Analysis of Real-Time Multi version Concurrency Control Algorithms using Serialisability Graphs

K M Prakash Lingam
University Visvesvaraya College of Engineering, India

ABSTRACT
This paper analyses the correctness of Multiversion Concurrency Control(MVCC) algorithms that are commonly deployed in Real-time Databases. Database systems for real-time applications must satisfy timing constraints associated with transactions. Typically, a timing constraint is expressed in the form of a deadline and is represented as a priority to be used by schedulers. MVCC Algorithms used here makes use of a specialized version of Serialisation Graph, Called MultiVersion Serialisation Graph(MVSG) to resolve data conflicts to maintain the serialization order among conflicting transactions. Using MVSG,MVCC algorithms can determine which lower priority transactions should be aborted to avoid deadlocks.

Keywords: Transaction, Multiversion, Schedule, Serialisable.

1. INTRODUCTION
The prime objective of real-time database systems is to maximize the number of transactions that should be executed with timing constraints[1]. A real-time database systems should schedule the transactions based on its priority i.e., its deadline, its importance in the real-time scenario, elapsed time etc., For example, supposing there are two transactions T_1 and T_2 with T_1 having the smaller deadline. The Real-time transaction manager schedules and favors T_1 so that T_1 can be finished before T_2. Whereas in contrast, Conventional database techniques schedules and treats all transactions with equal priority. So, the deadline requirement of the transactions cannot be met with Conventional database techniques.

Concurrency control techniques are required for transaction scheduling in order to maintain consistency of data. Common real-time concurrency control techniques are based on Two-Phase locking(2PL) and Validation(Optimistic Techniques).Concurrency control is the activity of synchronizing operations issued by concurrently executing programs on a shared database. The purpose of the concurrency control is

- To produce an execution that has the same effect as a serial (noninterleaved) one.
- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

When a transaction performs conflicting operations on data objects among active transactions(Transactions that are in execution and not yet committed),the transaction manager resolves conflicting operations using one of the following two approaches.

(i) Blocking(Transactions made to wait)

(ii) Rollback.

Two-Phase Locking uses (i) to resolve data conflicts by blocking the transaction that causes it.2PL requires information about the transactions such as read/write data sets to be locked and the timestamp interval.Optimistic approach uses (ii) to resolve data conflicts by aborting the transactions that conflict with the transaction having a smaller timestamp. But it outsmarts 2PL by discarding tardy transactions and it also ensures that it doesn't disturb other active transactions. The disadvantage of this approach is that it could result in unnecessary restarts of the aborted transactions but this could be controlled by adjusting serialization order among concurrent transactions.

Multiversion concurrency control (MVCC) algorithms have been tailored to suit real-time databases by using priorities of transactions. Since every write operation produces a new version of data objects, there is no conflict among write operations. The main difference between multiversion and lock models is that in MVCC, locks acquired for querying (reading) data don't conflict with locks acquired for writing data and so reading never blocks writing and writing never blocks reading.

The scope of this paper is to provide a comprehensive view on MVCC algorithms deployed in practice for real-time databases to increase the degree of concurrent transactions and avoid unnecessary aborts which could degrade the performance of the system. This paper is organized as follows: Section 2 discusses the serialisability concepts behind MVCC. Section 3 brings out the technical details in MultiVersion Serialisation Graph(MVSG). Section 4 describes the MultiVersion Timestamp Ordering algorithms. Section 5 describes the MultiVersion Two-Phase Locking Algorithms. Both Section 4 and Section 5 hints on the problems in applying these algorithms to real-time databases and proposes the solution to it. Finally, Section 6 gives concluding remarks.

2. MVCC Serialisability Concepts
Multiple transactions can be executed concurrently by interleaving their operations. Operations include read (r), write (w), commit (c), abort (a).A Schedule (or history) S of n

transactions $T_1,T_2,..T_n$ is an ordering of the operations of the transactions subject to the constraint that, for each transaction T_i that participates in S, the operations of T_i must appear in the same order in which they occur in T_i. However the operations from other transactions T_j can be interleaved with the operations of T_i in S.

