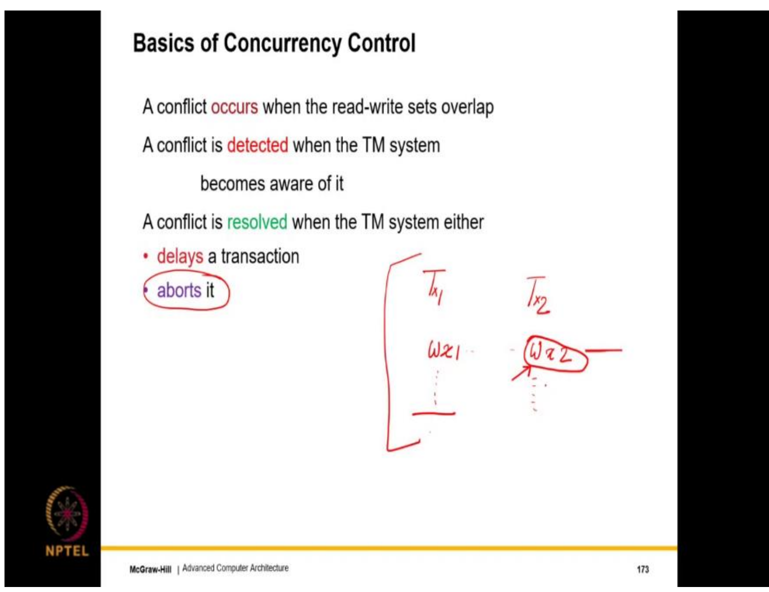


What happens after an abort? When the transaction needs to restart and re execute is possible that it may abort again. So, one good idea is that you might want to wait for a random duration of time to minimize future conflicts and then retry. So, this is automatically handled by the transactional memory system.

So, what we can have is a do while loop where we will continue to read write transactions until the transaction commits, so Tx dot commits. So, typically we use Tx to represent a transaction in the transactional memory literature. So, until the transaction commits we continue to loop.

So, we just continue to just loop and loop and until it commits. So, of course, to reduce the probabilities of a conflict we can do a random back off. Random back off means that we deliberately insert a sleep statement let us say over here where we just wait for a random amount of time and then again read, write.

(Refer Slide Time: 21:43)



Now, let us look at the basics of concurrency control. Concurrency control basically refers to the subsystem that manages these transactions and takes care of this is the detection takes care of conflicts if there is a need. Say conflict occurs when the read write sets overlap this we have seen a conflict needs to be detected by the TM system and a conflict is resolved when a TM system either it can do two things.

We have seen the second option which is aborting a transaction the other one could be that if let us say there is a write conflict. I am writing to some address let us say x and then T 2 will be writing to some other it is the same address x, but the values are different.

So, what we can do is that if there is a conflict over here we can let us say deliberately stall T 2 and delay this instruction over here and just allow T 1 to complete. After T 1 completes

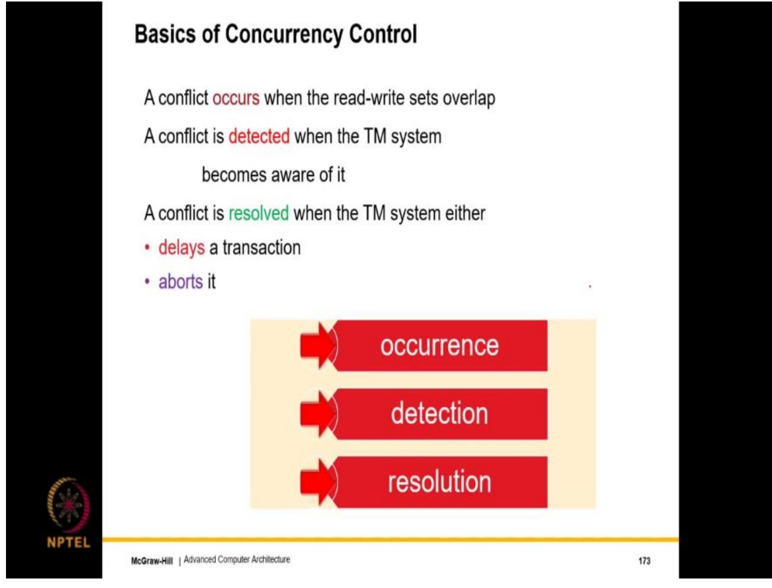
we can restart transaction T 2 from this point, I should rather add T x 1 and T x 2. So, I should rather use this terminology. So, what we can do is we can stall transaction two T x 2 at this point not allow the write to happen. So, essentially stall the write.

Once T x 1 completes we can release the write and then we can continue the transaction, in this case at least there will be no conflicts because of this write. So, even delaying a transaction to avoid conflicts which means to avoid deliberately avoid overlaps in the read write sets is also a valued technique. We can visualize this technique in a different manner assume that just before fetching the right instruction because of some reason in the transaction work stalled.

Well let us say the processor just went to sleep then the transactional memory would have never detected a conflict. And then the TM system transactional memory system would have happily allowed T x 1 to complete. And later on if T x 2 would have woken up it would have executed W x 2 no conflict would have been detected and then T x 2 would have gone forward.

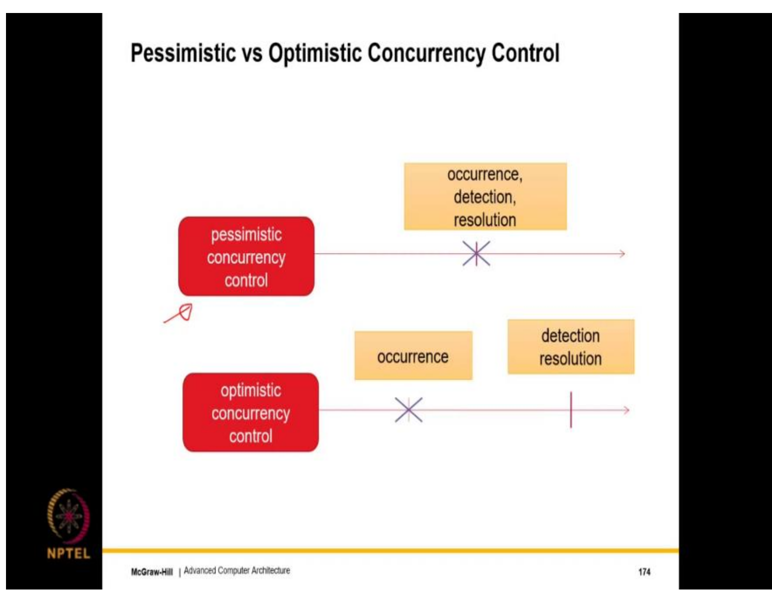
So, the delay mechanism is pretty much doing that whenever it sees a conflicting access it just stalls a transaction until other conflicting transactions complete and it restarts from there. So, if you think about it both the arguments are the same and correctness is not hampered.

(Refer Slide Time: 24:23)



So, there are three aspects in any transactional memory system, which deserve our attention. These are the occurrence of conflicts of course; this is an artifact of a program detection of a conflict. So, there are many ways to detect conflicts both in software as well as in hardware and resolution of a conflict. So, there has to be there has to be a means to resolve the conflict. So, that is the key idea that we see a conflict happening we need to detect and resolve.

(Refer Slide Time: 25:09)



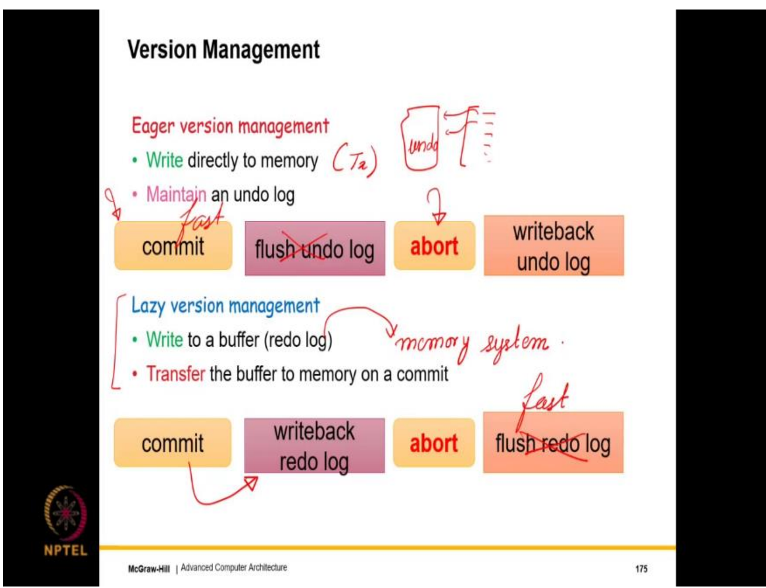
So, there are two paradigms the first is pessimistic concurrency control. So, in pessimistic concurrency control as soon as the conflict happens we detect it and we try to resolve it. Resolve it could be either delaying the transaction or aborting the transaction these are the two options that are available to us. But the in pessimistic approaches the moment that a conflict happens we detect and resolve.

In optimistic concurrency control the moment a conflict occurs we do not do anything at that point of time we wait till a later point of time and at the later point of time we detect and resolve. So, this basically allows the system to continue despite the fact that a conflict has happened which is safely allow the system to keep on executing as if we just allow the system to continue.

And at a later point of time we detect the conflict and resolve. So, both have their own pluses and minuses so pessimistic protocols are more overheads because we need to keep on detecting conflicts at every part of time which adds to our overheads. And in optimistic

concurrency control we typically detect the conflict at the end which is when we want to decide whether we commit or abort this whole transaction.

(Refer Slide Time: 26:37)



Now, let us come to version management. So, even in version management we will discuss a lot more about version management, but here also broadly speaking there are two approaches optimistic and pessimistic approaches. So, we can either have eager version management or we can have lazy version management.

So, in eager version management what happens that we write directly to memory. So, any transaction which is executing it just writes directly to memory nothing else. And then we maintain and undo log which means that just in case the transaction gets aborted whatever it wrote it writes its write set we just buffer the values that were overwritten for the first time.

So, the those are kept in an undo log and once the transaction aborts we take values from the undo log and repair the state. So, what is that we do if we commit; if we commit the transaction then of course, since all the writes have been returned to memory and nothing needs to be done we just flush the undo log.

