1994

An Adaptable Constrained Locking Protocol for High Data Contention Environments

Shalab Goel

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
94-049

An Adaptable Constrained Locking Protocol for High Data Contention Environments

Shalab Goel and Bharat Bhargava
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

CSD-TR-94-049
July, 1994

An Adaptable Constrained Locking Protocol for High Data Contention Environments

Shalab Goel Bharat Bhargava
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
Email: {*sgoel,bb*}*@cs.purdue.edu*

Multiversion concurrency control schemes are often limited in their practicability due to their storage requirements for multiple versions of the data. However, a class of multiversion schemes utilize only the versions, maintained for the purpose of recovery, to improve the concurrency by allowing the concurrent execution of "non conflicting" read-write lock requests on different versions of data in an *arbitrary* fashion. A transaction that accesses a data item version which is later diagnosed to lead to a incorrect execution, is aborted. This act is reminiscent of the validation phase in the optimistic concurrency schemes. Various performance studies suggest that these schemes perform poorly in high data contention environments where the excessive transaction aborts result, due to the failed validation. We propose an adaptable constrained two version two phase locking (*C2V2PL*) scheme in which these "non conflicting" requests are allowed only in a *constrained* manner. C2V2PL scheme *assumes* that a lock request failing to satisfy the specific constraints will lead to an incorrect execution and hence, must be either rejected or blocked. This eliminates the need for a separate validation phase. When the contention for data among the concurrent transactions is high, the C2V2PL scheduler in *aggressive* state rejects such lock requests. The deadlock free nature of C2V2PL scheduler in this state further reduces the duration for which locks are held by a transaction. The C2V2PL scheduler *adapts* to the low data contention environments by accepting the lock requests that have failed the specific constraints but contrary to the assumption will not lead to an incorrect execution. Thus improving the performance due to reduced transaction aborts in this *conservative* state.

1 Introduction

Many multiversion concurrency control schemes using a bounded number of versions for the data items have been proposed for improving the performance of transaction processing. These schemes have been broadly categorized under mixed and pure multiversion schemes in [BHG87]. The mixed multiversion schemes [CFL+82, Wei87, AS89, BC91] have two types of transactions, i.e. the read-only transactions and the update transactions. The read-only transactions read the old but consistent versions while the update transactions manipulate *only* the "current" version via two phase locking (2PL) protocol. Even if we assume that the transaction type can be determined for every transaction when it starts executing, which is not the case for at least the on-line transactions, the increase in the size and frequency of the update transactions because of increased acceptance of the transaction as an organizational concept for a wider variety of applications (e.g. the database servers [Val93] on the information superhighways), limits the performance of the system if only the "current" version is available for their synchronization. In high data contention applications like stock exchange databases [PR88], the mixed schemes will pose the problems for the update transactions same as in ordinary two phase locking schemes [TGS85].

Pure multiversion schemes using two phase locking [BHR80, SR81, BHG87, KSI91] utilize the versions, maintained by the system for the reasons of recovery, for allowing the concurrent execution of the conflicting transactions. The two phase locking for write-write synchronization puts an upper bound on the number of versions for every data item. Since the concurrent access to the conflicting read-write actions is allowed on different versions of a data item in an *unrestricted* fashion, the execution of each transaction must be validated before its effects can be committed. This validation is usually performed at the end of the transaction execution, either because it is computationally expensive to validate each

action executed on behalf of the transaction [BHR80, SR81] or because the scheme does not allow any other validation point [BHG87]. In any case, the effort in executing the transaction that fails the validation is wasted. These pure multiversion schemes will be recognized as *optimistic* concurrency schemes in the taxonomy of schedulers by [BHG87]. In the optimistic schemes, the transaction aborts due to the failed validation grows rapidly with the increase in contention for data [ACL87]. The effect of these aborts on the system performance becomes more prominent as the size of the transaction grows.

In this paper, we present an *adaptable* Constrained **T**wo **V**ersion **T**wo **P**hase **L**ocking (*C2V2PL*) scheme for synchronising the read and write lock requests on the different versions of a data item in only a constrained manner. The constraints are specified in terms of timestamps on the lock requested and on the locks held for the data item. The correctness of the transaction execution is guaranteed if the transaction can announce its completion, by submitting its commit action, to the scheduler. No separate validation phase for validating the transaction execution is required. A maximum of two *committed* versions of a data item are available at any given time. A read request is completed by using the Read rule similar to the multiversion timestamp ordering (MVTO) read rule in [BG83]. The action taken by the scheduler on the lock request that fails to satisfy the constraints is dependent on the scheduler `state`. When the conflicts for data is high, such lock requests are rejected and the scheduler is said to be in *aggressive* state. When the data contention is low, these lock requests are blocked and the scheduler is said to be in *conservative* state. In the aggressive state, since no lock request gets blocked for indefinite periods of time, the conflicting transactions never deadlock on a lock request. In the conservative state, the blocking of these lock requests may lead to deadlock, but may also improve the transaction throughput by avoiding the *unnecessary* abort of the transactions.

