Real-Time Multiversion Concurrency Control Using Precedence Relationship*

Seok Hee Hong    Yoon-Joon Lee    Myoung Ho Kim†

Dept. of Computer Science
Korea Advanced Institute of Science and Technology
373-1, Kusung-dong, Yusung-gu, Taejon, 305-701, Korea

Abstract

Transactions in a real-time database system (RTDBS) attempt to satisfy timing constraints associated with those criticalness. Concurrency control in real-time database systems must not only enforce data consistency, but also satisfy timing constraints of those transactions. In this paper, we propose a priority-based multiversion 2PL algorithm which adjusts a serialization order between the conflicting transactions by maintaining precedence relationship. It increases concurrency among transactions by allowing multiple versions on data. In the algorithm, precedence relationship, which implies a serialization order among the conflicting transactions, is used to resolve data conflicts. Our proposed algorithm resolves data conflicts in favor of higher priority transactions, and selectively aborts a lower priority transaction when it conflicts with the higher priority transaction.

1 Introduction

Real-time database systems (RTDBS) have transactions that should be executed with timing constraints. In general, a timing constraint is expressed in the form of a deadline until which the user wants to complete the transaction[7, 11, 12, 13]. Conventional database systems differ from real-time database systems in that they do not take into account timing constraints. Applications of real-time database systems, such as computer aided manufacturing, military command and control, traffic control and process control, are being increasingly expanded.

It is non-trivial that techniques of conventional database systems are adapted to real-time database systems. In conventional database systems, the performance objective is to minimize average response time and maximize the throughput. The primary performance goal in real-time database systems, on the other hand, is to maximize the number of transactions that satisfy their deadlines[11, 12]. In contrast to a conventional database system whose transactions are executed with fair opportunities, transactions in a real-time database system should not be executed under the same condition. This is because the transaction must satisfy its own deadline. A priority comprises information such as the absolute deadline, criticalness, the elapsed time, the slack time, etc. of the transaction. Transaction manager for real-time databases schedule all the transactions based on a priority assigned to each transaction. Consequently, conventional scheduling techniques, where all transactions are fairly scheduled in general, are not appropriate for real-time transaction processing since conventional transactions are fairly scheduled.

Concurrency control schemes are required for the transaction scheduling in order to maintain data consistency. Many previous works on a real-time concurrency control are based on conventional concurrency control schemes such as two-phase locking (2PL) and optimistic strategy. When a transaction performs the conflicting operations on data objects shared among active transactions, the transaction manager resolves the conflicting operations by controlling the concurrent execution of the transactions. There are two approaches to resolve data contention among the conflicting transactions: blocking and roll-backs of those transactions. A conventional locking mechanism blocks the transactions that issue a conflicting operation. In contrast to a locking mechanism, the optimistic strategy aborts the transactions that conflict with a transaction having a smaller timestamp. These conflict resolution approaches can be extended for real-time concurrency control.

A locking protocol, which is widely-used and well-performed mechanism under conventional database environment, has been intensively studied for a real-time database system. 2PL-HP[1, 2, 3] is the real-time locking protocol that favors the higher priority transactions to resolve data conflict. It, however, has the problem of requiring a prior knowledge of transactions. Recent

*This paper was supported in part by NON DIRECTED RESEARCH FUND, Korea Research Foundation.
†To whom correspondences should be addressed (e-mail: mhkim@dbserver.kaist.ac.kr)

Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995
© World Scientific Publishing Co. Pte Ltd
study[5, 6] showed that the optimistic strategy can outperform 2PL under firm real-time database environment by ensuring that eventually discarded transactions due to missing their deadlines do not abort other active transactions. In [9], they focused on the effect of unnecessary restarts, which degrade the performance of optimistic algorithms. Their optimistic algorithm was designed to avoid unnecessary restarts by dynamically adjusting serialization order among concurrent active transactions. The study showed that optimistic algorithms including those in [1, 3] outperform 2PL-HP by discarding tardy transactions from the system. However, optimistic algorithms as in 2PL-HP require information of running transactions such as the read/write data set and the timestamp interval. There has been a concurrency control algorithm that combines the optimistic strategy and the locking protocol to increase the degree of concurrency[10]. In this algorithm, the higher priority transaction is never blocked by the uncommitted lower priority transaction, while the lower priority transaction may not have to be aborted even in case of conflicting with the higher priority transactions.

Multiversion concurrency control algorithms have been tailored to real-time concurrency control by using priority of transactions. In [8], they developed real-time multiversion 2PL algorithms that dynamically adjust the serialization order without requiring a prior knowledge of running transactions. The benefit of allowing multiple versions on data is that since write operation produces a new data version, there is no data conflict among the write operations. As a result, the multiversion 2PL scheduler can maintain the serialization order among the conflicting transactions with more freedom than the single version scheduler. However, this algorithm may suffer from performance degradation by delaying abortion of the transactions which will be eventually aborted. In particular, GMV2PL[8] resolves data conflict between read and write operation in the end of the transaction execution. Since the scheduler delays abortion of the transaction which will eventually be aborted, it wastes resources of the system until the transaction is aborted.