But if we abort, so, in this case of eager version management with an undo log commits are fast. But if we abort for any reason then there is a need to read in the undo log once again and write back the undo log to repair the state. So, whatever may be written since

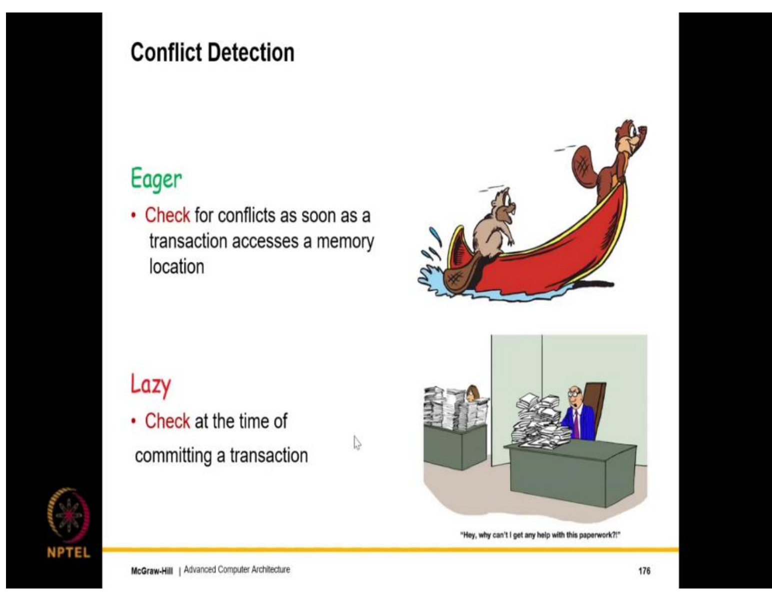
you want that none other writes are visible there is a need for state repair. Lazy version management, so, for lazy version management we write to a buffer which is a redo log in a sense whenever we do a writes we do not write it directly to memory.

But so we do not directly write it to the memory system, but we write it to a separate region which is known as a redo log. And once there is a commit we transfer the contents of the redo log to the actual memory system. So, the moment that there is a commit we write back the redo, so far so good.

That the moment that we commit a transaction the redo log is return back to the memory system; however, in this case aborts are fast. So, in the case of lazy version management aborts are fast in the case of eager version management commits are fast. So, aborts are fast, so, the thing is that since you have not written anything to the memory system all that we need to do is we need to flush the redo log and this we can do quite happily.

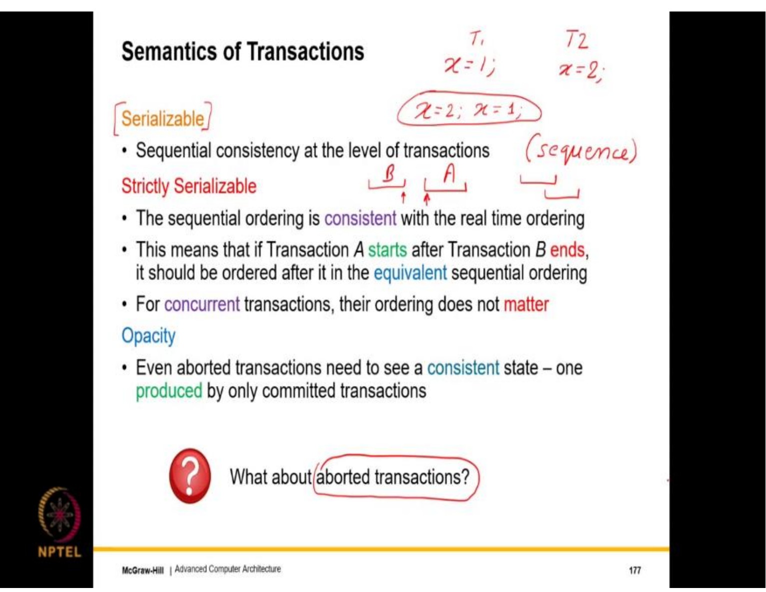
So, to summarize there are two approaches ether is that we are extremely optimistic and eager. So, we directly write to memory, but we maintain an undo log such that if there is an abort the state can be fixed. Or, we had an a pessimistic, so we write to a redo log and it is transaction commits later on then the writes from the redo log and return to the memory system.

(Refer Slide Time: 30:01)



So, let us now come to conflict detection. So, here also we can either be eager or lazy. So, you can look at eager, so as you can see the little bares on the right we check for conflicts as soon as a transaction accesses a memory location. This does increase our overheads wide substantial in the sense that accessing a memory location dose increase our overheads, because every single access now has to be burdened with checking for conflicts. So, you can take a lazy approach if you look at the files over here. So, in this checking is done at the time of committing a transaction.

(Refer Slide Time: 30:45)



So, let us look at the semantics of transactions. So, we had discussed different consistency models when it came to regular memory accesses. So, we have something quite similar in the case of transactions as well. So, the matrix that is commonly used is serializable.

So, this is also used in database transactions to a large extent, this is sequential consistency at the level of transactions. It basically means, it is possible to arrange all the transactions in a single sequence where we are respecting the program order of transactions within each thread.

So, it is this is like s c at the level of transactions we are strictly serializable. So, if you would recall then sequential consistency we do not really care about any real time ordering. So, let me give an example let us say that we set x = 1 in thread 1 and we set x = 2 in thread 2. So, let us say the first write happens and then the second write happens 2 days later.

Still in the sequential ordering we could have the write that happened 2 days later appear first and then the write that happen happened 2 days earlier appear later. So, sequential consistency will happily allow us absolutely no problem because the time at which an event actually happened is it does not matter it is symmetric.

And what really matters are, so whether program ordering is preserved or not and whether it is possible to arrange all the accesses in a single sequence which is legal. But strictly serializable adds a real time constraint. So, it makes the ordering consistent with real time ordering which basically means that if let us say there are two transactions that do not overlap in time, this means that transaction A starts after transaction B ends.

So, after transaction B ends if transaction A starts it should be ordered after it in the equivalent sequential ordering. So, this basically means we will look at all transactions and let us say you consider each pair in this case as you are seeing transaction A is starting at a point which is after the time at which transaction B ends. This means that, in the sequential ordering and mind you these transactions are issued by different threads not the same thread.

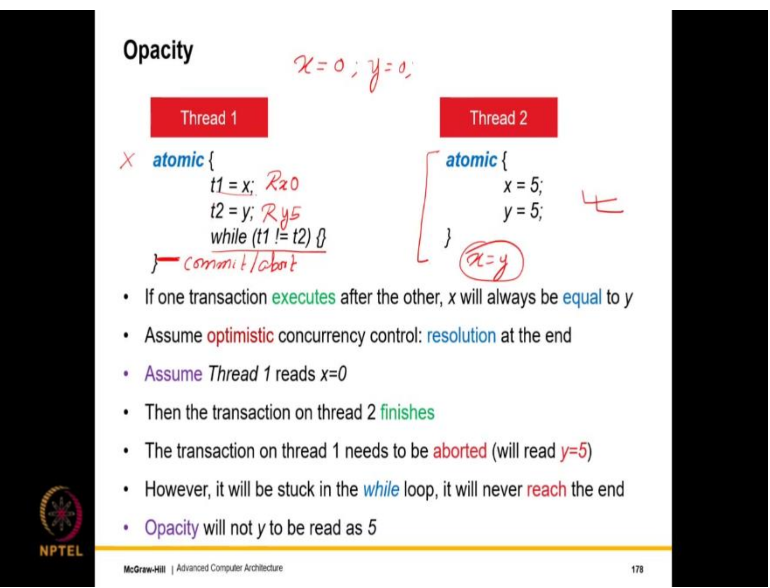
So, there is no program order between. Nonetheless in the sequential ordering final sequential ordering A will have to appear after B. Of course, if they are concurrent in the sense if there is an overlap between them then of course, the ordering does not matter. So, then the ordering is not specified, but otherwise strict serializability which is also known as linearizability, but I do not want to get into linearizability.

Because that has a different definition and it technically is not meant for transactions what is meant for transactions is strictly serializable? where we take the order into account. And the order basically says that they are non overlapping and one starts after the other, then the same needs to reflect in the equivalent sequential ordering.

Now, we will come to opacity, opacity is tricky opacity is tricky, but in a large number of cases it is actually required. So, we have only talked about committed transactions we have not said anything at all about aborted transactions. So, we have kind of remain mum about aborted transactions. so, to aborted transactions need to see a consistent state, which is produced.

So, a consistent state is a state that is produced by only committed transactions. So, what kind of a state should an aborted transaction see? This means that, an aborted transaction is it allowed to see a state which is always consistent or can it be slightly different at and what implications would that have?

(Refer Slide Time: 35:05)



So, let us study this in the context of an example. So, not in atomicity, but to study opacity. Let us look at two atomic blocks. So, initially as is our convention x is initialized to 0 y is initialized to 0, so that is our convention. So, the second transaction sets both = 5 which means at the end of this x = y.

And the first transaction does not write to x and y, but it only reads their values. So, what you would see is that in any execution? where transactions commit x = y. If let us say these are the only transactions x = y, but let us assume optimistic concurrency control where we detect conflicts and also resolve them at the end?

So, assume thread 1 reads x = 0, then the transaction on thread 2 finishes. Say the transaction on thread 2 finishes then clearly there is a conflict, but we at this point of time we realize that we will abort the transaction that is going on thread 1 and we will allow this transaction to commit.

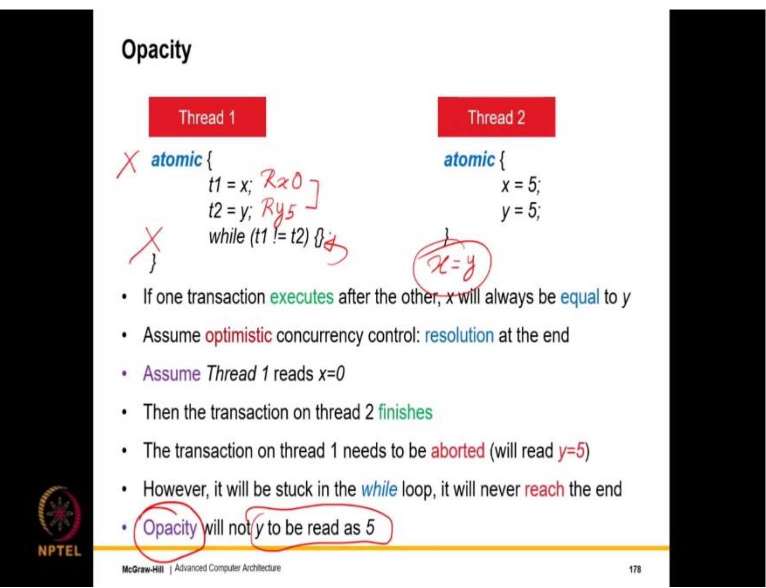
Then what will happen is that it will set x = 5 and y = 5 and x = y, but look at what was happened? What was happened is that even though this transaction is going to abort, but

since you are following an optimistic protocol they commit an abort decision will be made when we reach the end of the transactions. So, the compiler will add some degree of extra code to make a decision at this point of time. Well, the problem that has been caused is basically that we have read x = 0 and then this transaction has stalled.