The rest of the paper is organized as follows. In section 2, we present the transaction model and the database model used in C2V2PL. We present the adaptable C2V2PL scheme in conservative and aggressive states in section 3. The comparative behavior of C2V2PL in these states is illustrated via sample execution. The correctness of C2V2PL scheme is proved in section 4. We conclude the paper in section 5.

2 Transaction Model

A transaction is a partial order on a set of read and write actions. The last action of the transaction, commit or abort, indicates whether its execution has completed successfully or not. Each transaction T_i is assigned a unique timestamp $ts(T_i)$. For simplicity, we assume that $ts(T_i) = i$. Each action maintains the timestamp of its transaction.

We assume that the *C2V2PL* scheduler starts in an initial correct and consistent database state D_0, with a single version x_0^0 for each data item x in the database. The notation x_j^k is used as follows: k is the timestamp of the transaction T_k that wrote the version x_j^k of the data item x; $j = ts(x_j^k)$ is the current timestamp of the version x_j^k used in version selection to process a read action on data item x. As shown in the figure 1, a version for a data item x is created as x_k^k by the transaction T_k, becomes accessible to other transactions as x_k^k after T_k **commits**, and can be accessed as x_0^k after T_k **terminates**. Thus, the version x_0^k of data item x is always due to a *terminated* transaction T_k, and the version x_j^j is always due to either *active* or *committed but not yet terminated* transaction T_j. We will explain the termination and commitment of a transaction later in this section.

2.1 Concurrency Control

A write action on data item x in transaction T_i, $W_i(x)$, uses the following locking protocol.

	rl	w l	vl
rl	√	√	√
wl	√	×	×
vl	√	×	×

Table 1: The higher level lock conflict matrix

1. T_i requests a write lock on the data item x.

2. scheduler grants the $wl_i(x)$ write lock on data item x if there are no conflicts and the lock request satisfies the specified *constraints.*

3. T_i creates a new version x_i^i for the data item x.

As shown in Table 1, since the write locks conflict, there can be utmost one uncommitted version x_i^i written by some transaction T_i, where T_i holds the $wl_i(x)$ lock. As we will see later, the conflict of write lock with verified (vl) lock limits the number of *committed* versions of any data item x, available at a given time, to a maximum of two versions: x_0^j and x_i^i; where the version x_0^j is due to the *most recently* terminated transaction T_j that wrote x or x_0^j is in initial consistent database state D_0, and the version x_i^i is written by the *currently* committed but not yet terminated transaction T_i. The constraints that the lock request must satisfy to be granted are described in the section 3.

A read action is completed in accordance with the Read rule similar to the multi version timestamp ordering (MVTO) Read rule in [BG83].

Read Rule: *The* `committed` *version of the data item with the largest timestamp less than or equal to the timestamp of the transaction making the read request is selected.*

The scheduler maintains two versions of the read lock for each data item x, i.e. $rl^0(x)$ lock and $rl^{\neq 0}(x)$ lock. Since utmost two committed versions of a data item are available for the scheduler to choose from, there is a one to one correspondence between the read lock version granted and the data item version selected.

A read action on a data item x in transaction T_i, $R_i[x]$, is completed as follows.

1. T_i requests a read lock on the data item x.

2. scheduler grants the $rl_i^0(x)$ or $rl_i^{\neq 0}(x)$ read lock corresponding to whether the version x_0^j or version x_k^k (if it exists and is committed) is selected in accordance with the Read rule; and this read lock version satisfies the specified *constraints.*

3. T_i reads the selected version of x after obtaining the corresponding read lock version.

The scheduler processes the read action $R_i[x]$ by selecting the committed version x_k^k if $ts(T_i) \geq ts(x_k^k)$; and the version x_0^j otherwise; after granting the read lock version $rl_i^{\neq 0}(x)$ or $rl_i^0(x)$ to T_i, respectively. However, to avoid the incorrect execution as explained in Section 3, the read lock request for $R_i[x]$ is blocked if the version x_k^k, with $ts(T_i) > ts(x_k^k)$, exists but is not committed. This lock request is said to have failed a *constraint* and must not be allowed to proceed. As we see in the next section, a version x_0^j (for some $j \geq 0$) always exists for each data item x, which implies by the Read rule that every read action $R_i[x]$ can be processed.

2.2 Version Control

We now describe the versioning control mechanism in C2V2PL scheme. A transaction can be in one of the three modes: `active` mode, `passive` mode or `done` mode. A transaction T_i is in active mode when it is executing its read/write actions or is blocked waiting for its

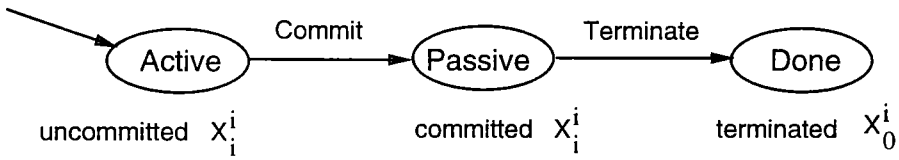

Figure 1: Life cycle of a transaction T_i

lock requests to be granted by the scheduler. A transaction T_i is in passive mode when the execution of all its read and write actions has been completed *successfully*. A transaction T_i is in done mode after the locks held by it can be released by the scheduler without compromising the future consistency of the database.