In this paper, we propose a multiversion 2PL algorithm which dynamically adjusts the serialization order between conflicting transactions by maintaining precedence relationship of each transaction. Since the transaction scheduler updates precedence relationship of a transaction whenever it issues a conflicting operation, the scheduler can resolve data conflicts by using enough information about the conflicting transactions. In addition, maintaining multiple versions on a data not only increases the degree of concurrency, but also makes it easy to adjust the serialization order among the conflicting transactions.

The remainder of this paper is organized as follows. In Section 2 we present a multiversion 2PL algorithm that adjusts the serialization order among conflicting transactions and describe how to decide the serialization order among the conflicting transactions. Section 3 shows that our proposed algorithm is correct and free from deadlock. Finally, section 4 presents concluding remarks.

2 Real-Time Multiversion Concurrency Control

In conventional multiversion 2PL(MV2PL), the scheduler can increase the degree of transaction concurrency by allowing transactions to perform multiple write operations on a data object. However, there are many problems in introducing MV2PL into real-time concurrency control. Since the MV2PL scheduler resolves data conflicts by blocking a transaction which requests a conflicting operation, it suffers from priority inversion problem. Priority inversion occurs when a high priority transaction is blocked by a low priority transaction, which may result in making a high priority transaction miss its deadline. In real-time database systems, it is important for concurrency control algorithms to minimize priority inversion. In what follows, we present problems of MV2PL in real-time database environment, and propose a new multiversion 2PL algorithm, which takes into account priority of conflicting transactions and enables to adjust the serialization order dynamically.

2.1 Basic Concept

There are three kinds of locks in MV2PL: read locks, write locks and certify locks. Since a write operation on a data object produces a new data version on a, concurrent write operations on the same data object do not lead to data conflicts. Also, since read and write locks are compatible each other, a data object can be held by read and write locks at the same time. In a transaction’s commit phase, the scheduler converts all of the transaction’s write locks into certify locks. By enforcing only one transaction to certify a data version, the scheduler ensures that a transaction reads the most recent version produced by a committed transaction. Such data version is referred to as certified version. In MV2PL, data conflicts among the transactions are resolved by blocking conflicting transactions. As we mentioned, blocking transactions may lead to not only deadlock, but also priority inversion problem. The following example shows such problems.

Example 1 : Suppose the scheduler resolves data conflicts by using MV2PL. Let $l_i(x)$ denote lock $l$ on a data object $x$ by a transaction $i$, where $l$ can be a read(R), write(W) or certify(C) lock. Suppose that transactions $T_1$ and $T_2$, where $T_2$ has a higher priority than $T_1$, and their concurrent execution history $H$ are as follows.

$$H = W_1[b]; R_1[a]; C_1[b], \quad T_2 = R_2[b]; W_2[a]; C_2[a]$$

In history $H$, when $T_2$ requests $C_2[a]$, it is blocked due to $R_1[a]$ held by $T_1$. Blocking $T_2$ leads to priority inversion problem since a high priority transaction must wait for a low priority transaction. In addition, when $T_1$ requests $C_1[b]$, it must wait for $T_2$ to release $R_2[b]$. Such waiting also produces deadlock situation where neither of transactions $T_1$ and $T_2$ can proceed. $\Box$

In the previous example, suppose that the scheduler resolves data conflicts according to priorities of conflicting transactions. When $T_2$ requests $C_2[a]$, the scheduler aborts
2.2 Serialization Ordering By Precedence Relationships

In our proposed algorithm, we assume that a transaction has a unique priority. A transaction also has two variables Before_Set and After_Set which imply precedence relationships of the transaction, that is, the serialization order. Precedence relationships Before_Set and After_Set play important roles of synchronizing conflicting transactions. Before_Set and After_Set of a transaction contain a set of transactions which immediately precede and succeed the transaction in the serialization order, respectively. We use Before_Set*[T_i](After_Set*[T_i]) to denote Before_Set(After_Set) of a transaction T_i. In addition, we define the following notations to represent precedence relationships among transactions.

Before_Set*[T_i] := Before_Set[T_i] ∪
{t | Before_Set*[k], k ∈ Before_Set[T_i]}
After_Set*[T_i] := After_Set[T_i] ∪
{t | After_Set*[k], k ∈ After_Set[T_i]}

In the above, Before_Set*[T_i] (resp. After_Set*[T_i]) means a set of transactions which precede (resp. succeed) a transaction T_i in the serialization order. For example, suppose that Before_Set[T_1] = (T_2,T_3), Before_Set[T_2] = (T_4,T_3) and Before_Set[T_3] = (T_1). By definition, Before_Set*[T_1] = (T_2,T_3,T_4,T_5,T_6). If both Before_Set*[T_1] and After_Set*[T_1] contain a transaction T_3, then the serialization order has a cycle consisting of T_1 and T_3, which results in deadlock. Our algorithm resolves data conflicts so that a transaction cannot exist in both Before_Set* and After_Set*.