Now, we finish the second one now we will read y = 5. If we read y = 5 if we look at this while loop over here which will basically loop 1 y t 1 ≠ t 2 will be stuck in the while loop you will never reach the end. And given the fact that we will never reach the end we will never reach the point where we make a commit or abort decision. Since, we will never reach that point or system is in jeopardy because one transaction has not aborted.

But it will just keep on running, it will keep on using the resources of the CPU will keep on running and the system will be in an incorrect state. So, the system will be in an incorrect state why? Well, because this transaction is just continuing to run in an infinite loop and no decision is being taken on it. So, if you think about it why if this happen?

(Refer Slide Time: 38:17)



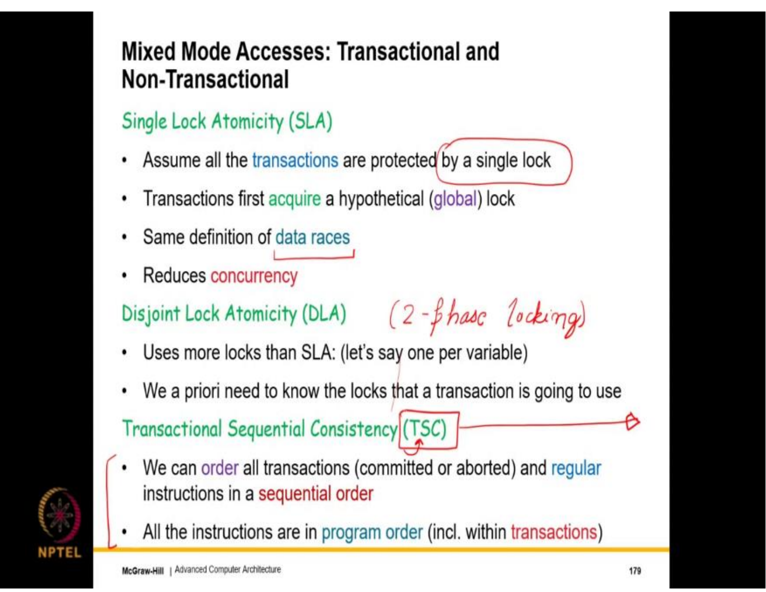
This basically happened because even though we were fully aware that this transaction is going to abort we allowed it to read an inconsistent state. We allowed it to read R x 0 and R y 5 which can never be produced by a consistent state, which is defined as just the set of all committed transactions that would never produce a result of this type. Because that will always produce a result which says x = y.

So, in a sense what has happened is that an aborted transaction has read an inconsistent state. So, if you want aborted transactions to always read a consistent state or I should add even aborted transactions to always read a consistent state, which is produced only by committed transactions? we will have the property called opacity. Opacity will not allow y to be read as 5, so this problem with opacity will never happen.

Which basically means? if we go back over here opacity means that, even aborted transactions need to read or see a consistent state, which is only produced by committed transactions, consequently these two reads will not happen and you will not be stuck in this while loop.

So, of course, this was kind of a contrived convoluted example you may argue, but many such cases do arise in real life. And many times arise inadvertently in real life and that is why opacity kind of safety. So, especially when you are looking at optimistic concurrency, particularly with software transactional memory opacity is definitely desirable.

(Refer Slide Time: 39:59)



Now, let us look at mixed mode accesses which look at both transactional accesses as well as non transactional accesses. Say, common correctness or consistence in model is called single lock atomicity where we assume that all the transactions are protected by a single lock. So, any time a transaction begins it acquires the lock and then it releases.

So, of course, this lock is a hypothetical global lock and data races are defined in the similar manner. But with this correctness model or let us say this can also be a practical model if you decide to implement transactions in this manner this will reduce concurrency that time. This will not allow disjoint access parallelism, not something that we want. See if we go back to our discussion on locks we could use standard two phase locking and have multiple locks.

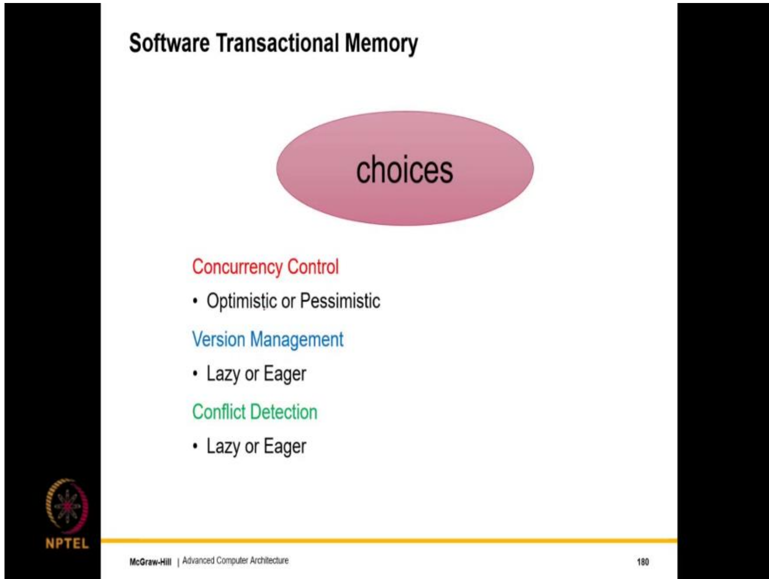
So, this is called disjoint lock atomicity. So, in this case we use many locks let us say one lock per variable, but the cache is that we need to a priori more which locks a transaction is going to use such that the locks can be acquired in a certain order right, in a certain lock order they can be acquired. So, this aspect was discussed quite a bit in the previous slide, but this is one more model where transactions use multiple locks let us say one per variable.

But we acquire them in a certain order to avoid deadlocks. Then we have this model transactional sequential consistency, where we basically assume that every non transactional instruction is basically a single instruction transaction. So, we can order all transactions either committed or aborted and regular instructions, where of course, what I am not saying here is a regular instruction is being assumed to be a single instruction transaction in a sequential order, everything.

Transactional, non transactional, committed, aborted does not matter everything can be arranged in a sequential order which needless to say follows program order and is legal. So, this is transactional sequential consistency this of course, gives us a lot it is hard to enforce in practice.

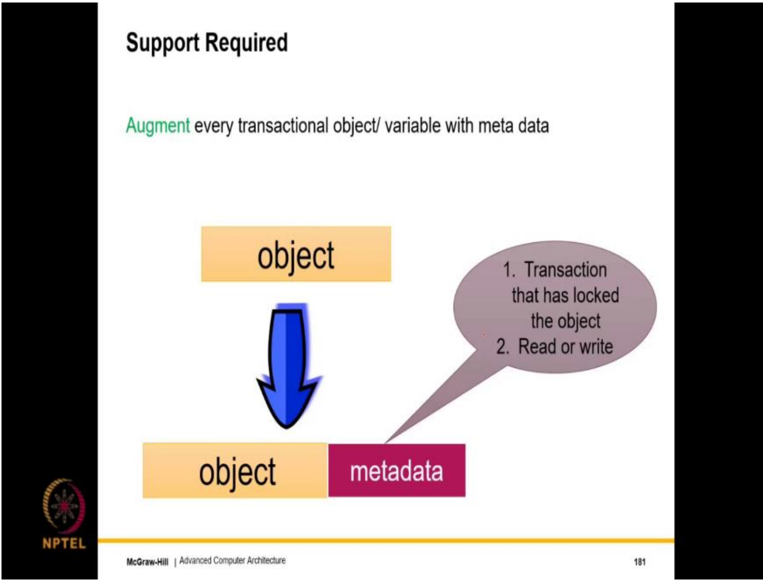
And as I said all the instructions in the equivalent sequential order are in program order including the ones which are within the transactions and transactions themselves, they are also in program order. So, this is like the gold standard of transactional memory. So, this is hard to implement in practice, but normally opacity is implemented.

(Refer Slide Time: 42:49)



So, now let us come to discussing practical protocol. So, we will discuss software transactional memory first. The choices that we have are a choice of the concurrency control mechanism optimistic or pessimistic, a choice of the version management mechanism lazy or eager, a choice of the conflict detection mechanism lazy or eager.

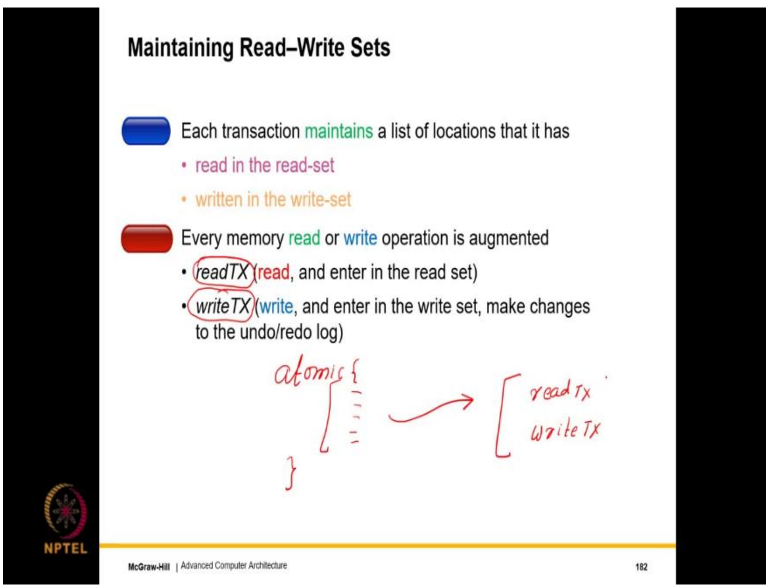
(Refer Slide Time: 43:19)



So, what is the basic support that is required? The basic support that is required is that we augment every transactional object or every variable with some degree of metadata. This basically means that if there is some metadata associated with an object we can keep track

of its state and also use that metadata to detect conflicts. Along with detecting conflicts we can also see, we can also lock variables and this can be used to stall other transactions. So, we can also associate a lock with the metadata.