Commit or Abort of a Transaction The transition from the active mode to the passive mode for a transaction T_i is triggered by the execution of its commit action, c_i. The scheduler processes the c_i by converting each of the $wl_i(x)$ lock held by T_i into a third kind of lock called the verified lock, $vl_i(x)$. This conversion makes the version x_i^i written by T_i accessible to the other active transactions. Thus, the commitment of a transaction represents the *growing phase* of the number of the *committed* versions of the data item written by it. None of the read locks held by T_i are released during this transition. As shown in Table 1, since the *vl* locks and the *wl* locks conflict, no other transaction is allowed to write x while T_i is in passive mode, i.e. while T_i is committed but has not yet terminated.

The abort action a_i for transaction T_i is processed by purging the new versions written by T_i and releasing all the locks held by it. Since only the committed versions of any data item can be accessed by the other transactions, the *cascading abort* of the transactions is avoided.

Termination of a transaction The transition from the passive mode to the done mode for a transaction T_i occurs when the scheduler invokes and executes the terminate action t_i. The invocation of t_i determines when, for each data item x for which transaction T_i has written a committed version x_i^i, can the existing [1] version x_0^j be deleted so that the version x_i^i can be converted into x_0^i and the $vl_i(x)$ lock held by T_i can be released. A transaction blocked on a write lock request on data item x can proceed only after T_i has released the $vl_i(x)$ lock. Thus, the termination of a transaction represents the *shrinking phase* of the number of *committed* versions of the data items written by it. The terminate action t_i is executed as an *atomic* operation and is processed as follows.

1. the read locks held by the transaction T_i are released.

2. for each $vl_i(x)$ lock held by T_i, convert all the $rl^{\neq 0}(x)$ locks, held by the other transactions in active mode, into $rl^0(x)$ locks.

3. for each $vl_i(x)$ lock held by T_i, purge the previously existing version x_0^j; convert the committed version x_i^i into a version x_0^i by resetting the timestamp to zero; release the $vl_i(x)$ lock.

Thus, for each data item x, there exists either a terminated (and hence committed) version x_0^j written by most recently terminated transaction T_j that wrote x or $x_0^0 \in D_0$; and at most one uncommitted or committed version x_i^i, written by a transaction T_i that holds the exclusive $wl_i(x)$ or $vl_i(x)$ lock on x respectively.

It must be noted that the terminate action for transaction T_i may not be invoked immediately after T_i commits. This is because the simple assignment of the new version for every data item request in the future does not work for the reasons of consistency. Consider

[1] written by most recently terminated transaction T_j that wrote x or $x_0^0 \in D_0$.

the two transactions $T_1 = R_1[x]R_1[y]$ and $T_2 = W_2[x]W_2[y]$ and the following history:

$$\mathrm{D_0} \;||\quad rl_1^0(x)\, R_1[x_0^0]\; wl_2(x)\, W_2[x_2^2]\; wl_2(y)\, W_2[y_2^2]\; c_2$$

The scheduler starts in an initial consistent database state D_0. It selects the version x_0^0 for processing $R_1[x]$ and grants $rl_1^0(x)$ lock to the transaction T_1. The transaction T_2 writes the versions x_2^2 and y_2^2 after it is granted the $wl_2(x)$ and $wl_2(y)$ write locks. The scheduler processes the commit action c_2 for T_2 by converting the $wl_2(x)$ and $wl_2(y)$ locks into $vl_2(x)$ and $vl_2(y)$ locks. The versions x_2^2 and y_2^2 become accessible to other active transactions. Suppose that the scheduler were allowed to terminate the transaction T_2. The previously existing versions x_0^0 and y_0^0 would be deleted, and the versions x_2^2 and y_2^2 would be converted into the versions x_0^2 and y_0^2 respectively. If the scheduler now processed $R_1[y]$ by selecting the *only available* version of data item y, i.e. the version y_0^2 in accordance with the Read rule (since $ts(T_1) > ts(y_0^2)$), there would be no serial execution of the transactions T_1 and T_2. This is because in reading the version x_0^0, T_1 saw the database in a state before the execution of T_2, and in reading the version y_0^2, T_1 saw the database in a state after the execution of T_2.

To determine when the terminate action for a committed transaction can be invoked by the scheduler, we define the following irreflexive, transitive relation.

T_i **precedes** T_j : $\Leftrightarrow$

$$(\exists x)\ [(rl_i^0(x) \text{ and } wl_j(x)) \text{ or } (rl_i^0(x) \text{ and } vl_j(x)) \text{ or } (rl_j^{\neq 0}(x) \text{ and } vl_i(x))\ ]$$

i.e. the transaction T_i *precedes* the transaction T_j if either T_i has read a previously existing version of a data item for which T_j has created a new version, or T_j has read the committed version of the data item written by T_i.

T_j **terminates** : $\Leftrightarrow$ *($\nexists$ T_i) (T_i* **precedes** *T_j)*

which says that the transaction T_j can not terminate, until each transaction T_i that has

either read the version x_0^k (for some k) or written the committed version x_i^i that has been read by T_j has terminated.