Since the proposed algorithm is based on a conventional multiversion 2PL[4], a transaction can perform three operations on a data object as in MV2PL: read, write and certify. Before a transaction executes an operation on a data object, it must request and then hold the corresponding lock. The scheduler performs the following steps when a transaction t requests a lock l on a data object x:

\[ \text{LOCK}(t, l) \]

1. If t' holds a lock l on x, then
   1.1 Update precedence relationships of t and t', if needed
   1.2 Resolve conflict between t and t'
      using the compatibility matrix
2. Otherwise, grant a lock l to t

We discuss how the scheduler updates Before_Set[t] and After_Set[t] in the following paragraph. Step 1.2 states that the scheduler refers to the compatibility matrix in resolving data conflict between transactions t and t'. We will in depth discuss the compatibility matrices later.

The scheduler maintains Before_Set and After_Set on each transaction, which imply the serialization order among the conflicting transactions. If a transaction t requests a lock l held by other transaction t' and there are no precedence relationships between t and t', then the scheduler may have to update precedence relationships on t and t'. To discuss how and when the scheduler updates precedence relationships, we use the following notations. T_0 and T_1 denote a lock owner and a lock requester, respectively. We also use \( R_{T_0}[x] \vdash W_{T_0}[x] \) to represent that T_0 holds a read lock on a data object x and subsequently, T_0 requests a write lock on x. T_0 and T_1 are used to denote a high and a low priority transaction, respectively. The following two cases describe how the scheduler updates Before_Set and After_Set on T_0 and T_1, when there is no precedence relationship between T_0 and T_1.

1. \( R_{T_0}[x] \vdash W_{T_0}[x] \) : Since T_0 has already read a certified version on x before the certify operation of T_1, the write operation of T_1 has no effect on T_0. It means that T_0 precedes T_1 in the serialization order. Therefore, the scheduler inserts T_0 into Before_Set[T_1] regardless of priorities of T_0 and T_1. That is, the scheduler performs Before_Set[T_1] := Before_Set[T_1] ∪ \{T_0\} and After_Set[T_1] := After_Set[T_1] ∪ \{T_0\}.

2. \( W_{T_0}[x] \vdash R_{T_1}[x] \) : Although T_0 executed the write operation before the read operation of T_1, the write operation of T_0 does not affect T_1. Since it leads to the serialization order where T_0 precedes T_1, the scheduler makes sure that T_1 certifies a new version of x after T_0 terminates. Therefore, the scheduler performs Before_Set[T_1] := Before_Set[T_1] ∪ \{T_1\} and After_Set[T_1] := After_Set[T_1] ∪ \{T_1\}.

3. \( C_{T_1}[x] \vdash R_{T_0}[x] \) : As we mentioned, a transaction can request a certify lock in only the commit phase.
A read or a write lock must be also requested before the commit phase. Therefore, $T_o$ is in the commit, i.e., close to termination, but $T_r$ is not. In this case, the scheduler makes $T_r$ wait for $T_o$ to commit. Even though $T_r$ has a higher priority than $T_o$, since $T_r$ will resume its work sooner, the system does not suffer from serious performance degradation. We will discuss in the next section that the scheduler can avoid unnecessary abortions of lower priority transactions at a cost of momentary priority inversion. Updating precedence relationships, the scheduler performs $Before_Set[T_r] := Before_Set[T_r] \cup \{T_o\}$ and $After_Set[T_o] := After_Set[T_o] \cup \{T_r\}$ in a way that $T_o$ precedes $T_r$ in the serialization order.

As shown in the above cases, note that even though the scheduler prefers a high priority transaction in resolving data conflicts, a high priority transaction does not always precede lower priority transactions in the serialization order.

### 2.3 Multiversion 2PL Algorithm By Precedence Relationship

In our algorithm, the scheduler controls the serialization order among conflicting transactions by using precedence relationships such as $Before_Set$ and $After_Set$. Once the scheduler updates precedence relationship among conflicting transactions, it refers to the compatibility matrix to resolve data conflicts. The compatibility matrix contains actions to synchronize conflicting transactions according to their priorities. We first describe the commit phase of a transaction before discussing the compatibility matrix. The following shows the steps in a transaction $T$'s commit phase.

\[
\text{COMMIT}(T)
\]

1. Convert all of its write locks into certify locks
2. Block $T$ until $Before_Set^*[T]$ is empty
3. Begin a Critical Section
4. Convert all of its read locks into certify locks
5. Convert all of its write locks into certify locks
6. Commit the new data versions
7. Release all of its read locks
8. Release all of its write locks
9. End a Critical Section

The step 2 states that no transaction can read or certify a data version until there are no transactions which precede $T$ in the serialization order. This guarantees that the scheduler produces a schedule whose history is strict and that avoids cascading aborts. Steps 3, 4 and 5 are performed in a critical section such that no transaction reads a data version certified by $T$ until its termination. Step 3 states that the scheduler certifies all data versions produced by $T$. In the final step, the scheduler updates $Before_Set$ of transactions in $After_Set^*[T]$.