A schedule S is serial, if for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule, without any interleaved operations from the other transaction. So it represents inefficient processing and can lead to low CPU utilisation while a transaction waits for disk I/O, or for another transaction to terminate, thus slowing down processing considerably. Every serial schedule is correct because only one transaction at a time is active - the commit (or abort) of the active transaction initiates execution of the next transaction.Therefore, all the serial schedules can leave the database in a consistent state.

For *non-serial* schedules the goal is to determine which, are correct and which are erroneous. The concept used to characterize schedules in this manner is called serialisability of the schedule. A schedule *S* of *n* transactions is serializable if it is equivalent to some serial schedule of the same *n* transactions. Serializable is not the same as being serial. Serializable implies that the schedule is a correct schedule. So, it will leave the database in a consistent state. The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

2.1 Conflict Equivalence

Eg: T_1: r(x) w(x) r(y) w(y) c

T_2: r(x) w(x) c

Two operations are said to **conflict** if they satisfy all three conditions:

(i) they belong to different transactions

(ii) they access the same item

(iii) at least one is a write operation

Ex.: A sample schedule

S_1: $r_1(x)$ $r_2(x)$ $w_1(x)$ $r_1(y)$ $w_2(x)$ $w_1(y)$

In the above example, $r_2(x)$ and $w_1(x)$ are conflicting operations because they satisfy all the above three conditions. Similarly $w_1(x)$ and $w_2(x)$, $r_1(x)$ and $w_2(x)$ are the other two conflicting operations.

Two schedules are conflict equivalent, if the order of any two conflicting operations is the same in both schedules. A schedule is Conflict Serialisable if it is conflict equivalent to some Serial schedule. Ex: Another sample schedule.

S_2: $r_2(x)$ $r_1(x)$ $w_1(x)$ $w_2(x)$ $r_1(y)$ $w_2(x)$ $w_1(y)$

Now S_1 and S_2 are conflict equivalent because the order of all the three conflicting operations are preserved. Interleaving of operations occurs in an operating system through some scheduler, which allocates resources to all processes. Factors such as system load time of transaction submission and priorities of processes contribute to the ordering of operations in a schedule. It is difficult to determine how the operations in a schedule will be interleaved beforehand to ensure serialisability.

2.2 View Equivalence

Another less restrictive definition of equivalence of schedules is called View Equivalence. Two schedules are said to be **view equivalent** if they have the same reads-from relationships and the same writes. The same reads-from relationships amounts to having the same Read operations. Consider the following schedule of three transactions

T_1: $r_1(x)$, $w_1(x)$ T_2: $w_2(x)$ and T_3: $w_3(x)$

Schedule S_3: $r_1(x)$, $w_2(x)$ $w_1(x)$ $w_3(x)$

In S_3, the operations $w_2(X)$ and $w_3(X)$ are blind writes, since T_2 and T_3 do not read the value of X. S_3 is **view serializable**, since it is view equivalent to the serial schedule T_1, T_2, T_3. However, S_3 is **not conflict serializable**, since it is not conflict equivalent to any serial schedule. In a multiversion concurrency control algorithm, each Write on a data item x produces a new version of x. The Transaction manager(TM) that manages x therefore keeps a list of versions of x, which is the history of values that the TM has assigned to x. For each Read(x), the scheduler tells the TM which one of the versions of x to read.

The cost of maintaining multiple versions is storage space but still many recovery techniques such as Deferred Update and Immediate Update requires some before image (BFIM) information, at least of those data items that have been updated by active transactions. BFIM of a data item corresponds to its list of old versions and the recovery algorithm makes use of same in case any active transactions aborts. It is a small step for the TM to make those versions explicitly available to the scheduler.