By the unary relation *terminates*, in the example above, since T_1 has read the previously existing version x_0^0 of data item x and T_2 has created the new committed version x_2^2, the termination of T_2 must be delayed until after T_1 has terminated. This allows the scheduler to make the correct version selection for $R_1[y]$ from the two available committed versions y_0^0 and y_2^2, i.e. the version y_0^0 with $ts(y_0^0) < ts(T_1) < ts(y_2^2)$. The latter requirement in *terminates* is not as obvious and its need is illustrated with the help of another example. Consider the following transactions $T_3 = R_3[x]R_3[y]$, $T_4 = W_4[x]$, $T_5 = R_5[x]W_5[y]$ and their execution history:

$$\mathrm{D}_0 \quad || \quad rl_3^0(x)\, R_3[x_0^0] \;\; wl_4(x)\, W_4[x_4^4] \;\; c_4 \;\; rl_5^{\neq 0}(x)\, R_5[x_4^4] \;\; wl_5(x)\, W_5[x_5^5] \;\; c_5$$

As explained above, transaction T_4 can not terminate until transaction T_3 terminates. However, suppose that transaction T_5 were terminated and the previously existing version y_0^0 replaced by version y_0^5 obtained from the committed version y_5^5. If the scheduler now processed $R_3[y]$ by selecting the *only available* version of data item y, i.e. the version y_0^5 in agreement with the Read rule (since $ts(T_3) > ts(y_0^5)$), there would be no serial execution of T_3, T_4 and T_5. T_3 sees the database state before T_4 in executing $R_3[x_0^0]$, T_5 sees the database state after T_4 in executing $R_5[x_4^4]$, and T_3 sees the database state after T_5 in executing $R_3[y_0^5]$.

It must be noted that the processing of the commit action for a transaction does not require a validation phase to check for the correctness of its execution. The execution of a transaction is *guaranteed* to be correct if its commit action can be submitted to the scheduler. This is because the read and write lock requests on the different versions of a data item are allowed in such a constrained manner that every read action $R_k[x_a^j]$ can

be processed in conformity with the Read rule and without leading to a non-serializable execution. The lock requests failing the constraints are handled in a manner concomitant with the scheduler state. Since the inconsistencies due to incorrect version access of a data item always manifest as a lock request failing the constraints, the effort in executing the transaction completely, only to find during the validation phase (in comparable schemes) that it has been executed incorrectly, can be saved by not granting such lock requests.

3 Adaptable Constrained Two Version 2PL

The C2V2PL scheme utilizes the unique timestamp associated with a transaction for ordering the "non conflicting" read and write lock requests on the different versions of a data item. It rejects or blocks the lock requests that fail to observe this ordering which is imposed by a set of constraints stated below. The *anticipated* invalidating lock requests coincide with these lock requests failing the constraints. It must be noted that not every such lock request will actually lead to the invalid execution of the transaction. The scheduler executes in one of the two states - **conservative** or **aggressive** depending on the contention for data among the transactions in the system. If the data contention is high, to avoid deadlocks and to minimize the duration for which the locks will be held by a transaction, these requests failing the constraints are rejected. However, if the data contention is low, to avoid the unnecessary abort of the transactions, these requests are blocked.

As described in the previous section, for each data item x, there is *always* a version x_0^k with timestamp equal to zero, and *utmost* one committed version x_j^j with timestamp equal to j. Thus, an appropriate version of data item x can always be selected for processing $R_i[x]$ and the corresponding read lock version can always be granted. However, a read lock request on a data item x by the transaction T_i must satisfy the following constraint:

Constraint$_1$: *If a transaction T_j holds $wl_j(x)$ lock, then $ts(T_j) \geq ts(T_i)$.*

Since the transaction T_j holds the $wl_j(x)$ lock, there is only one available committed [2] version of data item x , i.e. the version x_0^k for processing $R_i[x]$. Suppose this version were selected by the scheduler. If the T_j commits and makes the version x_k^k accessible to T_i, then $R_i[x]$ has not read the committed version with the largest timestamp less than $ts(T_i)$, i.e. the version x_j^j; hence breaking the Read rule. Thus, the lock request for $R_i[x]$ must remain blocked until it satisfies the *Constraint$_1$*, i.e. until $wl_j(x)$ is converted into $vl_j(x)$ lock, or in other words until T_j commits.

A write lock request $wl(x)$ for transaction T_i must satisfy the following constraint:

Constraint$_2$: *There does not exist a transaction that holds $wl(x)$ or $vl(x)$ lock and for all transactions T_j that hold $rl_j^0(x)$, $ts(T_i) \geq ts(T_j)$*

Note that no transaction could not be holding a $rl^{\neq 0}(x)$ lock since no other transaction is holding a $vl(x)$ lock. This stems from the fact that terminate action always converts each of the $rl^{\neq 0}(x)$ locks into a $rl^0(x)$ lock before it releases its $vl(x)$ lock . The failure of *Constraint$_2$* by a write lock request may lead to the following scenario. Consider the two transactions $T_6 = R_6[x]W_6[y]$ and $T_7 = R_7[y]W_7[x]$ and the following history of execution:

$$D_0 \quad || \quad rl_6^0(x)\, R_6[x_0^0] \;\; rl_7^0(y)\, R_7[y_0^0] \;\; wl_7(y)\, W_7[x_7^7] \;\; c_7$$