(Refer Slide Time: 44:03)



So, let us now see how we have maintain read write sets each transaction maintains a list of locations that it has read in the read set and written in the write set. Furthermore, since we are talking abort software transaction memory every memory read or write operation is augmented.

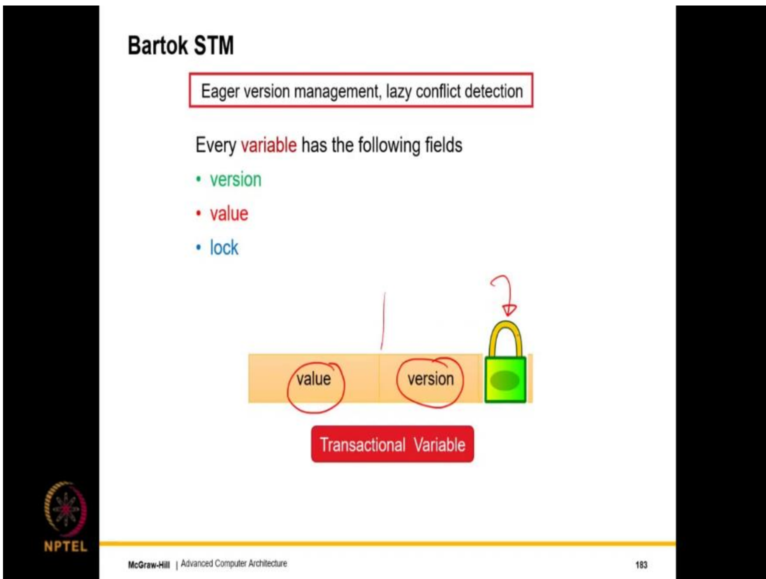
So, a read is augmented or it is replaced by a function now the function in this case can be something of the form like readTX which is basically you send the read operation to the memory and you enter the variable the address of the variable in the read set. WriteTX is the same function analog for a write operation where you are sending the fact that you want to write of course, the details of the variable.

So, that is implicit over here and we enter the address in the write set if it is not already there in addition whatever changes are required to the undo or redo log they are done. So, all of this is done automatically within the transaction the only thing that we do in the read the only argument that we actually sent is the address and the rest is all done internally.

Similarly, the only arguments that are sent to the write TX function by the program are basically the address that needs to be written in the value. So, many times a programmer

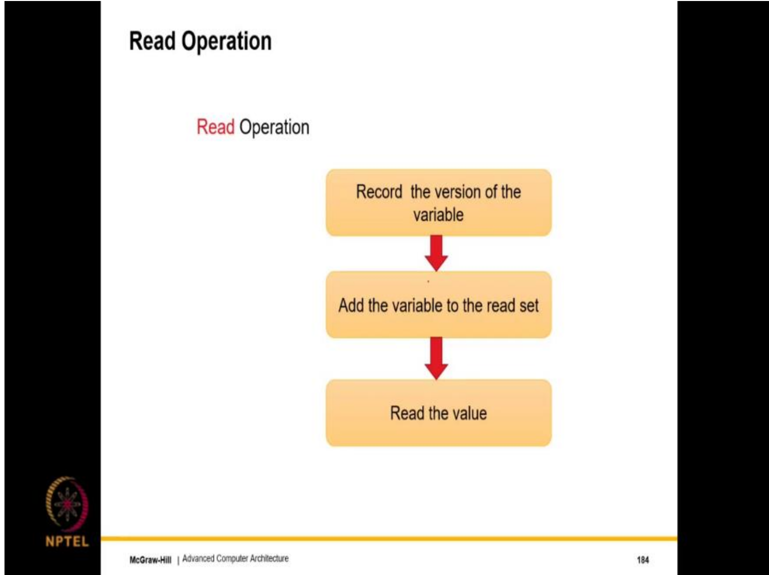
writes a normal program, but once it is read and closed in an atomic block the compiler automatically converts it to a program that has these read TX and write TX functions within them. So, what it does is that every access is replaced by a function call and the function call is readTX and writeTX.

(Refer Slide Time: 46:09)



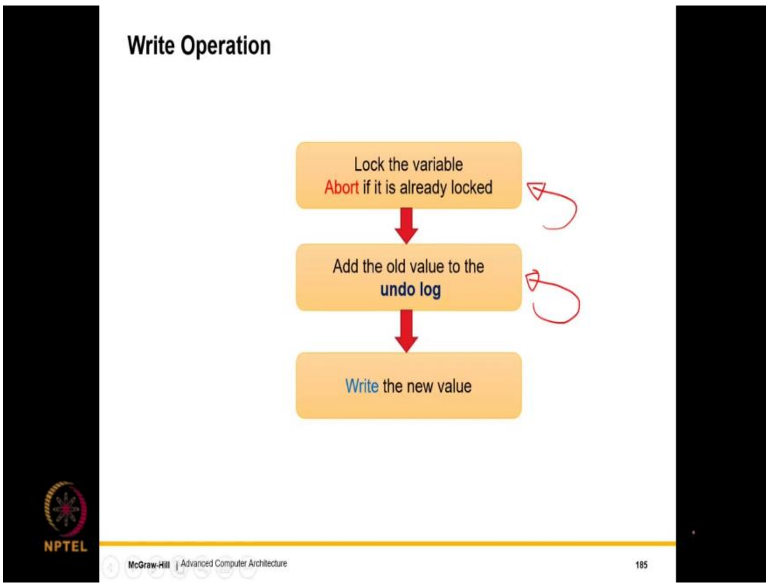
So, we first talk about the Bartok STM which is nice easy and simple and the references to this you will find in the book. So, here we have eager version management which means we have an undo log and we have lazy conflict detection. So, typically we do have lazy conflict detection in many software transactional memory proposals. So, this should not come as a surprise say every variable has the following fields it has the value, it has a version and it has a lock. So, this is what a transactional variable looks like then along with the value the extra fields are the version and the lock.

(Refer Slide Time: 46:51)



So, this is what a read operation would look like that when we read we record the version of the variable. So, we record what is the version and the variable that we have read in our read set. We add the variable to the read set and then we read the value it is as simple as that. So, reading in this case is an ultra fast operation here all that we do is we just record the version put the variable in the read set if required and return.

(Refer Slide Time: 47:23)



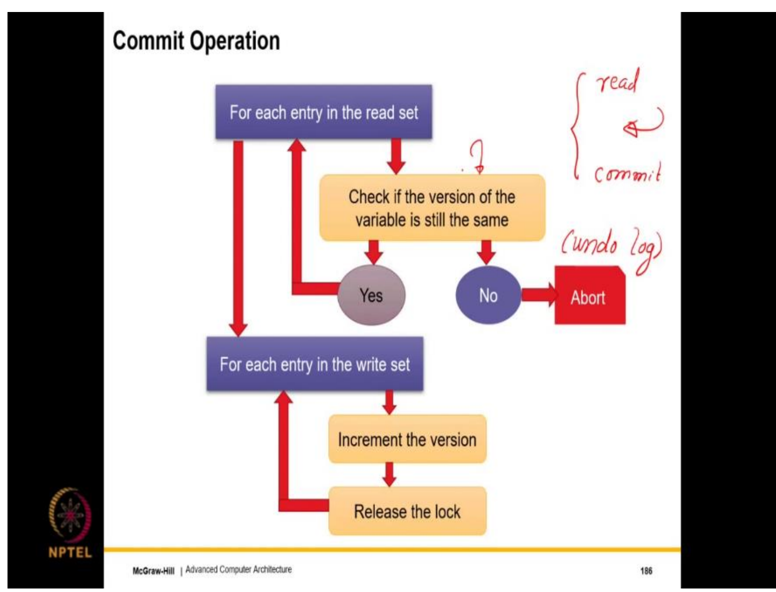
Writes in comparison are more elaborate we will see why. So, in this case we first lock the variable, we will see why there is a need to lock it we lock it at write time and release it at

commit time. So, we will see what is the need to actually do it, we add the old value to the undo logs recall that we are using eager version management.

See if let us say you are overwriting data then we add the old value to the undo log, but let us say there are two writes to the same address in a transaction. So, the second write this is not required. We will write the new value, that is all.

So, there is a key point over here is that the write is made more elaborate because we are locking the variable. And it is also being made more evaporate if let us say we are writing the variable for the first time in the transaction, so there is a need to update the undo log.

(Refer Slide Time: 48:37)



So, of course, if you are not able to find the lock for a variable then we abort. Now, let us come to the all complex commit operation. So, here what we do is that for each entry in the read set we check if the version of the variable is still the same. So, what you will see later on in the same slide is that after we commit we inclement the version of every variable.

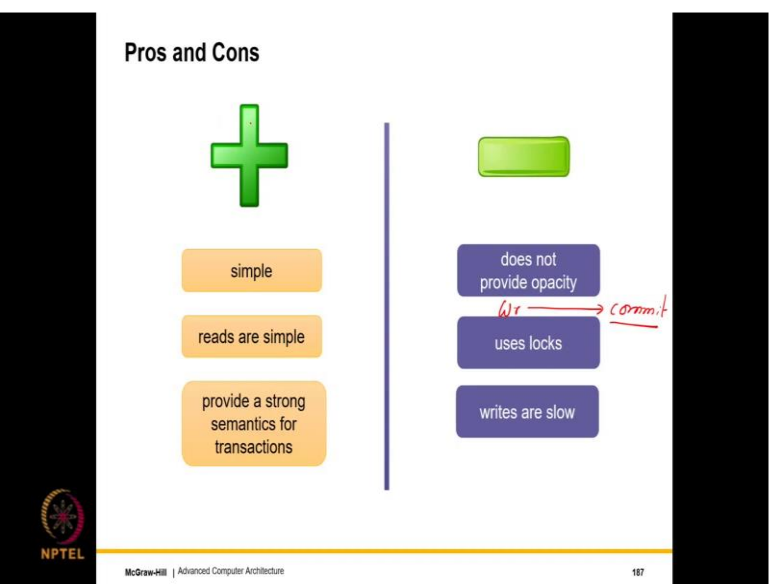
So, this will basically tell us from the time that we read the variable till the commit point has any write come in between or not. If a write has come in between then of course, there is a case for an abort. So, this will easily be found out if you check the version of the variable. So, recall that we are tying the value on the version of the variable together.