In our algorithm, the scheduler may allow priority inversion when a high priority transaction requests a lock held by a low priority transaction that is in the commit phase. Consider that $T_H$ requests a lock held by $T_L$. Suppose also that $T_L$ is in the commit phase but $T_H$ is not. If the scheduler aborts $T_L$ and grants the lock to $T_H$, then it can avoid priority inversion at the expense of wasting work done by $T_L$ that is close to termination. In this case, it would be better to enforce $T_H$ to wait for $T_L$ to finish, i.e., allow priority inversion. Since $T_L$ is close to commit and then $T_H$ can resume its execution sooner, such priority inversion is not serious problem. Suppose now that $T_H$ is in the commit phase too. Since both $T_H$ and $T_L$ are in the commit phase, the scheduler resolves data conflicts in favor of a high priority transaction, i.e., avoiding priority inversion problem. Notice that a transaction $T$ in the commit phase always waits for transactions which have a higher priority than $T$. That is, $T$ has a lower priority than transactions in $Before_Set^*[T]$. In step 3, when a transaction $T$ performs lock conversion, the scheduler aborts lower priority transactions for which $T$ must wait. This is achieved by the compatibility matrices.

In our algorithm, there are two important factors in scheduling conflicting transactions: priorities and precedence relationships. The scheduler prefers a high priority transaction in resolving data conflicts by sacrificing lower priority transactions. That is, a high priority transaction tends to precede lower priority transactions in the serialization order. Furthermore, precedence relationships are used to avoid unnecessary abortions or blockings of transactions. There are two compatibility matrices according to which of a lock owner and a lock requester has higher priority. The conflict resolution strategy in the compatibility matrix depends on factors such as the priority, precedence relationships and whether a transaction is in the commit phase. When a lock requester has a higher priority than a lock owner, Figure 1 shows the compatibility matrix for the conflicting transactions. The compatibility matrix is divided into two compatibility matrices according to precedence relationships between a lock owner and a lock requester. The scheduler determines the serialization order by using precedence relationships of $T_H$ and $T_L$.

![Figure 1](image-url) The compatibility matrix when a high priority transaction $T_H$ requests a lock held by a transaction a low priority $T_L$.

Since there is no conflict between write locks as in MV2PL, the scheduler allows to hold multiple write locks on the same data object. In most cases, since the scheduler
resolves a data conflict in favor of a high priority transaction, \( T_H \) can be allowed to hold a lock without disturbing \( T_L \). However, there are conflicting cases where a lock owner must be aborted. Suppose that when \( T_L \) that holds a lock on a data object \( x \) precedes \( T_H \) in the serialization order (i.e., \( \text{BeforeSet}*[T_H] \) includes \( T_L \)), \( T_H \) requests a certify lock on \( x \) in the commit phase. By precedence relationships between \( T_H \) and \( T_L \), \( T_H \) must certify a data version after \( T_L \) terminates. To avoid priority inversion, the scheduler should make sure that \( T_H \) in the commit phase is never blocked by lower priority transactions. As a result, since \( T_H \) cannot wait for \( T_L \) to terminate, the scheduler enables \( T_H \) to hold a certify lock by aborting \( T_L \). In this way, a high priority transaction, which is in the commit phase, does not suffer from priority inversion problem.

As mentioned, however, the scheduler may cause priority inversion to save system resources used by lower priority transactions. This is not critical because of short blocking time of a high priority transaction. Suppose now that when \( T_L \) that holds a write or a certify lock on a data object \( x \) precedes \( T_H \) in the serialization order, \( T_L \) is in the commit phase and subsequently \( T_H \) that is not in the commit phase requests a read lock on \( x \). To enforce a strict schedule, \( T_H \) must read a data object \( x \) after \( T_L \) commits. If the scheduler depends on only priorities of transactions in resolving data conflicts, then it will simply abort \( T_L \). However, the scheduler let \( T_H \) wait for \( T_L \) instead of aborting \( T_L \). This strategy has the advantage of saving the work done by \( T_L \) at the cost of blocking \( T_H \) shortly.