$W_6[y]$ arrives and suppose $wl_6(y)$ lock were granted. T_6 now submits its commit action. The scheduler would process the request by converting the $wl_6(y)$ lock into $vl_6(x)$ lock. There is no serial execution of T_6 and T_7. But this contradicts our claim that a transaction that can submit its commit action is *guaranteed* to have executed correctly. The write action $W_6[x]$ is a *missed* write in the terminology of the MVTO scheme [BG83] and is rejected. However, in the adaptable C2V2PL scheme, such a write request may be rejected or blocked

[2]The other version x_j^j written by the transaction T_j is still uncommitted.

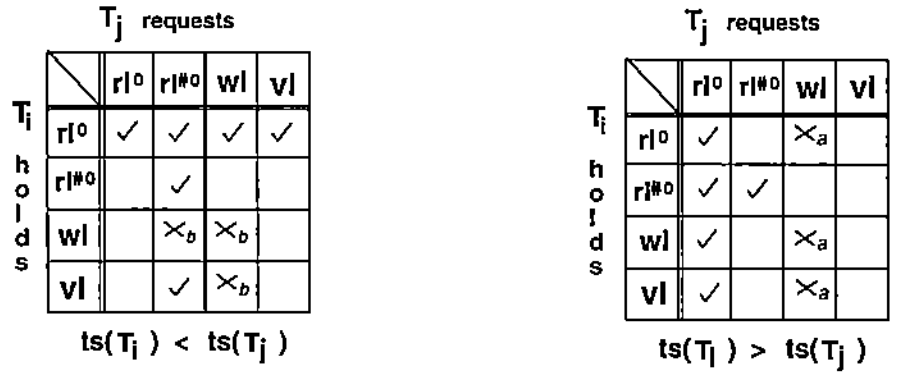

Figure 2: Constrained Conflict Graph for Aggressive State for C2V2PL

depending upon the *state* of the scheduler.

3.1 Aggressive State

The *C2V2PL* scheduler in the aggressive state uses the following rule for avoidance of deadlocks due to conflicting *wl* and *vl* locks:

Conflict Resolution Rule: *If a transaction T_i holds a $wl_i(x)$ or $vl_i(x)$ lock, then the write $wl_j(x)$ lock request $wl_j(x)$ by the transaction T_j is rejected if $ts(T_j) > ts(T_i)$; and is blocked otherwise.*

The *conflict resolution rule* along with the rejection of the write lock requests that fail the $Constraint_2$ makes the *C2V2PL* scheduler in aggressive state, deadlock free. Figure 2 shows how the timestamped lock requests are handled by the *C2V2PL* scheduler in the aggressive state. "X_a" and "X_b" refer to the constrained conflicting request which is rejected and blocked respectively. For example, if the transaction T_j requests a $wl_j(x)$ lock when T_i holds $rl_i^0(x)$, with $ts(T_i) > ts(T_j)$, the action taken by the scheduler is "X_a", since $wl_j(x)$ lock request has failed to satisfy the $Constraint_2$. Furthermore, every read action can be completed by granting either rl^0 or $rl^{\neq 0}$ read lock. The action of the scheduler for the

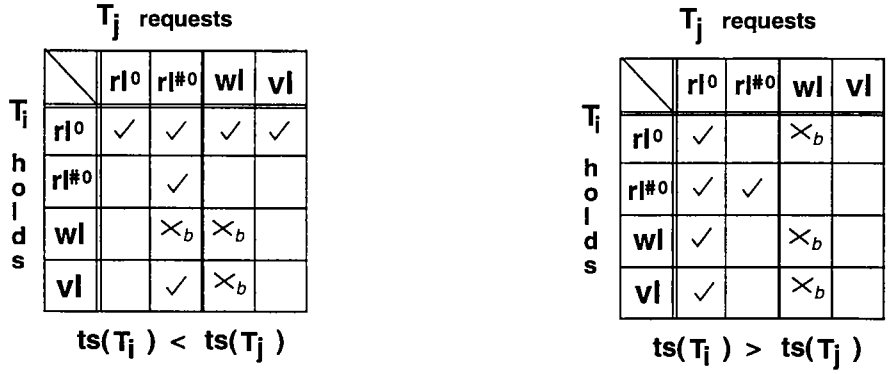

Figure 3: Constrained Conflict Graph for Conservative State for C2V2PL

read lock request that fails the *Constraint*$_1$ is "X_b". Notice, that since the lock requests are blocked only in an asymmetric fashion, i.e. only a transaction with higher timestamp may be blocked by a lock held by a transaction with a lower timestamp, there can be no deadlocks in aggressive state of C2V2PL scheduler.