And so, then you have also both are like part of the same package, so we just check the version of the variable and that is it. Say the versions are not the same write has come in the middle, so we straight forward we abort. Otherwise, what we do is that the recheck passes, so we move to the write check and needless to say since we are using an abort the undo log has to be restored.

So, everything all the writes of the undo log have to be return back for each entry of the write set we increment the version and we release the lock. So, incrementing the version basically ensures or it provides a signal to others that the value of the variable has changed. So, just in case you read an older value now you need to abort, because the new value of the variable is what I am writing to write now and we release the lock.

So, the locking mechanism also ensures that the writes are mutually disjoint where period are disjoint you cannot write to the same variable by two threads at the same point of time because the second thread will never get the lock. And so, given the fact that the second thread will never get the lock you will never have a write write conflict and the read write conflicts can be detected by the version of the variable.

(Refer Slide Time: 50:53)



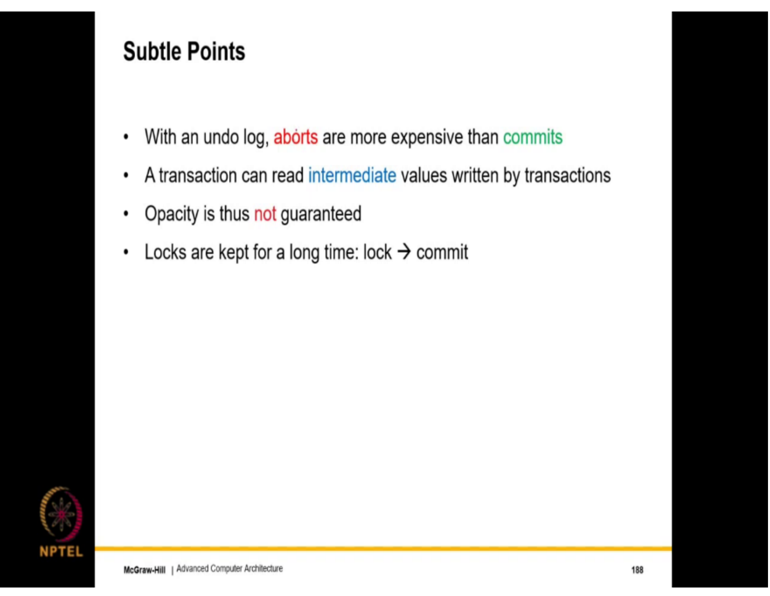
So, this is the simple idea, so the pros and cons are reads are simple they are fast it provides a strong semantics for transactions. But the negative aspect is that it does not provide opacity. So, opacity is something where an aborted transaction actually sees a consistent state when this is not provided.

And the other thing is that we are using locks and also a lock is being held for a large duration. It is being held from the time of a write till they commit you are kind of holding on to the lock and writes themselves become slow because of this locking. Now, before you would ask the question let me also say why it does not provide opacity?

The reason is that if there is an aborted transaction it will still because we are using the undo log base mechanism its writes will still be visible in memory, see if you go back to the right slide, which is over here you will see that its writes will be visible. So, we are writing the new value that is directly going to memory.

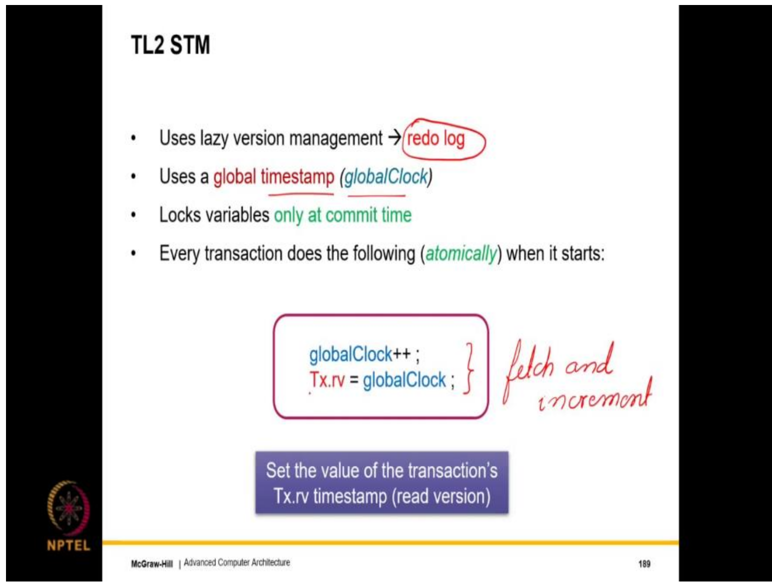
So, of course, for a variety of reasons we may decide to abort at commit time, but nonetheless its writes will be visible. So, because its writes will be visible and aborted transaction will not see may not see a consistent state and that will cause trouble this is why this algorithm does not provide opacity.

(Refer Slide Time: 52:23)



So, with an undo log also aborts become more expensive than commits, but you expect a transaction to commit most of the time aborts are not an issue. Furthermore, a transaction can read intermediate values which are written by other transactions hence as we have argued in the previous slide opacity is not guaranteed. And locks are also held and kept for a long time, so locking overhead is high.

(Refer Slide Time: 52:49)



So, now we will discuss the TL2 STM which is slightly more sophisticated in the sense it does provide for opacity and it uses a different method. Instead of eager version management it uses lazy version management which is a redo log. So, one thing you can see quite quickly is that if let us say opacity is supposed to be provided at the software level or redo log is required, because that will ensure that writes are not visible.

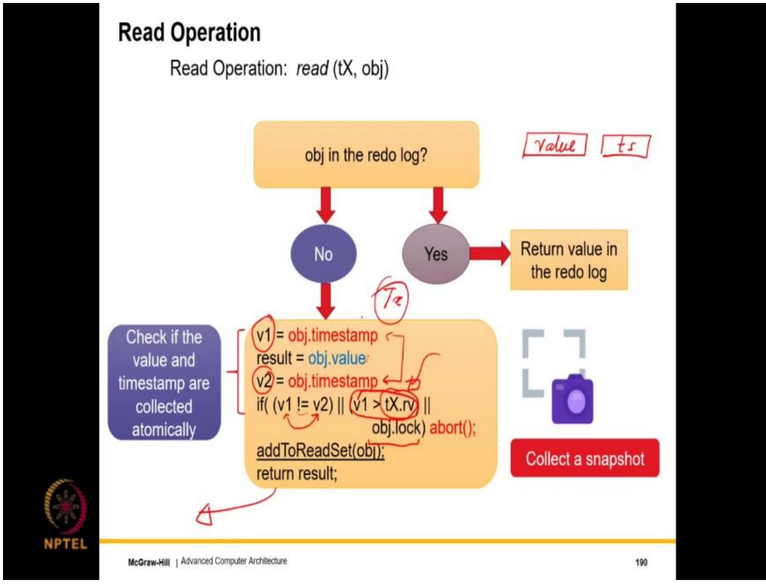
Writes will only be visible when a transaction commits which means that any other transaction be it aborted or committed will always see a consistent state. And a state which has been created by committed transactions. But here again there is a problem it uses the global time stamp or a global clock which needs to be accessed by all transactions. But a good aspect of this algorithm is that variables are locked only at commit time and every transaction atomically does the following right, at the time when it starts.

So, it increments the global clock, so both of done at these are atomics it is like a fetch and increment think of it this way. So, in this case it is more like an increment and fetch, but the idea of being the same. So, atomically does both of these operations which are easily supported in hardware the key point is that increments the global clock and every transaction has a read version at Tx dot rv a read time stamp.

So, the read time stamp is set to the value of the incremented global clock. And since they are happening atomically both of them together as a transaction, so of course, this is not a transaction in an transaction sense, but this is implemented using atomic instructions that

we studied when we are discussing coherence. These operations can be done and then the global clock is stored in the field Tx dot rv read version.

(Refer Slide Time: 54:53)



Now, what do we do when we are reading? Well, when we are reading a when you are reading of course, the transaction id is important and the object. Is the object there in the redo log? So, the object in this case can be a memory variable as well. So, if it is there in the redo log we return the value from the redo log if it is not there then we do a little bit of a trick.

So, what we do is that we first read the time stump. So, here the time stamp is being used as the version we use we read the time stamp into a temporary variable v 1 we read the value of the object. And then we read the time stamp yet once again and then we check if both the versions or the time stamps are the same or not.

So, why do we do that? The aim is to collect a snap shot because you have a value and a time stamp. So, what can happen is that you can read a value and then because of a race condition the time stamp may change or we may read a time stamp and because of a race condition associated value may change. So, both may be out of sync because one thing you need to realize is that there can be an arbitrary delay between these instructions.

So, maybe we may read a very old time stamp and then we may read the value and then we may read another time stamp. So, because of this arbitrary delay what we should verify

is whether the time stamps are the same. If they are the same then we can be sure that this value corresponds to this time stamps. This is a very standard programming area in concurrent systems where we actually read the time stamp twice such that the value time stamp pair is collected atomically.

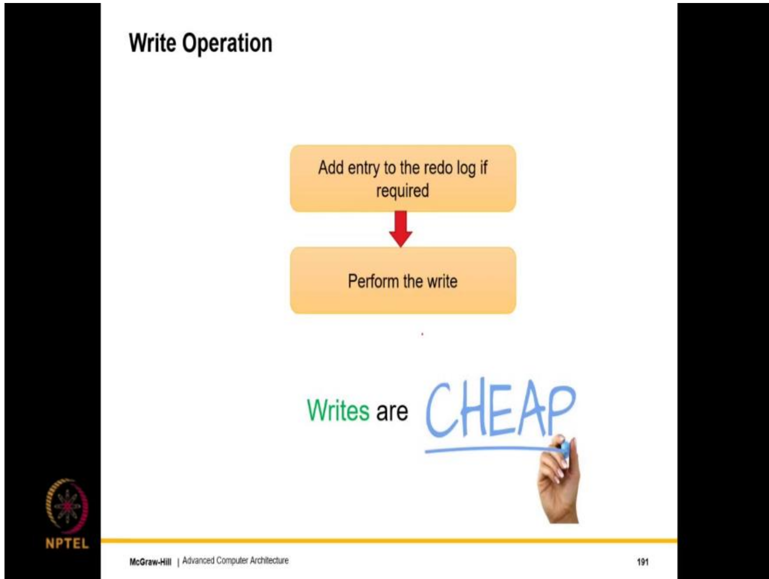
Where you know that look this value corresponds to this time stamp and this time stamp corresponds to this value fine. So, we verify that and we also verify, so, in this case too v1 = v2 otherwise, v will abort. We also verify if v1 which now we have proven which has = v2. If this is greater than the read version of the transaction which basically means that, some other transaction has written to this variable after this transaction started, so which makes the value steal.