The existence of multiple versions is only visible to the scheduler and TM, not to user transactions. Transactions still reference data items, such as x, y, etc. Users see as if there is only one version of each data item, namely, the last one that was written from that user's perspective. When the scheduler decides to assign a particular version of x to Read(x), the value returned may be one produced by an active or committed transaction. If the version read is one produced by an active transaction, recoverability imposes one constraint. The constraint is that no transaction T in Schedule S commits until all transactions that have written an item (version of data item) that T reads have committed. In other words, reading transaction's commitment would be delayed until the transaction that produced the version has committed.

In MVCC, multiversion (MV) schedules represent the TM's execution of operations on a multiversion database, and single version (1V) schedules represent the interpretation of MV schedules in the users' single version view of the database. Serial 1V schedules are correct. But the system actually produces MV schedules. So, to prove that a MVCC is correct, we must prove that each of the MV schedules that it can produce is equivalent to a serial 1V schedule. For each data item x, the versions of x are designated by x_i, x_j, . . . , where the subscript denote the index of the transaction that wrote the version. Thus, each write in an MV history is always of the form $w_i[x_i]$, where the version subscript equals the transaction subscript. Reads are

denoted in the usual way, such as $r_i[x_j]$. Now, let us consider a sample MV schedule

$S_4 = w_1[x_1]c_1\ w_2[x_2]\ c_2\ r_3[x_1]w_3[y_3]c_3$

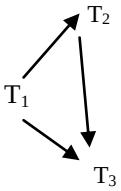
Now, according to the definition of conflicts, the operations that conflict are $w_1[x_1]$and $r_3[x_1]$. But $w_2[x_2]$ doesn't conflict with either $w_1[x_1]$ and $r_3[x_1]$ because x_2 and x_1 are different data items i.e. they are different versions of x. So, they don't conflict. Now, if we map S_4 into an equivalent 1V schedule, we get

$S_5= w_1[x]c_1w_2[x]c_2r_3[x]\ w_3\ [y]c_3$

The conflicting operations, namely $w_1[x_1]$and $r_3[x_1]$in S_4 occur in the same order in S_5 also. But in S4, $w_2[x_2]$ doesn't conflict with $r_3[x_1]$but their corresponding 1V operations, namely $w_2[x]$ and $r_3[x]$ do conflict. So, here applying the conflict equivalence is not suitable. Moreover, in S_4, T_3 reads x from T_1, whereas in S_5, T_3 reads x from T_2. Since T_3 reads a different value of x in S_4 and S_5, it may write a different value in y. So, S_4 is not view equivalent to S_5. Let us denote the serialization graph of a schedule S, by SG(S). Now, to prove that every MV schedule that MVCC can generate is equivalent to a serial 1V schedule, We will prove that SG of an MV schedule is acyclic, thereby making it serial. Then we will check that every serial MV schedule is equivalent to serial 1V schedule or not. Now, consider an MV schedule

$S_6 =w_1[x_1]w_1[y_1]c_1r_2[x_1]r_2[y_1]w_2[x_2]w_2[y_2]c_2r_3[x_1]\ r_3[y_2]c_3$

$SG(S_6)$ =



An edge from T_i-> T_j indicates that T_j reads some data item X from T_i. These edges are called as **Reads from Edges**. Now S_6 is serial and the corresponding 1V schedule.

$S_7 = w_1[x]w_1[y]c_1r_2[x]r_2[y]w_2[x]w_2\ [y]c_2r_3[x]\ r_3[y]c_2$

S_6 is not equivalent to S_7 because in S_6 ,T_3 reads x from T_1, whereas in S_7, T_3 reads x from T_2.Therefore,not all serial MV schedules are equivalent to serial 1V schedules.

Only a subset of serial MV schedules, called l-serial MV schedules, are equivalent to serial 1V schedules. So, in order to prove that MVCC is correct, we must prove that its MV schedules are equivalent to serial 1V schedules.