3.2 Conservative State

Based on the assumption, that in low data contention environments, there will be little inconsistent access to data, the *C2V2PL* scheduler in conservative state does not reject but blocks the lock request it *anticipates* will lead to an incorrect transaction execution. This lock request will result in a deadlock if its execution can *indeed* lead to an invalid execution; and will be rejected when the scheduler times out to resolve this deadlock. As shown in the Fig. 3, the action of the scheduler for a write lock request that fails *Constraint*$_2$ is "X_b". A transaction with a write lock request is *unconditionally* blocked if another transaction already holds a write or a verified lock on that data item. The *C2V2PL* scheduler in this state avoids unnecessary rejects of the lock requests that, governed by the failure of constraints are anticipated to, but do not actually lead to an incorrect execution.

Illustrative Example The following sample execution compares the behavior of the C2V2PL scheduler in aggressive and conservative state. Consider the following transactions $T_8 = R_8[z]W_8[x]$, $T_9 = R_9[x]R_9[z]W_9[y]$, $T_{10} = R_{10}[y]W_{10}[z]$ and the following order of requests submitted to the scheduler: $R_8[z]$, $R_9[x]$, $R_{10}[y]$, $W_8[x]$, $R_9[z]$, $W_{10}[z]$, $W_9[y]$. Assume an initial consistent database state D_0.

The *C2V2PL* scheduler in *aggressive* state processes $R_8[z]$, $R_9[x]$, and $R_{10}[y]$ as $R_8[z_0^0]$, $R_9[x_0^0]$, and $R_{10}[y_0^0]$ after granting the $rl_8^0(z)$, $rl_9^0(x)$, and $rl_{10}^0(y)$ locks to the transactions T_8, T_9 and T_{10} respectively. The $wl_8(x)$ lock request for $W_8[x]$ fails to satisfy $Constraint_2$ and is rejected. $R_9[z]$ is processed as $R_9[z_0^0]$ after the $rl_9^0(z)$ lock is granted to T_9. The $wl_{10}(z)$ lock request for $W_{10}[z]$ is granted and T_{10} writes the version z_{10}^{10}. T_{10} commits and $wl_{10}(z)$ lock is converted into $vl_{10}(z)$ lock. The $wl_9(y)$ lock request for $W_9[y]$ fails to satisfy $Constraint_2$ and is rejected. The scheduler invokes the terminate action t_{10} and the $rl_{10}^0(y)$ and $vl_{10}(z)$ locks are released and the version z_0^0 is replaced by z_0^{10} obtained from version z_{10}^{10}.

The *C2V2PL* scheduler in *conservative* state processes $R_8[z]$, $R_9[x]$, and $R_{10}[y]$ in exact same way as in aggressive state. The $wl_8(x)$ lock request fails $Constraint_2$ and is blocked. $R_9[z]$ and $W_{10}[z]$ are processed as in aggressive state. T_{10} commits. The $wl_9(y)$ lock request fails $Constraint_2$ and is blocked. A deadlock situation now results. To terminate T_{10}, the scheduler must wait until T_9 releases its $rl_9^0(z)$ lock. On the other hand, T_9 is waiting for T_{10} to release its $rl_{10}(y)$, so that the $wl_9(y)$ lock request can be unblocked.. The deadlock is resolved by aborting the transaction T_9. The $wl_8(x)$ lock request blocked by the failure of $Constraint_2$, can now be granted. T_8 commits and is eventually terminated by the scheduler. The scheduler can now terminate the transaction T_{10}.

The case of reduced number of transaction aborts in low data contention environment

at the expense of increased blocking is motivated by the *C2V2PL* scheduler in conservative state. In higher data contention environments, the blocking of the transactions is minimized at the cost of increased number of transaction restarts by the *C2V2PL* scheduler in aggressive state.

4 Correctness of C2V2PL

We will prove the correctness of *C2V2PL* scheme by describing it in multiversion serializability theory and confirming that all the histories produced by C2V2PL are 1SR. The interested reader is directed to the theory of multiversion serializability in [BG83].

Let H be a history over { T_1, T_2, T_3, ...} produced by C2V2PL. Then H must satisfy the following properties.

C2V2PL$_1$: For each T_i, there is a unique timestamp $ts(T_i)$. For simplicity, we assume that $ts(T_i) = i$.

C2V2PL$_2$: For each T_i, the terminate action t_i follows the commit action, c_i; i.e. $c_i < t_i$.

C2V2PL$_{3a}$: For each $R_k[x_0^j] \in H$, either (1) $t_j < R_k[x_0^j]$ and $j > 0$; or (2) $x_0^0 \in D_0$.

C2V2PL$_{3b}$: For each $R_k[x_j^j] \in H$, either (1) $c_j < R_k[x_j^j] < t_j < t_k$ and $ts(x_j^j) < ts(T_k)$; or (2) $W_j[x_j^j] < R_k[x_j^j]$ and $j = k$.

C2V2PL$_4$: For each $R_k[x_a^l]$ and $W_k[x_k^k] \in H$; if $W_k[x_k^k] < R_k[x_a^l]$ then $a = k$ and $l = k$.

Properties $C2V2PL_{3a,3b}$ together say that every Read $R_k[x]$ either reads a committed version or reads a version written by itself (i.e. T_k). In either case, it reads the version with the timestamp less than or equal to $ts(T_k)$. $t_j < t_k$ in property $C2V2PL_{3b}$ follows from the definition of unary relation *terminates*. Property $C2V2PL_4$ says that if T_k wrote x before the scheduler received $R_k[x]$, it translates the request to read the version written by T_k.

