CAN-FOLLOW CONCURRENCY CONTROL*

Peng Liu
School of Info. Sci. and Tech.
Pennsylvania State University
University Park, PA 16802, USA
pliu@ist.psu.edu

Sushil Jajodia, Paul Ammann
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030, USA
{jajodia, pammann}@gmu.edu

Jie Li
Institute of Info. Sci. and Electronics
University of Tsububa
Tsukuba Science City, Japan
lijie@is.tsukuba.ac.jp

Abstract

We present an efficient protocol which permits a transaction to read (write) an item write-locked (read-locked) by another transaction with almost no delays when there is a specific readset-writeset relationship, namely, *can-follow*, between the two transactions, and they arrive at the system in a desired order. One *transient* old committed version for some data item is kept for read purpose. This approach mitigates the lock contention not only between update and read-only transactions, but also between update and update transactions. This approach is simple (i.e., it is built on top of 2PL and needs only one list for each active transaction to maintain the can-follow relationship), flexible (in terms of the number of versions maintained and the amount of can-follow relationships used), efficient, and can be easily extended to distributed database environments.

Keywords: can follow, concurrency control, transaction processing

1 Introduction

Concurrency control protocols are used in a database system to coordinate concurrently executing *transactions*. Informally, a transaction is an execution of a program that accesses a shared database (one or several times). An access can be either a read operation or a write operation. A read operation on a data item does not conflict with another read operation, since neither modifies the data item. A write operation, on the other hand, conflicts with both read and write operations. Concurrency control protocols are applied to ensure that the interleaving of conflicting operations of transactions do not violate the consistency of the database.

Standard locking (i.e., strict two-phase locking (2PL)), where a data item can be accessed only after a lock on that item has been acquired, is almost exclusively used by current database management systems(DBMS) [3, 5]. The performance of these DBMSs may be limited by lock contention, since both the requirement for achieving higher transaction throughputs and the increasing transaction complexity can result in an increase in the probability of lock conflict. In the case of standard locking this is in the form of transaction blocking due to lock conflicts, which may lead to thrashing behavior in which the majority of transactions in the system are blocked [10].

To avoid or reduce the lock contention between conflicting transactions, it has been proposed, for example, to keep shadow pages around for use by the read operations that arrive late. R. Bayer et al. [2] have designed a locking scheme where two versions are kept but it requires an extra lock mode and the continuous and costly maintenance of a graph which tracks the dependencies among transactions to avoid nonserializable executions of transactions (of course, this graph is useful for detecting deadlocks also). And it needs to analyze the dependency graph to determine which version of a data item should be read.

P. A. Bernstein et al. [3] have described a two version two phase locking protocol (2V2PL) (a similar protocol proposed in [8]) which decreases the complexity of [2] substantially, for example, the dependency graph need no longer be maintained. However it has the following two disadvantages: (1) Similar to [2], it needs to introduce another lock mode, *certify* locks. When a transaction T_i commits, T_i must first convert all its write locks to certify locks. The conversion needs to examine every data item x written by T_i and will be blocked if there are any read locks on x. Thus using certify locks costs unneglectable processing resources. (2) Compared with 2PL, 2V2PL can cause extra deadlock overhead in many situations. For example, when two transactions T_i and T_j are involved in a 2PL deadlock where each one wants to read an item write-locked by the other, these reads by 2V2PL can be processed until the time of lock conversion. Processing these reads is a waste of time.

We remedy this situation by proposing an alternative locking strategy where the can-follow relationships between transactions are exploited. Informally, transaction T_1 can follow transaction T_2 if T_2 does not read data items written by T_1, and they write different sets of

*Jajodia, Ammann were partially supported by Rome Laboratory, Air Force Material Command, USAF, under agreement number F30602-97-1-0139. Liu is partially supported by DARPA and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-00-2-0575. Jie Li is supported in part from the Ministry of Education, Science, Sports and Culture of Japan under Grands-in-Aid for Scientific Research (C) (No. 11680410).

data items. The basic idea is when T_1 runs first allowing T_2 to access the data items locked by T_1 without being blocked even if there are lock conflicts. Later, a transient old version of some data item written by T_2, is given to T_1 for read purpose, thus T_1's reads are not blocked by T_2's writes. Why is this so? As long as T_1 will not ask for write locks on the data items accessed by T_2, T_2 can appear after T_1 in a serialization of the schedule. Our approach mitigates the lock contention not only between update and read-only transactions but also between update and update transactions, needs only two lock modes, read and write, avoids the extra deadlock overhead of 2V2PL substantially (especially for two-way deadlocks), and can be easily extended to distributed database environments.

The paper is organized as follows. Section 2 presents the database model and the correctness criteria used in the paper. We present our concurrency control protocol in Section 3. In Section 4 we propose a model to evaluate the performance of the protocol. Section 5 concludes the paper.

2 The Model

A *database* is a set of data *items*. A *transaction* T_i is an ordered pair ($\sum_i, <_i$), where $\sum_i$ is the set of operations in T_i, and $<_i$ indicates the execution order of those operations. A read(write) operation executed by a transaction T_i on item x is denoted as $r_i[x]$ ($w_i[x]$). The execution of a transaction must be *atomic*, that is, a transaction either *commits* (denoted c_i), with all its changes being applied to the database, or *aborts* (denoted a_i), with all its changes being discarded. Consider a set of committed transactions $\mathbf{T} = \{T_1, T_2, ..., T_n\}$, the execution of $\mathbf{T}$ is modeled by a structure called a *history*. A history H over $\mathbf{T}$ is a partial order ($\sum, <_H$), where $\sum$ is the set of all operations executed by transactions in $\mathbf{T}$, and $<_H$ indicates the execution order of those operations*.