Consider now the case that \( T_H \) precedes \( T_L \) in the serialization order, i.e., \( T_H \) is in \( \text{BeforeSet}*[T_L] \). We discuss first the situation where \( T_L \) is aborted. Suppose that when \( T_L \) holds a read lock on a data object \( x \), \( T_H \) requests either a write or a certify lock on \( x \). By precedence relationships between \( T_H \) and \( T_L \), \( T_L \) must read a certified version after \( T_H \) terminates. However, since \( T_L \) has already read a certified version before \( T_H \) commits, read of \( T_L \) is not consistent operation. As a result, the scheduler must abort \( T_L \). Notice that even though \( T_L \) holds a certify lock, the scheduler allows \( T_H \) to hold a certify lock on the same data object. This is different from MV2PL in that certify locks are not always incompatible each other. Even though \( T_L \) holds a certify lock, it must wait for \( T_H \) that has a higher priority than \( T_L \) (i.e., are in \( \text{BeforeSet}*[T_L] \)) in the commit phase. Therefore, \( T_L \) never perform a certify operation before \( T_H \) terminates. In this way, since the scheduler resolves data conflicts among certify locks in favor of a high priority transaction, certify operations by \( T_H \) and \( T_L \) can be synchronized.

Figure 2 shows the compatibility matrix for the conflicting transactions, when a lock requester has a lower priority than a lock owner. Consider that \( T_H \) precedes \( T_L \) in the serialization order, i.e., \( \text{BeforeSet}*[T_L] \) includes \( T_H \). When \( T_H \) holds a lock on a data object \( x \), let \( T_L \) request a certify lock on \( x \). Since the scheduler prefers higher priority transactions in resolving data conflicts, even though \( T_L \) is in the commit phase, the scheduler blocks \( T_L \) until \( T_H \) commits. Also, let \( T_H \) hold a write or a certify lock on \( x \). In case that \( T_L \) requests a read lock on \( x \), the scheduler blocks \( T_L \) until \( T_H \) performs a certification on \( x \) such that \( T_L \) can read a data object \( x \) certified by \( T_H \).

Consider now that a lock requester \( T_L \) precedes a lock owner \( T_H \) in the serialization order, i.e., \( \text{BeforeSet}*[T_H] \) includes \( T_L \). In this case, since \( T_L \) requests a lock later than \( T_H \) that follows \( T_L \) in the serialization order, the scheduler may have to abort \( T_L \). Suppose that \( T_H \) holds a read lock and subsequently reads a certified version on a data object \( x \). In this case, if \( T_L \) requests a write or a certify lock on \( x \), then since this violates the serialization order where \( T_L \) precedes \( T_H \), the scheduler must abort a low priority transaction \( T_L \). Suppose now that \( T_H \) holds a certify lock on a data object \( x \). When \( T_L \) requests a write or certify lock on the same data object, the scheduler must abort \( T_L \). However, if the scheduler grants a write or a certify lock to \( T_L \), then \( T_H \) may have to wait for \( T_L \) in the commit phase, resulting in priority inversion. Priority inversion in the commit phase leads to blocking higher priority transactions that may meet those deadlines soon.

At a result, the scheduler must make sure that priority inversion does not occur in the commit phase by aborting \( T_L \) before its commit phase. The following example shows how the proposed algorithm works.

**Example 2:** Suppose that three transactions \( T_1, T_2 \) and \( T_3 \) request locks as follows.

\[
T_1 = R_1[a]; W_1[a]; W_1[b]; C_1[a]; C_1[b]
\]
\[
T_2 = R_2[b]; W_2[a]; C_2[a]
\]
\[
T_3 = W_3[a]; W_3[b]; C_3[a]; C_3[b]
\]

\( T_1, T_2, T_3 \) have the highest and the lowest priority, respectively. Figure 3 shows a schedule produced by our proposed algorithm. For simplicity, we take into account only lock request. \( T_3 \) with the lowest priority first requests a write lock on a data object \( a \). At time \( t_1 \), since \( T_3 \) requests a read lock on a data object \( a \), the scheduler produces precedence relationship of \( \text{AfterSet}*[T_2] = \{T_2\} \) and \( \text{BeforeSet}*[T_3] = \{T_3\} \). At time \( t_2 \), by \( R_3[a] \), precedence relationship among \( T_1, T_2 \) and \( T_3 \) is updated to \( \text{AfterSet}*[T_1] = \{T_1\}, \text{AfterSet}*[T_3] = \{T_3\} \), and \( \text{BeforeSet}*[T_2] = \{T_2\} \). Also, at time \( t_5 \), a transaction \( T_2 \) requests a write lock on a data object \( a \). Then since there is no precedence relationship between \( T_1 \) and \( T_2 \), the scheduler updates precedence relationship to