C2V2PL$_{5a}$: For every $R_k[x_0^j]$ and $W_i[x_i^i] \in H$; either $t_i < R_k[x_0^j]$ or $R_k[x_0^j] < t_i$.

Property $C2V2PL_{5a}$ says that $R_k[x_0^j]$, i.e. a Read on the version x_0^j, created by the terminated transaction T_j, is strictly ordered with respect to the terminate action of every transaction that writes x. This is because each transaction T_i that writes x_i^i holds a verified lock $vl_i(x)$, while it waits for each transaction that has read the existing version x_0^j to terminate, before it can terminate and release $vl_i(x)$ lock. Since the vl and wl locks conflict, for each transaction T_k that reads x_0^j, either T_i must have terminated before T_j even got the $wl_j(x)$ lock, i.e. $t_i < wl_j(x) < t_j < R_k[x_0^j]$; or T_i must have terminated after T_k reading the version x_0^j had terminated, i.e. $R_k[x_0^j] < t_k < t_i$.

C2V2PL$_{5b}$: For every $R_k[x_j^j]$ and $W_i[x_i^i] \in H$; if $W_i[x_i^i] < R_k[x_j^j]$ then (1) $t_i < R_k[x_j^j]$; else (2) $R_k[x_j^j] < t_i$ and $t_k < t_i$.

Property $C2V2PL_{5b}$ says that $R_k[x_j^j]$, i.e. a Read on a committed version x_j^j due to a committed but not terminated T_j is strictly ordered with respect to the terminate action of every transaction that writes x. (1) says that since the vl and wl locks conflict, T_i must have terminated and released the $vl_i(x)$ lock before T_j even got the $wl_j(x)$ lock, i.e. $t_i < wl_j(x) < c_j < R_k[x_j^j]$; (2) By definition of the terminate action, t_j converts the version x_j^j read by $R_k[x_j^j]$ into x_0^j; converts the $rl_k^{\neq 0}(x)$ lock into $rl_k^0(x)$ lock; and then releases the $vl_j(x)$ lock. By the Property $C2V2PL_{3b}$, $t_j < t_k$. Thus after T_j terminated and before T_k terminates, if $ts(T_k) > ts(T_i)$, $wl_i(x)$ lock request must wait for T_k to terminate and release the now $r_0^k(x)$ lock in accordance with *Constraint*$_2$, i.e. $R_k[x_j^j] < t_j < t_k < wl_i(x) < t_i$; otherwise T_i obtains the $wl_i(x)$ lock, writes the version x_i^i, and then waits for T_k that has read, the now version x_0^j to terminate. i.e. $R_k[x_j^j] < t_j < wl_i(x) < t_k < t_i$.

C2V2PL$_{6a}$: For every $R_k[x_0^j]$ and $W_i[x_i^i]$, (i, j, k distinct); if $t_i < R_k[x_0^j]$ then $t_i < t_j$.

$C2V2PL_{6b}$: For every $R_k[x_j^j]$ and $W_i[x_i^i]$, (i, j, k distinct); if $t_i < R_k[x_j^j]$ then $t_i < t_j$.

Property C2V2PL$_{6a}$ says that $R_k[x_0^j]$ reads the most recently terminated version of x.

Assume to the contrary that $t_j < t_i$. But then, the version x_0^j generated when T_j terminated must have been deleted and replaced by x_0^i when T_i terminates, and thus $R_k[x]$ could not have accessed x_0^j. Property C2V2PL$_{6b}$ combined with Property C2V2PL$_{3b}$ says that $R_k[x_j^j]$ either reads the version written by itself or the most recently committed version x_j^j. Since the *vl* and *wl* locks conflict, if $t_i < R_k[x_j^j]$ then $t_i < W_j[x_j^j] < c_j$, which combined with Property C2V2PL$_{3b}$ says $t_i < t_j$.

$C2V2PL_{7a}$: For every $R_k[x_0^j]$ and $W_i[x_i^i]$, $i \neq j, j \neq k$, if $R_k[x_0^j] < t_i$ then $t_k < t_i$.

$C2V2PL_{7b}$: For every $R_k[x_j^j]$ and $W_i[x_i^i]$, $i \neq j, j \neq k$, if $R_k[x_j^j] < t_i$ then $t_k < t_i$.

Property $C2V2PL_{7a,7b}$: says that T_i cannot terminate until every transaction that has read the existing terminated version, has terminated. Property $C2V2PL_{7a}$ follows directly from the definition of unary relation *terminates*. Property $C2V2PL_{7b}$ follows from Property $C2V2PL_{5b}$.

$C2V2PL_8$: For every $W_i[x_i^i]$ and $W_j[x_j^j]$, either $t_i < t_j$ or $t_j < t_i$.

Property $C2V2PL_8$ says that the termination of every two transactions that write the same data item are atomic with respect to each other.

Theorem: Every history H produced by the C2V2PL scheduler is 1SR.