So, if this is the case a v1 > tX . rv then also we abort which basically means a transaction started, but some others transaction has written to the variables after we started and it has committed also. So, given the fact that the variables have become still be bale out we abort and of course, if the object is locked then also we abort.

So, we check for these three thing, so this makes reads expensive. And then we add the read to the read set if this check succeeds and we return the result. So, the important points over here is this idea of checking the time stamp twice to ensure that the value and the time stamp are in sync.

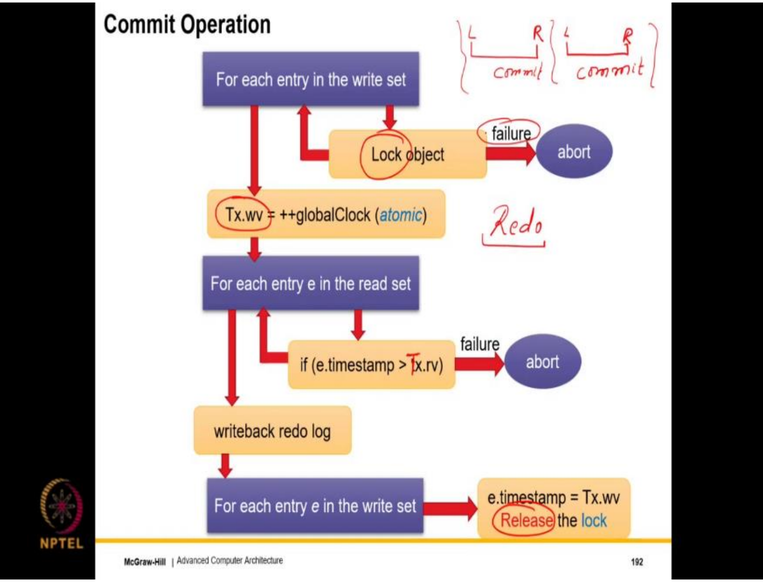
Because otherwise we can have arbitrary delays between these statements and the value in the time stamp may not be in sync. Because of the delays you might read a value, but the time stamp maybe for some other value that is the reason you read twice and compare. And also to ensure that once a transaction begins if any other transaction writes to the value after that then there is an overlap. So, we will have a read write conflict. So, this check over here eliminates the possibilities of this read write conflict.

(Refer Slide Time: 58:53)



Now, let us look at the write operation, so in this case thankfully the write operation is simple. We add the entry to the redo log if required which means that if let us say this is the first write to that address there is a need to add the entry to the redo log. Perform the write, writes are cheap, so in writes nothing need to be done writes are cheap no problem.

(Refer Slide Time: 59:17)



Coming to the all complex commit operation yet once again. So, recall that in the Bartok STM we were like heavenly into verifying reads, but in this case given the fact that we

have made reads expensive we actually do more with verifying writes. So, for each entry in the write set what do we do? We lock the object.

So, this is only a temporary locking for the period of the commit operation. So, in this case we hold locks for a much smaller duration. So, we lock the object and of course, the locking is done in such a way to avoid deadlocks, in case there is a deadlock it is detected. So, the programmer need not bother it is detected and recovery is initiated. So, the programmer as such need not bother, but for each entry we lock it this is the failure we abort.

Then what we do is that we inclement the global clock yet one second atomically and the incremented value becomes the write version. So, we have two time stamps rv and wv, so rv is read version, wv is write version. So, for each entry in e in the read set. So, this should be capital Tx, so wherever just have capital Tx. So, for each entry e in the read set what do we do? We check if the time stamp is still greater than Tx dot rv.

So, this is the same check that we were doing over here it is exactly the same check that we were doing over here, we were checking the version with Tx dot rv. So, that correction will be made, but here the key idea is that we do this check once again, which means that the at the time of committing also we are ensuring that there is no read write conflict.

The reasons remain the same after that what we do is that we write back the redo log and for every entry in the write set we incremented its time stamp. How is that done? Well, the time stamp is set equal to the right counter over here and we release the lock. So, this is all that needs to be done. So, the commit operation is kind of similar at least in principle conceptually looks the same as the Bartok STM, but of course, in this case we have a redo log.

And the locking is for a much shorter duration in the sense we lock over here we release over here. And of course, before we lock in that case we are incrementing the version in this case we increment the time strap. And for reading we just verify that the there is no read write conflict.

Now, how do we verify if there is no write write conflict? Well, that is quite simple given the fact that we are using a redo log it could very well be possible that many other transactions are writing to the same write set and there could be a write write conflict. But

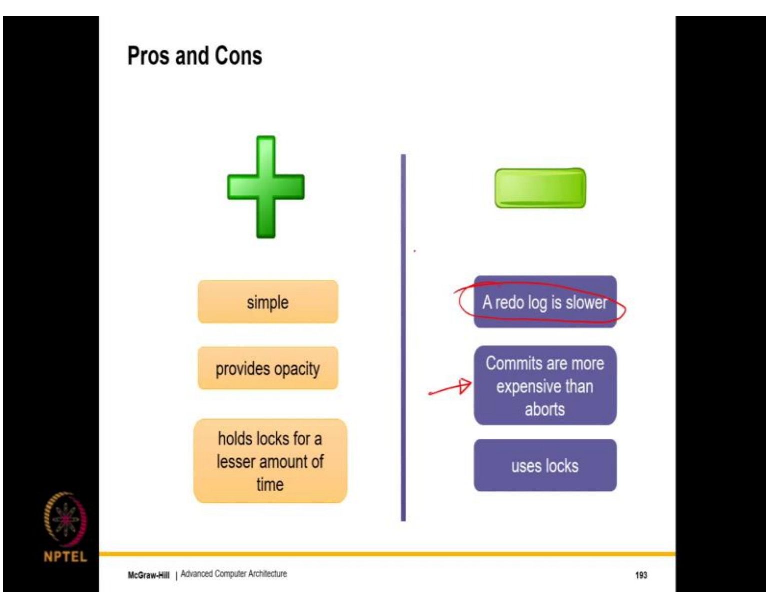
in this case keep in mind that for the entire duration of the commit we are essentially locking the entire write set.

If there is an overlap and we are trying to write at the same point of time one of the lock operations will fail and that is when one of the transactions will abort. If let us say two transactions with overlapping write sets are succeeding it basically means that one transaction commits. So, committing meaning at the beginning it locks the variable and then it releases then the second transaction commits, lock and release.

So, in this case as you can see there is no problem even if the write sets do overlap because of a redo log the write are visible only after a successful commit. And further more write sets can overlap there is no issue because the commits are still being arranged in sequential order. But, if there is an overlap in terms of time, which is actually not allowed, because that will become a concurrent and conflicting access.

Then the lock failure will indicate that two transactions are writing to the same lock, same object at the same point of time and that would indicate that there is a need to abort. So, that is why a write write conflict with the redo log is not a big deal the read write conflict is the bigger deal, but that we are solving using time stamps.

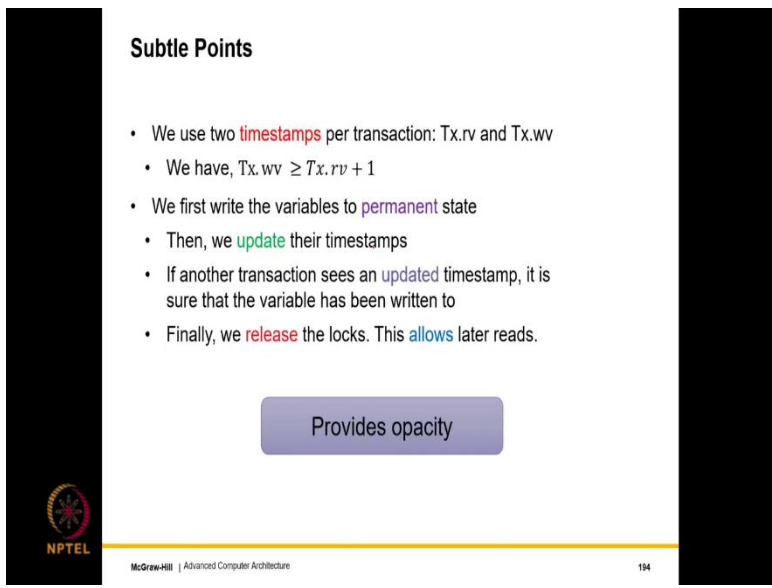
(Refer Slide Time: 64:01)



Pros and cons well a simple idea it does provide opacity well, why opacity? Because the moment we use the redo log we only see the state provided by, created by committed

transactions you also hold logs for a much it is for lesser amount of time. Of course, the redo log is slower it makes commit slow that is ok we are getting a lot we are getting opacity. So, a redo log even though it is slower and commits are slower that is ok. You still use locks, so we will try to remove locks when we go to hardware transactional memory. But in general in software transactional memories locks are required.

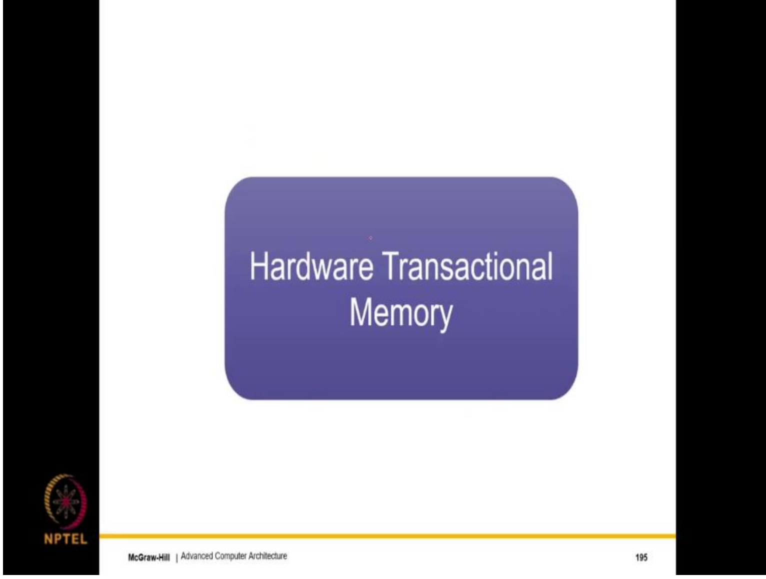
(Refer Slide Time: 64:43)



So, what is the subtle point? Well, the subtle point is we are using two time stamps per transaction Tx . rv and Tx . wv. So, it will have Tx . wv $\geq$ Tx . rv + 1. So, why greater than equal to? Well, it could be equal to if there is no transaction in the middle because there are two increments, greater than it can be if other transactions have committed in the middle. So, you first write the variables to permanent state then we obtain their time stamps.