For this, we will use a modified version of serialisation graph called MultiVersion Serialisation Graph (MVSG). An MV schedule is equivalent to a l-serial MV schedule iff (if and only if) it has an acyclic MVSG.

3. MultiVersion Serialisation Graph (MVSG)

We know that two 1V schedules over the same transactions are view equivalent if they contain the same operations, have the same reads-from relationships and the same final writes. The same definition is applicable for MV schedules except the term final writes can be omitted because if two schedules are over the same transactions then they have the same writes. Since no versions are overwritten, all writes are effectively final writes. Thus, if two MV schedules over the same transactions have the same operations and the same reads-from relationships, then they have the same final writes and are therefore view equivalent. Here the meaning of reads from relationship is slightly different. Transaction T_j reads x from T_i, in MV schedule S if T_j, reads the version of x produced by T_i. Next, we will see the equivalence of an MV schedule S_{Mv} to a 1V schedule S_{1v}. S_{Mv} and S_{1v} must be over the same set of transactions and their operations must be in one-to-one correspondence. That is,the mapping of S_{Mv} to S_{1v} is defined by mapping c_i to c_i, a_i to a_i, $r_i[x]$ to $r_i[x_j]$ for some version x_j of x and $w_i[x]$ to $w_i[x_i]$. Since the operations of S_{Mv} and S_{1v} are in one-to-one correspondence, their reads-from relationships would be the same. All of the final writes in S_{1v} must be part of the state produced by S_{Mv}, because S_{Mv} has all versions written in it. So, just like MV schedules, an MV schedule and 1V S_{Mv} are equivalent if they have the same reads-from relationships.

[2]Two operations in an MV schedule are said to be conflict if they are from different transaction ,operate on the same version and one is a Write. Only one form of conflict is possible in an MV schedule: $w_i[x_i]$ and $r_j[x_i]$ conflict provided the former precedes the latter .The other way is not possible because T_j cannot read x_i until it has been produced. Conflicts of the form $w_i[x_i] < w_j[x_i]$ are impossible, because each write produces a unique new version. Thus, all conflicts in an MV schedule correspond to reads-from relationships.

But since only one kind of conflict is possible in an MV history, SGs are simple to draw. Let S be an MV schedule. SG(S) has nodes for the committed projection C(S),which includes only the operations in S that belong to the committed transactions and also edge T_i-> T_j is present iff for some X, $r_j[x_i]$ (i # j) is an operation of C(S) A serial MV schedule S is 1-serial if for all i,j, and x, if T_i reads x from T_j then i = j, or T_j is the last transaction preceding T_i, that writes into any version of x.In other words, a serial MV history is 1-serial if for each reads-from relationship, say T_i reads x from T_j, Tj is the last transaction preceding T_i, that writes any version of x. Schedule S_6 is not l-serial because $w_0[x_0]< w_1[x_1] < r_2[x_0]$. Here T_2 should have read x from T_1 because it has written x last. Consider another sample schedule:

$S_8 = w_1[x_1]\ w_1[y_1]\ w_1[z_1]\ c_1\ r_2[x_1]\ w_2[y_2]c_2\ r_3[x_1]\ r_3[z_1]\ w_3[x_3]c_3\ r_4[z_1]\ w_4[y_4]\ w_4[z_4]\ c_4\ r_5[x_3]\ r_5[y_4]\ r_5[z_4]$. The schedule S_8 is 1-serial.

[6]An MV schedule is 1- Serializable (or 1SR) if its committed projection is equivalent to a l-serial MV schedule. A serial schedule can be 1SR even though it is not 1-serial.

Eg: $S_9 = w_1[x_1]c_1\ r_2[x_1]\ w_2[x_2]c_2\ r_3[x_1]\ c_3$.