![Figure 2](image-url)
3 Correctness of the Proposed Algorithm

In this section, we prove that our proposed algorithm is correct and free from deadlocks. The proof procedure is based upon one in [4]. We start by presenting some notations for the proof. \( r_i[x] \) denotes that a transaction \( T_i \) performs a read operation on a data item \( x \). Also, \( w_i[x] \) (resp. \( c_i[x] \)) denotes that a transaction \( T_i \) performs a read (resp. certify) lock on \( x \). We use \( r_i[x] \) to represent that a transaction \( T_i \) holds a read lock on a data item \( x \), and \( w_i[x] \) (resp. \( c_i[x] \)) to indicate that a transaction \( T_i \) holds a read (resp. certify) lock on \( x \). \( r_i[x] \) (resp. \( c_i[x] \)) represents that a transaction \( T_i \) releases a read (resp. certify) lock on a data item \( x \). However, since a read lock is not released but converted to a certify lock, we do not need to use \( w_i[x] \) to denote that a transaction \( T_i \) releases a read lock on \( x \). A partial order between operations. For example, if \( r_i[x] \prec w_j[x] \) is in a history, it implies that \( r_i[x] \) precedes \( w_j[x] \) in the execution order. \( C(H) \) indicates a complete history over the set of committed transactions in a history \( H \). In the following proof, we take no account of a priority of conflicting transactions. The reason for this is that the committed history contains not only the transactions with the higher priority, but also the transactions with the lower priority unless it has been aborted by the higher priority transaction. That is, a priority cannot play an important role in the committed history.

To state property of histories produced by the scheduler, we present all the orderings of operations in which the completed transactions perform on database according to our proposed algorithm. To perform an operation on database, a transaction obtains the corresponding lock, and then releases it after execution of the operation. In what follows, we states some properties of our proposed algorithm. We use these properties to prove that our proposed algorithm produces a serializable schedule. The following property formalizes that a read lock must be held before the read operation is performed.

**Property 1** : Let \( H \) be a history produced by the scheduler. If \( r_i[x] \) is in \( C(H) \), then \( r_i[x] \prec r_i[x] \prec r_i[x] \).

Also, a transaction which performs a write operation must obtain a write lock and then convert it to a certify lock before a certify operation. A write lock does not need to be explicitly released according as a write lock is converted to a certify lock. The following definition captures this.

**Property 2** : Let \( H \) be a history produced by the scheduler. If \( w_i[x] \) is in \( C(H) \), then \( w_i[x], c_i[x] \), \( c_i[x] \) and \( w_i[x] \) are in \( C(H) \) such that \( w_i[x] \prec c_i[x] \prec c_i[x] \prec w_i[x] \).

If a serialization order implies both \( T_i \) precedes \( T_j \) and \( T_j \) precedes \( T_i \), the scheduler cannot guarantee a serializable schedule due to a cycle that \( T_i \) precedes \( T_j \) and then precedes \( T_i \) in their serialization order. In case that two transactions \( T_i \) and \( T_j \) execute conflicting operations \( w_i[x] \) and \( r_j[x] \) on a data item \( x \) respectively, the data conflict has to be resolved such that either \( T_i \) precedes \( T_j \) or \( T_j \) precedes \( T_i \) in their serialization order. So, if \( T_j \) precedes \( T_i \) in their serialization order, the scheduler must ensure that \( r_i[x] \) follows \( c_i[x] \) and then \( r_j[x] \) is performed after \( c_i[x] \). Otherwise, to enforce that \( r_i[x] \) is followed by \( c_i[x] \), the scheduler must guarantee that \( T_i \) performs \( c_i[x] \) after \( r_i[x] \). The following definitions formalize these arguments.

**Property 3** : Let \( H \) be a history produced by the scheduler. If \( r_i[x] \) and \( w_j[x] \) (\( i \neq j \)) are in \( C(H) \), then \( w_i[x] \), \( c_i[x] \), \( c_i[x] \) and \( r_i[x] \) are in \( C(H) \) such that either \( w_i[x] \prec c_i[x] \prec c_i[x] \prec r_i[x] \).

A write operation must be performed before a certify operation. This is similar to saying that a transaction can certify a data version after obtaining a write lock and subsequently a certify lock. Thus, if \( T_i \) and \( T_j \) performed their write operations, either \( c_i[x] \) precedes \( c_i[x] \) or \( c_i[x] \) precedes \( c_i[x] \) according to their serialization order.

**Property 4** : Let \( H \) be a history produced by the scheduler. If \( w_i[x] \) and \( w_j[x] \) (\( i \neq j \)) are in \( C(H) \), then \( c_i[x] \) and \( c_j[x] \) are in \( C(H) \) such that either \( c_i[x] \prec c_j[x] \) or \( c_j[x] \prec c_i[x] \).
To guarantee a serializable schedule, our proposed algorithm forces a transaction to follow a 2PL rule. Let the lock interval be the time interval between locking and unlocking on a data item. If all of transaction T's lock intervals always overlap each other, we say that T follows a 2PL rule. That is, a transaction cannot hold any locks after releasing a single lock.

**Property 5**: Let H be a history produced by the scheduler. If \( p_i[x] \) and \( q_i[y] \) are in C(H), then \( pl_i[z] < qu_i[y] \).