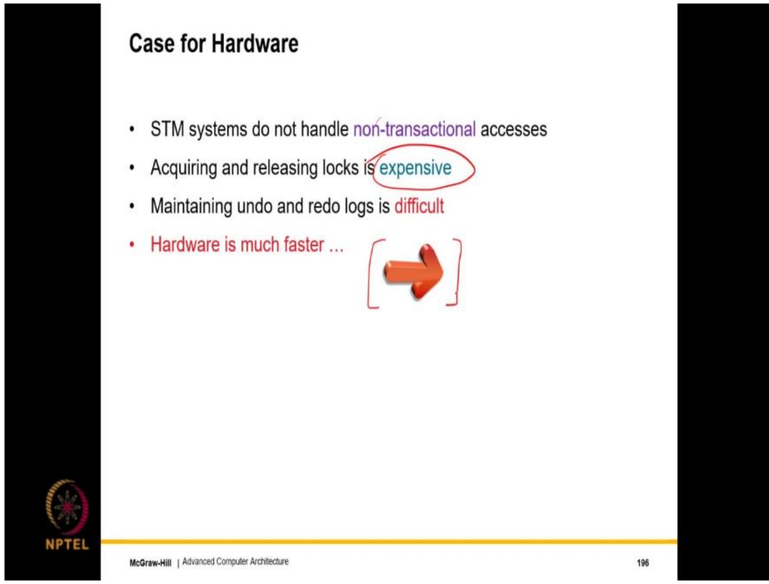
This is typically what is done when you have a value time stamp for more you first write the variables themselves to permanent state and then we update the time stamps. If another transaction sees an update its time stamp it will automatically figure out that the value is there in permanent state, this justifies the order. And finally, we release the locks, so this allows later reads to both, so this provides for opacity as well. So, now, we will discuss hardware transactional memory as you can see.

(Refer Slide Time: 65:49)



So, hardware transactional memories main advantage would be that it will not use locks.

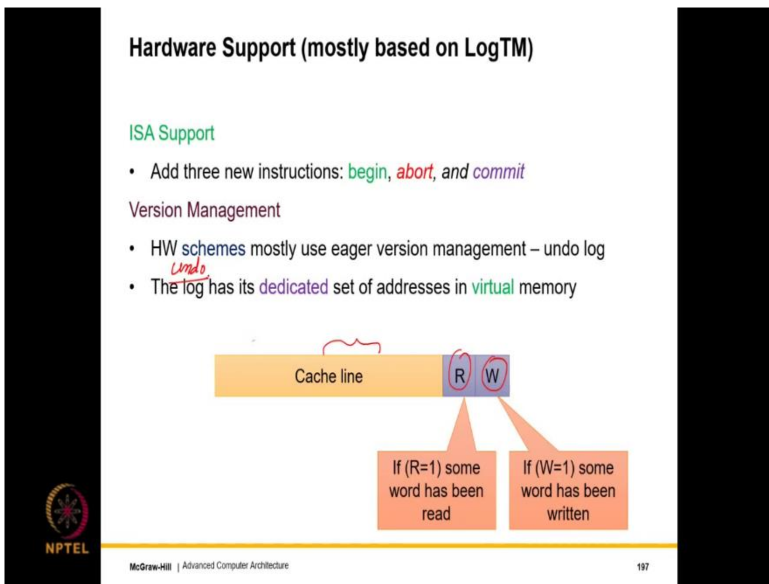
(Refer Slide Time: 66:01)



So, let us now make a case for hardware. So, STM systems do not handle non transactional accesses, the reason being that these accesses are not instrumented. So, we have not been replacing them with function calls. So, that is the reason non transaction accesses even though they can cause a lot of trouble data races and so on, but they are not handled. Acquiring, releasing, managing locks is quite expensive, maintaining undo and redo logs

particularly in software is also difficult. Hardware is much faster, but you have to create the hardware which is there are it is expensive there are high start up costs

(Refer Slide Time: 66:39)

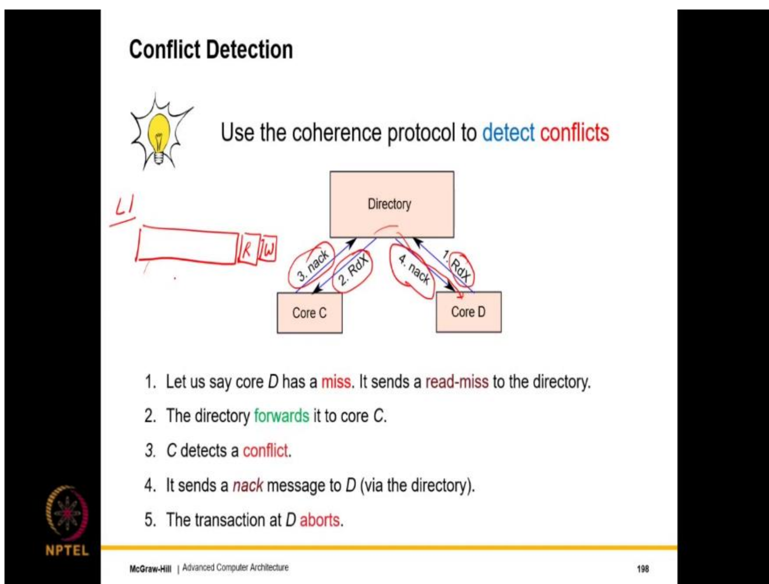


So, the hardware support which is which will be discussed is mostly based on the widely popular LogTM proposal you will find references for it in the book. So, the basic ISA support that is the read is we need three new instructions for begin abort and commit. Version management hardware schemes mostly use eager version management they use the undo log.

It is hard to maintain a redo log because you know you need a separate region and so on, but the undo log is the cache, think of it the basic L 1 cache, L 2 caches they can be used as the undo log. And it is much easier to do. So, we will see how that is the reason eager version management is used. The log has its dedicated set of addresses in virtual memory. So, in this case, this is if we are using a redo log that is otherwise, we just simply use the cache.

So, the cache line every cache line is augmented with 2 bits read and write. If R = 1 this means at some word some 4 byte word out of this cache line has been read and it is a part of a transaction, so the it is made a part of the read set. And if any word has been written to then we said W = 1 which means that some word has been written to, so it is made a part of the write set.

(Refer Slide Time: 68:15)



So, here is the, so see here is the key idea we modify we make a small change to the regular directory based cache coherence protocol to detect conflicts. So, this is the hallmark of any of these transactional memory schemes that instead of designing a new protocol we use the regular cache coherence protocol to augmented to detect conflicts in the case of transaction memory.

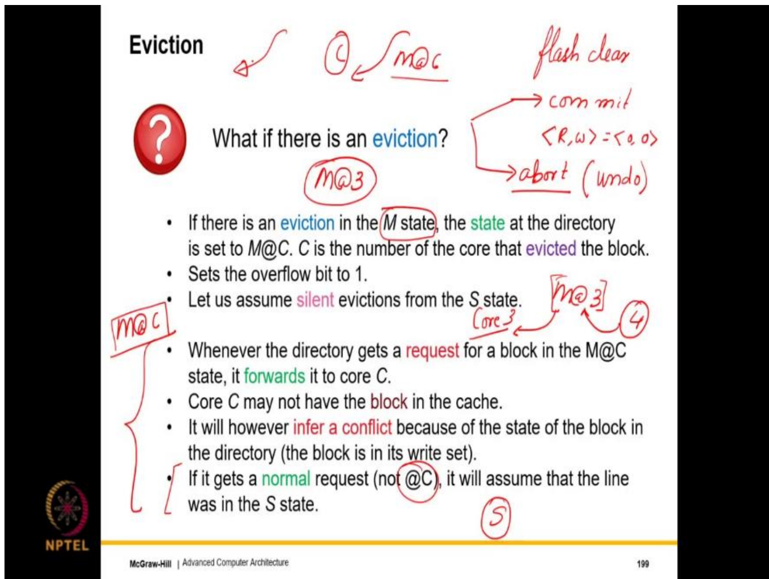
So, let us say core D has a miss it will send, so let us say core D has a miss it will send a read miss to the directory. The directory will find that core C is a sharer it will forward the read miss to core C, core C will detect a conflict. How will I do that? Well, it will try to read the lines from its cache it will find that it has returned to its which means it is a part of its write set.

So, if you see that there is a conflict, so it will send in nack back to the directory, then nack message will then be bounced from the directory it will be send to core D. Core D will realize that there is a conflict, so the transaction at D will abort. This can be done either in hardware or with some software support. So, the support that is required is, so the transaction you will have to clean up the state and retry. So, typically a signal may be sent to software to retry the transaction.

But the key point the operative point here is and every cache line is being augmented with two bits a read bit and a write bit. And they are indicating whether the line is in the read

set or in the write set. But of course, what happens if the line gets evicted here goes to L 2 cache those are all corner cases we will gradually deal with them.

(Refer Slide Time: 70:01)



So, the key point whenever the cache is being used as the undo log is that if let us say the entire read and write set can be confined to the cache then there is no problem. It can be confined to the L 1 cache for example, then there is no problem. Because if there is a commit all that we need to do is that we need to set the read write bits to 0 0, which means they are not the part of they are not a part of a read write set anymore because the transaction has committed.

And there are mechanisms known as flash clear mechanisms where basically what we do is that in the SRAM array, we have these read write sets as a separate sub array and we set all of them together in one go. So, descriptions are there in the book regarding how exactly this is done and there are any patents as well. But the key idea is that this is the known mechanism it is a simple mechanism of how you set a large number of bits to 0 and 1 go.

If of course, there is an abort then also we need to do a flash clear, but there is a need to bring in the undo log and write it. So, one small point is due over here. So, when I meant that the log has a dedicated set of addresses if you are using a redo log for sure yes, but even when you are using an undo log the undo log has to be stored somewhere.

So, where will we store it? The undo log will be stored somewhere in a separate region in virtual memory, which will be accessible to the process and to the hardware. So, if there is a need to undo the hardware will automatically read all the variables from there and update the state, such that the state is similar to what it was before the aborted transaction started.