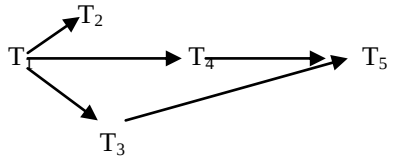
is not l-serial because $w_1[x_1]< w_2[x_2] < r_3[x_1]$.But it is lSR, because it is equivalent to

$S_{10} = w_1[x_1]c_1\ r_3[x_1]\ c_3\ r_2[x_1]\ w_2[x_2]c_2$

To determine if a MVCC is correct, we must determine if all of its schedules are 1SR. MVCC algorithms sort the versions of each data item into a total order. MVSG uses this total order of versions to include the edges.These edges are called as **Version Order Edges(VOE).[6]**

Given an MV schedule S and a data item x, a version order denoted by << for any x in S is a total order of versions of any x in H. A version order for H is the union of the version orders for all data items. Eg; For S_8, a version order is $x_1 << x_2$,, $y_1 << y_2 << y_4$, and $z_1 << z_4$.

SG(S_8) =

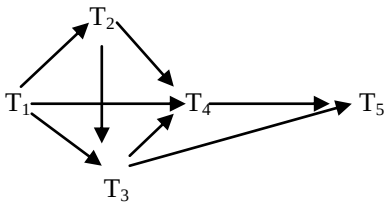


Explanation for SG(S_8):

Edge(ROE)	Comments
T_1 ->T_2	$r_2[x_1]$ because T_2 reads x from T_1
T_1 ->T_3	$r_3[x_1]$ because T_3 reads x from T_1
T_1 ->T_4	$r_4[z_1]$ because T_4 reads z from T_1
T_3->T_5	$r_5[x_3]$ because T_5 reads x from T_3
T_4->T_5	$r_5[z_4]$ because T_5 reads z from T_4

MVSG(S) is SG(S) with the inclusion of version order edges for any serial MV schedule S.MVSG(S) should be constructed for SG(S),provided SG(S) is acyclic. If $w_i[x_i]$ and $r_k[x_j]$ are in C(S), then the version order edge forces $w_i[x_i]$ to either precede $w_j[x_j]$ or follow $r_k[x_j]$ in S.More formally, for each $r_k[x_j]$ and $w_i[x_i]$ in C(S) where i, j, and k are distinct, if $x_i << x_j$, then include T_i-> T_j otherwise include T_k-> T_i as version order edge. The purpose of adding version order edges is to prevent S's reads-from relationships from change when version operations are mapped into data item operations.

MVSG(S_8)=



Explanation for MVSG(S_8):

VOE	Comments	(i,j,k)
T_2 ->T_3	$r_2[x_1]$ and $w_3[x_3]$ since $x_1 << x_3$	(3,1,2)
T_3 ->T_4	$r_3[z_1]$ and $w_4[z_4]$ since $z_1 << z_4$	(4,1,3)
T_2 ->T_4	$w_2[y_2]$ and $r_5[y_4]$ since $y_2 << y_4$	(2,4,5)

Now, we will explore two commonly used MVCC algorithm in practice, namely

i) MultiVersion Timestamp Ordering(MVTO) and

ii) MultiVersion 2PL(MV2PL)

4.MultiVersion Timestamp Ordering (MVTO)

Timestamp is a unique identifier created by the DBMS (DataBase Management System) to identify a transaction. Timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the transaction start time, denoted by TS(T), where T is the transaction. Let $TS(T_i)=i$ for all transaction i. In MVTO, the value of version x_i of each data item and the following two timestamps are maintained.[4]

(i)RTS (x_i)- The read timestamp of x_i is the largest of all the timestamps of all transaction s that have successfully read x_i.

(ii)WTS (x_i)-The write timestamp of x_i is the timestamp of the transaction that wrote the value of x_i.