Proof: By C2V2PL$_2$, C2V2PL$_{3a,3b}$, C2V2PL$_4$, H preserves reflexive reads-from relationship and is recoverable. Hence it is a MV history. Define a version order $\ll$ as $x^i \ll x^j$ only if $t_i < t_j$. By C2V2PL$_8$, $\ll$ is indeed a version order. We will prove that all edges in MVSG(H, $\ll$) are in the termination order. That is $T_i \rightarrow T_j$ in MVSG(H, $\ll$) then $t_i < t_j$.

Let $T_i \rightarrow T_j$ be in SG(H). This edge corresponds to a reads-from relationship such as T_j reads x from T_i. By C2V2PL$_{3a}$ $t_i < R_j[x_0^i]$ and from C2V2PL$_2$ $R_j[x_0^i] < t_j$. Hence $t_i < t_j$. Similarly, by C2V2PL$_{3b}$ for any $R_j[x_i^i]$, $t_i < t_j$.

Consider a version order edge induced by $W_i[x_i^i]$, $W_j[x_j^j]$ and $R_k[x_0^j]$, (i, j, k distinct). There are two cases: $x^i \ll x^j$ or $x^j \ll x^i$. If $x^i \ll x^j$, then the version order edge is $T_i \rightarrow T_j$, and $t_i < t_j$ follows from the definition of $\ll$. If $x^j \ll x^i$, then the version order edge is $T_k \rightarrow T_i$. Since $x^j \ll x^i$, $t_j < t_i$ follows from the definition of the version order. By C2V2PL$_{5a}$ either $t_i < R_k[x_0^j]$ or $R_k[x_0^j] < t_i$. In former case, C2V2PL$_{6a}$ implies that $t_i < t_j$ contradicting $t_j < t_i$. Thus $R_k[x_0^j] < t_i$ and by C2V2PL$_{7a}$ $t_k < t_i$ as desired. The case of the version order edge induced by $W_i[x_i^i]$, $W_j[x_j^j]$ and $R_k[x_j^j]$, (i, j, k distinct) can be proved in exactly same way and is left for the reader to work out.

This proves that all edges in the MVSG(H, $\ll$) are in termination order. Since the termination order is embedded in a history, which is acyclic by definition, MVSG(H, $\ll$) is acyclic too. Thus, H is 1SR.

5 Conclusions

We have proposed a new concurrency control scheme which utilizes the versions maintained for the purpose of recovery, to allow the concurrent execution of read-write actions on different versions of a data item in a constrained manner. These constraints not only eliminates the need for validation phase in transaction execution, but in high data contention environment guarantees deadlock free execution which further reduces the lock holding times for a transaction. The constraints are specified using the unique timestamps on the transactions making the lock requests. The scheme *adapts* to the low data contention environments by accepting those requests that fail the constraints but do not lead to a non serializable execution.

References

[ACL87] R. Agrawal, M. Carey, and M. Livny. Concurrency Control Performance Modelling: Alternatives and Implications. *ACM TODS*, 12(4):–, December 1987.

[AS89] D. Agrawal and S. Sengupta. Modular Synchronisation in Multiversion Databases: Version Control and Concurrency Control. In *ACM Proceedings of SIGMOD*, pages 408–417. ACM Press, New York, 1989.

[BC91] Paul M. Bober and Michael J. Carey. On Mixing Queries and Transactions via Multiversion Locking. Technical report, Computer Sciences Department, University of Wisconsin-Madison, nov 1991.

[BG83] Philip A. Bernstein and N. Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM TODS*, 8(4):465–483, December 1983.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading Mass., 1987.

[BHR80] R. Bayer, H. Heller, and A. Reiser. Parallelism and Recovery in Database Systems. *ACM TODS*, 5(2):139–156, June 1980.

[CFL+82] A. Chan, S. Fox, W. Lin, A. Nori, and D. Ries. The Implementation of An Integrated Concurrency Control and Recovery Scheme. In *ACM Proceedings of SIGMOD*, pages 184–191. ACM Press, New York, 1982.

[KSI91] R. Kataoka, T. Satoh, and U. Inoue. A Multiversion Concurrency Control Algorithm for Concurrent Execution of Partial Update and Bulk Retrieval Transactions. In *Proceedings 10th Int.l Phoenix Conference on Computers and Communications*, pages 130–136. IEEE Computer Society Press, New Jersey, 1991.

[PR88] Peter Peinl and Andreas Reuter. High Contention in a Stock Trading Database: A Case Study. In *ACM Proceedings of SIGMOD*, pages 260–268. ACM, New York, 1988.

[SR81] Richard S. Stearns and Daniel J. Rosenkrantz. Distributed Database Concurrency Controls using Before-Values. In *ACM Proceedings of SIGMOD*, pages 74–83. ACM Press, New York, 1981.

[TGS85] Y. C. Tay, N. Goodman, and R. Suri. Locking Performance in Centralized Databases. *ACM TODS*, 10(4):415–462, December 1985.

[Val93] Patrick Valduriez. Parallel Database Systems: Open Problems and New Issues. *Distributed and Parallel Databases*, 1(2), April 1993.

[Wei87] William E. Weihl. Distributed Version Management for Read-Only Actions. *IEEE Transactions on Software Engineering*, 13(1):55–64, January 1987.