Our proposed algorithm uses precedence relationship which plays an important role of deciding the serialization order between conflicting transactions. For instance, suppose that \( T_i \) holds a read lock on a data item \( x \) and in turn \( T_j \) requests a write lock on \( x \). The scheduler updates precedence relationship between \( T_j \) and \( T_i \) such that \( T_i \)'s read operation can precede \( T_j \)'s certify operation in a serialization order. The following lemmas formalize a precedence relationship between \( T_i \) and \( T_j \).

**Lemma 1**: Let H be a history produced by the scheduler. If \( T_i \in Before.Set*[T_j] \) (\( i \neq j \)), then for operations \( ri[x] \) and \( wj[x] \) on a data item \( x \) in history H, \( ri[x] < cl_j[x] \).

**Proof**: Since \( T_i \in Before.Set*[T_j] \), two conflicting operations \( ri[x] \) and \( cl_j[x] \) must exist in the C(H) such that \( ri[x] < cl_j[x] \). Hence, Property 1 and 2 lead to the following.

1. \( rl_i[x] < rl_i[x] < cu_i[x] \), and
2. \( wj_i[x] < wj_i[x] < cl_j[x] < cl_j[x] < cu_i[x] \)

Additionally, by Property 3, either \( cu_i[x] < cl_j[x] \) or \( cu_i[x] < wj_i[x] \). The latter derives \( cl_j[x] < cu_i[x] \), but contradicts \( rl_i[x] < cl_j[x] \). Therefore, it is proved that \( pu_i[x] < cl_j[x] \).

**Lemma 2**: Let H be a history produced by the scheduler. If \( T_i \in Before.Set*[T_j] \) (\( i \neq j \)), then for operations \( w_i[x] \) and \( r_j[x] \) on a data item \( x \) in a history H, \( cu_i[x] < rl_i[x] \).

**Proof**: The reverse of Lemma 1.

**Lemma 3**: Let H be a history produced by the scheduler. If \( T_i \in Before.Set*[T_j] \) (\( i \neq j \)), then for operations \( w_i[x] \) and \( w_j[y] \) on a data item \( x \) in a history H, \( cu_i[x] < cu_j[y] \).

**Proof**: Since \( T_i \in Before.Set*[T_j] \) (\( i \neq j \)), two certify operations \( ci[x] \) and \( cj[y] \) must exist in the C(H) such that \( ci[x] < cj[y] \). Therefore, Property 3 leads to the following.

1. \( w_i[x] < w_i[x] < cl_i[x] < cl_i[x] < cu_i[x] \)
2. \( wj_i[x] < wj_i[x] < cl_j[x] < cl_j[x] < cu_i[x] \)

Additionally, by Property 4, either \( cu_i[x] < cu_j[x] \) or \( cu_i[x] < cu_j[y] \). The latter derives \( cj[y] < ci[x] \), but contradicts \( ci[x] < cj[y] \). Hence, we have \( cu_i[x] < cu_j[x] \).

By using the serialization graph for a history H, denoted SG(H), we can prove correctness of our proposed algorithm. We define the serialization graph by a directed graph whose nodes are the committed transactions in H and whose edges are \( T_i \rightarrow T_j \) such that \( T_i \in Before.Set*[T_j] \) (\( i \neq j \)). Therefore, if \( T_i \in Before.Set*[T_j] \), \( T_j \rightarrow T_j \in Before.Set*[T_i] \) in H, then there is a path \( T_i \rightarrow T_j \rightarrow \cdots \rightarrow T_n \) in the SG(H).

**Lemma 4**: Let H be a history produced by the scheduler. If a path \( T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \) in the SG(H), then for some data items \( x \) and \( y \) in H, either in H

1. for \( r_1[x] \) and \( w_n[y] \), \( ru_1[x] < cl_n[y] \), or
2. for \( w_1[x] \) and \( r_n[y] \), \( cu_1[x] < r_1[y] \), or
3. for \( w_1[x] \) and \( w_n[y] \), \( cu_1[x] < cl_n[y] \)

**Proof**: The proof is by induction on n. Since the proof of the cases 2 and 3 is similar to that of the case 1, we will present only the proof of the case 1. The basis step, for \( n = 2 \), intuitively follows from Lemma 1. For the induction step, suppose that the lemma holds for \( n = k \) for some \( k \geq 2 \). Let us prove that the lemma holds for \( n = k + 1 \). By the induction hypothesis, it is certain that a path \( T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \) is in the SG(H), and for some data items \( x \) and \( z \), and operations \( ri[x] \) and \( w_k[y] \) are in H such that \( ru_k[x] < cl_k[y] \). By \( T_k \rightarrow T_{k+1} \), and Lemmas 1 and 2, there must be a data item \( y \), either operation \( w_k[y] \) or \( w_{k+1}[y] \), and operation \( w_{k+1}[y] \) in H, such that either \( ru_k[y] < cl_{k+1}[y] \) or \( cu_k[y] < cl_{k+1}[y] \). By Property 5, we have either \( w_{k+1}[z] < ru_k[y] \) or \( w_{k+1}[z] < cu_k[y] \). By such partial orders, \( ru_1[z] < cl_{k+1}[y] \). The proof is completed.