A MVTO scheduler processes operations first-come first-served. It translates operations on data items into operations on versions to give an illusion as if it has processed these operations in timestamp order on a single version database. When the transaction, say i, issues a read operation on data item x, the scheduler processes $r_i[x]$ by first translating it into $r_i[x_k]$, where x_k is the version of x with the largest timestamp less than or equal to $TS(T_i)$, and then sending $r_i[x_k]$ to the TM. Otherwise ,it rejects $r_i[x]$. If $TS(T_i) > RTS(x_k)$ then set $RTS(x_k) = TS(T_i)$. Eg: If $r_3(x)$ comes for processing and x1 is the version with the largest timestamp, then the scheduler translates into $r_3(x_1)$ and if $TS(T_3) > RTS(x_1)$,then set $RTS(x_1)= TS(T_3)$

It processes $w_i[x]$ by considering two cases. If it has already processed a Read $r_j[x_k]$ such that $TS(T_k) < TS(T_i) < TS(T_j)$, then it rejects $w_i[x]$ and the transaction T_i is aborted and rollbacked. Otherwise, it translates $w_i[x]$ into $w_i[x_i]$,sends it to the TM and $WTS(x_i) = RTS(x_i)=TS(T_j)$. However, to ensure recoverability, T_i is not allowed to commit until after all the transactions that have written some versions that T_i has read have committed. Since MVTO need not process operations in timestamp order, a write could arrive whose processing would invalidate a Read that the scheduler already processed.

Eg: Suppose in S_{11}:$w_1[x_1] < r_3[x_1]$ represents the status of execution of MVTO scheduler. Now suppose if a write $w_2[x]$ comes and if the scheduler translates into $w_2[x_2]$, then it produces a schedule that no longer has the same effect as the operations that are executed in timestamp order on a single version database. The reason is in S_{11}, T_3 reads x_1,but it should have read the value written by T_2,namely x_2.So,the scheduler rejects $w_2[x]$ it has already processed $r_3[x_1]$ such that $TS(T_k) < TS(T_i) < TS(T_j)$ where i=2,j=3,k=1 in this example.

Now, if the read and write operations are processed in the above manner, all the schedules that are generated by MVTO is 1SR.To prove this, we must prove that MVSG is acyclic. For every edge T_i-> T_j in MVSG, if $TS(T_i) < TS(T_j)$ then MVSG is acyclic. This must be true for every VOE also. Version order is $x_i<x_j$ iff $TS(T_i) < TS(T_j)$.If T_j reads x from T_i, then $TS(T_i) < TS(T_j)$ because read operation is processed only if x_i timestamp is less than or equal to T_j. Let S be a schedule over $\{T_1, \ldots T_n\}$ produced by MVTO.Assume there exists $r_k[x_j]$ and $w_i[x_i]$ operations in schedule S where i, j, and k are distinct, where i, j, and k are distinct, and there are two possible way of generating version order edges.

(i) if $x_i << x_j$, then it implies T_i-> T_j is in MVSG(S) and since $x_i << x_j$, then $TS(T_i) < TS(T_j)$ which is needed to show that MVSG is acyclic.

(ii) if $x_j << x_i$, which implies either T_i-> T_j or T_k-> T_j.The former is not possible because the version order $x_j << x_i$ implies $TS(T_j) <TS(T_i)$.So, $TS(T_k) < TS(T_i)$ which is needed to show that

MVSG is acyclic. Hence all the schedules by MVTO is 1SR.So,they are equivalent to serial 1V history.

4.1 MVTO in Real-Time databases

[3][7]Unlike MV2PL, it doesn't suffer from deadlock. The problem with timestamp ordering is that there is no apparent way to bound the number of times that a transaction can be aborted. In the above example, in S_{11},if T_2 has higher priority than T_3 ,then $w_2[x]$ cannot be rejected. On the contrary, Transaction T_3 has to be aborted. Suppose transaction T_j issues an operation on data item x. ,which conflicts with an operation issued by transaction Ti . If $TS(T_i) < TS(T_j)$ does not hold, T_j must be aborted and restarted later with a larger timestamp.