So, fixing the state can be the; can be done by either software or by hardware. But the key point is that while a transaction is an execution if there is an eviction what happens? See if there is an eviction in the M state the state at the directory is set to M at C, where C is the number of the core that evicted the block.

For example, let us say if core 3 evicted the block the state of the directory will be M at 3. M at 3 basically means it was there in the modified state M core 3, but core 3 evicted the block from L 1 to let us say L 2, we set the overflow bit of core 3 to 1. In the S state, let us assume that there are silent evictions which means that in the underlying directory protocol also, if let us say there is an eviction in S state it does not necessarily inform the directory.

So, with that in mind here is what we do? Whenever, the directory gets a request for a block in the M at C state. So, basically this state let us say M at 3 and it gets a request from another core let us say core 4, then it will forward that request to core 3. So, core 3 in this case, so in this case core 3, but we are calling it core C actually. So, core C may not have the block in the cache, the reason being it may not have it in the L 1 cache because it is displaced to L 2.

It will nonetheless infer a conflict because of the state of the block, because the directory base also sending the its current state of the block along with the message. It will see that the directories current state is M at C because it is M at C it will infer that since the directory never lies that most likely I had it in the modified state and I evicted it and now somebody else wants to access it. But this block for sure is not my write set, so it will infer a conflict.

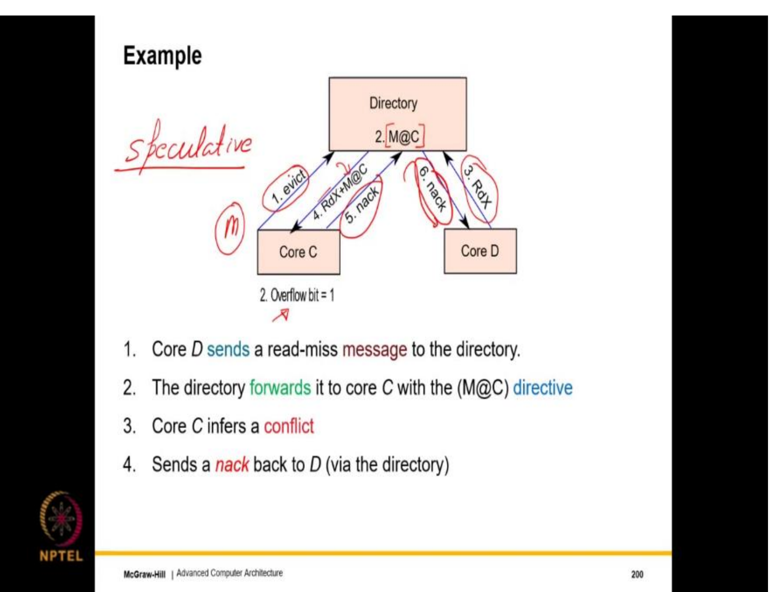
So, then what it will do is that it will need to do something. If it is a normal request, So, if it gets a message with M at C then it will assume that there is a conflict and it will send a nack back and the nack will lead to an abort. But if it is a normal request which means this

M at C is not coming then it will infer that the line was there in the S state and from the S state it was evicted.

So, this is what it will infer. And if the other transaction just wants to read there is no problem, but if it wants to write there is a read write conflict and a conflict has to be signaled. So, in this case what you see is you see a strange way of state management. So, what is happening is that depending upon the message that is coming from the directory.

See if the directory is sending a message with an M at C to core C, it will figure out the look yes I do not have the line currently with me in the L 1 cache. But since the directory is sending this message most likely had it in the M state and it got evicted. And let us say if no where if a request is still coming it will then assume that most likely had it in the S state and it got evicted.

(Refer Slide Time: 75:51)



So, let us look at an example over here core D will send core D starts the process. So, the first thing is of course, that, so I will start slightly from to the left word some t = 0. So, the first is core C had a line it evicted it, so it sent a message to the directory. The directory sets the state to M at C primarily because core C was the only core which had the block in the modified state.

So, it will set it to M at C and then what it is going to do is that it will not do anything it will wait for subsequent requests. When core D sends a read miss message to the directory

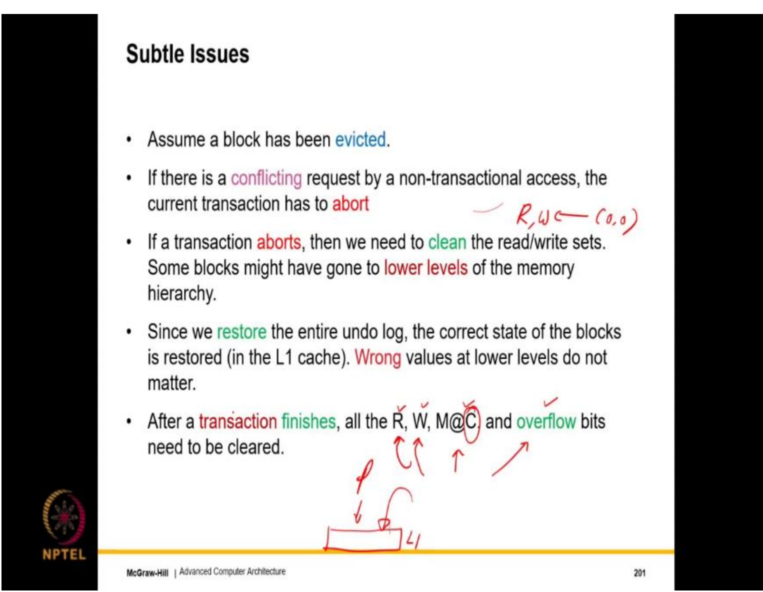
it will realize that its current state is M at C. So, it will send the read miss message and M at C directive, core C will realize several things.

It will realize that its overflow bit is set to 1 because this is the set just after the evict. So, overflow bit basically means that some line was in the read set or write set and it was evicted to the lower level, so, its overflow bit is 1. If its overflow bit is one and the state that is coming has an M at C, then what will happen?

It will automatically infer that there is a conflict because I had it in the modified state I sent it down that is ok. But its till in my write set in the modified state and somebody else wants to read it, so there is a conflict. So, it will send a nack back the nack will bounds from the directory go to core D and it will be a nack and core D will subsequently abort.

That is the key broad idea over here that core D will subsequently abort after it gets a nack from core C via this directory. So, this allows speculative data. So they, so that is the term speculative data because this data finds entries and read and write sets. So, it allows speculative data to be displaced from the higher level caches and send to the lower level caches.

(Refer Slide Time: 78:11)



There are of course, some subtle issues. So, I see my block has been evicted it is a conflicting request by a non transactional access then of course, the current transaction has to abort because the non transactional access cannot abort. If a transaction aborts then of

course, we have to flush the read write sets. So, the read write sets flushing basically means that all the R and W bits have to be set to 0, all the R and W bits have to be set to 0, 0.

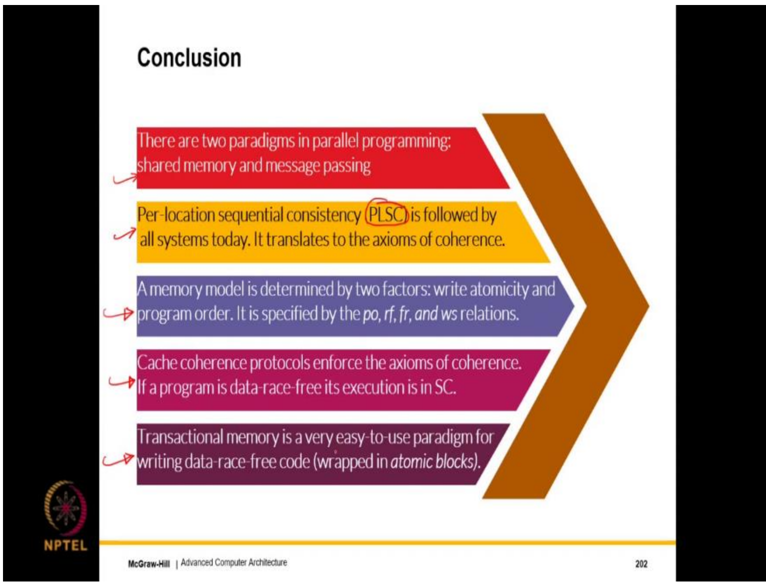
In this case what could have happened is some blocks might have been evicted and gone to the lower levels of the hierarchy. But since we restore the entire undo log what will happen is, so, the correct state of all the blocks will get restored in L1 cache. In this case even if there are wrong values written by an aborted transaction and they are floating around in the lower levels of the cache hierarchy it does not really matter.

Because if you think of a processor it only trusts all the data that is there in L1 cache something that is below L1 is assumed to be steeled anyway, because L1 cache has the most up to date data. Since the undo log is directly written to the L1 level can get evicted later that does not matter.

But if the undo log is written to the L1 level then as far as the processor is concerned the most up to date data is there, which is the undo log. Which means, any other value that was generated by the aborted transaction, even if it is floating around at lower levels its does not matter.

After the transaction finishes all the read write bits of each line, all the overflow bits with the core and the M at C kind of bits, bit C is the core number all of these are clear. Clearing these two is trivial, but we have already discussed how to clear these. And to clear these well there is something called a flash clear mechanism which has explained in some detail in the book.

(Refer Slide Time: 80:11)



Good, so now, we have completed this huge long humongous chapter. So, what we have seen is there are two broad paradigms in parallel programming shared memory and message passing. Message passing is far more suitable for a loosely coupled multi processing system, but for any multi core system shared memory programming has used.

Here of course, there is a need to create specifications. So, PLSC is something that all memory models follow an adhere to and that lays the foundation for cache coherence. Memory consistency models are of many types the key important variables are write atomicity and program order.

These relationships are captured by the po, rf, fr and ws relations which we have discussed in great depth. And to enforce the axioms of coherence where the serialization axiom comes from PLSC and the right propagation axiom comes more from common sense then we do not want writes to be lost.

There are many ways many protocols snoopy protocols, directory protocols to enforce the axioms of coherence. And then we looked at data race freedom and we looked and we defined, what is a properly synchronized program and if a program is data race free its execution is in SC.

Finally, we talked about transactional memory, it is correctness properties, different protocols, hardware protocols, software protocols and how easy it is to write correct data

race free code. Of course, wrapped in atomic blocks and transactions where the underlying system does the rest, so this concludes this chapter.