By the serializability theory, if a cycle is in the SG(H), the history H is not serializable. We will prove that the scheduler executed by our proposed algorithm has only acyclic SG(H) for H.

**Theorem 1**: All of the histories produced by the scheduler is serializable.

**Proof**: The proof is by the contradiction. Suppose that SG(H) for a history H produced by the scheduler contains a cycle \( T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1 \) (\( n > 1 \)). By Lemma 4, this implies that for data items \( x \) and \( y \), and some operations \( ri[x] \), \( r_j[y] \), \( w_n[x] \) and \( w_n[y] \), either \( ru_1[x] < cl_j[y] \), \( cu_1[x] < r_1[y] \) or \( cu_1[x] < cl_j[y] \), and contradicts Property 5 since any case does not follow a 2PL rule. Hence, SG(H) can't contain a cycle and H is a serializable history by the serializability theory. The proof is completed.

We finish this section by proving that our proposed algorithm is free from deadlocks. In general, deadlocks can be detected by a waits-for graph (WFG), which is a directed graph whose node is a transaction denoted \( T_i \) and edge \( T_i \rightarrow T_j \) represents that \( T_i \) is waiting for \( T_j \) to release some lock. If the WFG for a schedule contains a cycle, then
the schedule has a deadlock. Therefore, we will prove that the WFG produced by the scheduler is an acyclic graph. Property 3 and 4 represent waiting relationships between the transactions. \(ru_i[x] < cl_j[x]\) implies \(T_i \rightarrow T_j\), which represents that \(T_j\) is waiting for \(T_i\) to release a read lock on a data item \(x\). Although most waiting relationships indicate that the lower priority transaction must wait for the higher priority transaction to terminate, there may be the waiting relationship in which the higher priority transaction waits for the lower priority transaction.

**Lemma 5**: Let \(S\) be a schedule produced by the scheduler. If a path from \(T_i\) to \(T_j\) \((i \neq j)\) is in a WFG for \(S\), then \(T_i \in\) Before Set\(^*\)[\(T_j\)]

**Proof**: By Property 3 and 4, there are three kinds of waiting relationships such as \(ru_j[x] < cl_i[x]\), \(cu_j[x] < ru_i[x]\), and \(cu_j[x] < cl_i[x]\) in \(H\). Such partial orders lead to \(r_j[x] < c_i[x]\), \(c_j[x] < r_i[x]\) and \(c_j[x] < c_i[x]\), respectively. In any case, \(T_j\) precedes \(T_i\) in \(H\). Hence, \(T_i \in\) Before Set\(^*\)[\(T_j\)], as desired. \(\square\)

**Theorem 2**: A WFG for a schedule produced by the scheduler is an acyclic graph.

**Proof**: Suppose that a cycle \(T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1\) \((n > 1)\) is in the WFG for a schedule \(S\). By Lemma 5, \(S\) contains precedence relationship of \(T_j \in\) Before Set\(^*\)[\(T_i\)] and \(T_j \in\) After Set\(^*\)[\(T_i\)], for \(i, j\) \(1 \leq i, j \leq n\). Our proposed algorithm resolves data conflicts so that a transaction cannot be in both Before Set\(^*\) and After Set\(^*\) of any transaction. Hence in our proposed algorithm, a WFG cannot contain a cycle. The proof is completed. \(\square\)

4 Concluding Remarks

In this paper, we have proposed the real-time concurrency control algorithm based on MV2PL, which utilizes precedence relationship of conflicting transactions to adjust a serialization order dynamically. By incorporating MV2PL, our proposed algorithm can increase the degree of concurrent write operations and lead to the enhanced system performance.

Though the original proposal of MV2PL suffers from priority inversion problem, our algorithm try to minimize priority inversion by re-arranging conflicting transactions according to priority in their commit phases. Dynamic adjustment of a serialization order is achieved by updating precedence relationships among the transactions in data conflicts whenever needed. As a result, it is possible to resolve data conflicts by using more reliable information about the conflicting situation between the transactions.

A higher priority transaction is never aborted by lower priority transactions, but may be blocked by a lower priority transaction which is in the commit phase. This saves system resources used by a lower priority transaction at the expense of blocking a higher priority transaction. Our algorithm not only gives superior service to the urgent transactions, but also reduces the waste of system resources by avoiding unnecessary aborts of lower priority transactions.

We have also shown that our proposed algorithm produces a serializable schedule and does not lead to deadlocks which can significantly degrade the performance of a real-time database system.

References