Eg;S_{12}: $w_1[x_1]c_1$ $w_3[x_3]w_2[x_2]$ c_2 $r_3[x_1]$

T_3 will be aborted because the value of x_1 is overwritten. If the MVTO maintains several versions in memory, late read $r_3[x_1]$ won't be rejected. Normally version will be deleted from the oldest to newest in the database.

5.MultiVersionTwo-phase Locking (MV2PL)

A transaction is said to satisfy the two-phase locking (2PL) protocol if all of its locking operations precede the first unlock operation A transaction satisfying this protocol consists of 2 phases - the expanding phase during which new locks are acquired but no locks can be released and the shrinking phase during which locks are released but no new lock can be acquired.

In this locking scheme[4], Scheduler uses three types of locks, namely, read ,write and certify locks. In the standard locking scheme with only read and write locks, a write lock is an exclusive lock. That is, once a transaction obtains a write lock on an item, no other transactions can access that item. But, this is not so in MV2PL. The main difference between multiversion and lock models is that in MVCC, locks acquired for querying (reading) data don't conflict with locks acquired for writing data and so reading never blocks writing and writing never blocks reading. MV2PL allows other transactions to read an item x while a single transaction holds a write lock on x. This is accomplished by maintaining two versions for each item x. One version was written by a committed transaction and the other versions created by a transaction T that acquires a write lock on the item x but not yet committed. Other transactions can use the committed version of x while T holds the write lock. T can write the value of x but without affecting the committed version of x that the other transactions uses. This form of MV2PL is called 2V2PL because it uses two versions. But ,when T wants to commit, it must obtain a certify lock on all item that it currently holds write locks on before it can commit. The certify lock is exclusive lock. That is, the transaction cannot commit until all its write locked items are released by any reading transactions. Therefore, certify lock is not compatible with read locks and this is illustrated in the lock compatibility table.

Lock Compatibility Table

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

In 2V2PL,multiple reads can happen concurrently with single writes .But this comes at the expense of transaction's delay in commitment until it obtains exclusive certify locks on all the items that it holds write locks. But this avoids cascading aborts, since transactions are allowed to ∩□read the committed version of item x. Using more than two versions is called MV2PL.If the write locks don't conflict, then any number of uncertified versions may be used. It is possible for a transaction here to read any number of uncertified versions but it cannot be certified until all of the versions(uncertified) that it has read are certified. But, this may result in cascading aborts. Certify lock can be granted only if there are no read lock on the certified versions.

[6] We will now prove that all the schedules that are generated by 2V2PL is 1SR. Let S be a schedule over $\{T_1, \ldots T_n\}$ produced by 2V2PL. Let CEi denote the certification of transaction i. Version order is $x_i<x_j$ iff $CE_i <CE_j$...We can show that MVSG is acyclic if there exists an edge T_i-> T_j in MVSG, then $CE_i <CE_j$. This must be true for every VOE also. Let T_i-> T_j be in SG(S) Since certification is done after the read and write operation and before committing, $r_j[x_i]< CE_j$. Since every read operation reads the certified version (Even in MV2PL,even if transaction reads a uncertified version, it cannot be certified until the version that it has read is certified),so $CE_i < r_j[x_i]$.Therefore, $CE_i <CE_j$ and hence MVSG is acyclic. Assume there exists $r_k[x_j]$,$w_i[x_i]$ and $w_j[x_j]$ operations in schedule S where i, j, and k are distinct, where i, j, and k are distinct, and there are two possible way of generating version order edges.

(i) if $x_i << x_j$, then it implies T_i-> T_j is in MVSG(S) and since $x_i << x_j$,then $CE_i <CE_j$ which is needed to show that MVSG is acyclic.

(ii) if $x_j << x_i$, then the version order edge is T_k-> T_i. Transaction Ti that writes x must obtain a certify lock on X. For each transaction T_k that reads x, either T_i must delay its certification until T_k has been certified (if it has not already been so), or else T_k must wait for T_i to be certified before it can set its read lock on x.The former is not possible because it implies $CE_i <CE_j$ and $x_j << x_i$ implies $CE_j < CE_i$ which is a contradiction. Hence the latter leads to the conclusion $CE_k < CE_i$ and MVSG is acyclic. Hence all the schedules by 2V2PL is 1SR.So,they are equivalent to serial 1V history.

5.1 MV2PL in Real-Time databases

[3][5][9]Though MV2PL allows multiple writes, it may have priority inversion problem when the scheduler resolves data conflicts by blocking a transaction which requests a conflicting operation. Let CE_i (x) denotes the certify operation on x of transaction i. Assume that transaction T_2 has a higher priority than T_1.

Eg: $S_{13 =}$ $w_1(y)r_2(y)w_2(x)r_1(x)CE_2(x)CE_1(y)$

When the scheduler encounters $CE_2(x)$,it cannot be certified,T_2 is blocked because x is held by T_1.This is priority inversion because higher priority transaction T_2 is blocked by lower priority transaction T_1. [8]When T_1 requests $CE_1(y)$,it must wait for T_2 to release y .This results in circular wait and hence it results in deadlock. An attempt to resolve this is to abort the lower priority transaction, namely T_1 here. Deadlock prevention schemes such as Wait-die /wound-wait scheme can be used.

6. CONCLUSION

This paper uses serialisation graphs to analyse the correctness of MVCC algorithms in real-time databases. Both MVTO and MV2PL algorithm produces a serialisable schedule. MVTO algorithm doesn't incur deadlock and it can decrease the violation of time constraints due to resource waiting. The MVTO algorithm discussed here also proposes when the lower priority transaction has to be aborted. Generally transaction aborts prevents deadlocks, which could degrade the performance of real-time database systems. But it will result in wastage of system resources.So, aborts should be done whenever necessary.

In 2V2PL / MV2PL,priority inversion could result in deadlock. But this can be prevented by aborting the lower priority transaction. But this should be done when the higher priority transaction wants to acquire certify lock to commit. But supposing if a higher priority transaction T_2 requests a lock that conflicts with the lower priority transaction T_1,then T_2 can be made to wait for T_1 to release the lock. Hence the balance can be striked between the urgent demands of higher priority transactions and unnecessary aborts of the lower priority transaction.

REFERENCES

[1] Seok Hee Hong,Myoung Ho Kim, "A Real-Time Concurrency control algorithm:Use of Multiversion and Precedence Relationship," takenfrom *csd.ks.ac.kr/~shhong/sources/jsa.ps.gz*

[2] P.A. Bernstein and N. Goodman, "Multiversion Concurrency Control-Theory and Algorithms,"*ACM Trans. Database Systems*, vol. 8, no. 4, pp. 465-483, Dec. 1983.

[3] R.Abbott, "Scheduling Real-Time Transactions:A performance Evaluation," *ACM Trans. Database Systems,Vol.17,no.3* , pp. 513-560, Sep.1992

[4] Elmasri,Navathe, "Fundamentals Of Database Systems," 3rd edition,pp.629-678,Addison –Wesley,1997

[5] P. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems," Addison-Wesley, 1987.

[6] "Multiversion concurrency control,"chapter 5 taken from research.microsoft.com/en-us/people/philbe/chapter5.pdf

[7] S.H.Son, "Advances in Real-Time Systems,"PHI,1995

[8] Quilong Han,Haiwei pan, "A Concurrency Control Algorithm Access to Temporal Data in Real-Time Database Systems," imsccs, pp.168-171, 2008 International Multi-symposiums on Computer and Computational Sciences, 2008

[9] Michael J. Carey and Waleed A. Muhanna. "The performance of multiversion concurrency control algorithms," ACM Transactions on Computer Systems, 4(4):338{378, November 1986