2.1 Multiversion Histories

We consider a multiversion database in which each write operation on an item x produces a new *version* of x. The versions of x are denoted as x_i, x_j, ..., where the subscript is the *version number* of each version. The version number often corresponds to the index or the transaction number of the transaction that wrote the version.

A *multiversion* (MV) history H represents the sequence of operations on the versions of items. Thus, each $w_i[x]$ in an MV history is mapped into $w_i[x_i]$, and each $r_i[x]$ into $r_i[x_j]$, for some j. A transaction T_i *reads x from* T_j in H if T_i reads a version of x produced by T_j. Two MV histories over $\mathbf{T}$ are *view equivalent* if they have the same operations (thus they have the same reads-from relationships)[3].

An MV history H is *serial* if for every two transactions T_i and T_j that appear in H, either all of T_i's operations precede all of T_j's or vice versa. A serial MV history is *one-copy 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. An MV history is *one-copy serializable* if it is view equivalent to a one-copy serial MV history.

Two operations in an MV history *conflict* if they operate on the same version and one is write. The *serialization graph* of an MV history H, denoted $SG(H)$, is a directed graph whose nodes are transactions and whose edges are all $T_i \rightarrow T_j$ such that one of T_i's operations precedes and conflicts with one of T_i's. In order to determine whether an MV history H is one-copy serializable, a *multiversion serialization graph* of H, denoted $MVSG(H)$, can be built and analyzed. $MVSG(H)$ is $SG(H)$ with additional edges such that the following conditions holds: (1) For each item x, $MVSG(H)$ has a total order (denoted $\ll_x$) on all transactions that write x, and (2) For each item x, if T_i *reads x from* T_j and if $T_j \ll_x T_k$, then $MVSG(H)$ has an edge from T_i to T_k; otherwise, if $T_k \ll_x T_j$, then $MVSG(H)$ has an edge from T_k to T_i. The additional edges are called *version order edges*. An MV history H is one-copy serializable if $MVSG(H)$ is acyclic[3].

2.2 Can Follow

Transaction T_i *can follow* transaction T_j if T_j does not read items written by T_i, and they write (including updates as well as *blind-writes*†) different sets of items. More formally

Definition 1 Transaction T_i *can follow* transaction T_j if (a) $T_i.writeset \cap T_j.readset = \emptyset$, and (b) $T_i.writeset \cap T_j.writeset = \emptyset$ no matter on which database state they are executed.

The can-follow relationships between transactions can be captured in several ways. For example, in a variety of real world *canned* systems where there are limited number of transaction classes, can-follow relationships can be captured using transaction profiles and input parameters before transactions are executed [1].

There are some properties of can follow:
1. If T_i can follow T_j and T_j can follow T_i, then none of T_i's operations conflict with T_j's operations in any conflict serializable history.
2. The fact that T_i can follow T_j and T_j can follow T_k does not imply that T_i can follow T_k.
3. Read-only transactions can follow any transaction.

*[3] defines histories over all transactions (committed and aborted) and later introduces the notion of *committed projection* of a history for committed transactions. Since we are primarily interested in committed transactions, we define a history over committed transactions directly.

†Transaction T_i blindly writes item x if T_i writes x without first reading it.

Can follow relationships can be exploited to improve concurrency. Consider the following two transactions:

$$T_1: \ r_1[x]r_1[z]r_1[y]w_1[z]$$
$$T_2: \ r_2[x]w_2[x]r_2[y]w_2[y]$$

In 2PL, after $r_1[x]r_2[x]$ is executed $w_2[x]$ will be blocked until T_1 gets all the locks it needs, and then releases the lock on x. Since T_1 can follow T_2, we can allow T_2 to read the items write-locked by T_1 without the need to worry that T_2 may read an item which will later be written by T_1; And if T_1 later wants to read an item which has already been written by T_2, i.e., T_1 wants to read item y after the execution of $r_1[x]r_2[x]w_2[x]r_1[z]r_2[y]w_2[y]$, then the old value of y before $w_2[y]$ was executed can be given to T_1 to ensure the correctness of the execution.

The type of correct executions shown in the above example can be specified by the following MV history (we assume the initial value of each item is blindly written by transaction T_0):

$$r_1[x_0]r_2[x_0]w_2[x_2]r_1[z_0]r_2[y_0]w_2[y_2]r_1[y_0]w_1[z_1]c_1c_2$$

It is clear that the history is one-copy serializable.

3 The Protocol

Our protocol is built on top of 2PL [4]. For simpler presentation and to ensure recoverability, we assume the mechanism underlying our protocol is *strict* 2PL where 2PL is restricted to release all its locks at commit point, although our protocol can be built on top of 2PL directly. The two version algorithm maintains at most two versions for an item, but it may delay the commitment of some update transactions. The multiversion algorithm needs more space but it gives users more concurrency. Some issues about how these algorithms can be extended to distributed database environments are also addressed.

3.1 Two Version Algorithm

We maintain a *can-follow* set for each active transaction T_i, denoted $T_i.can\text{-}follow$, to keep the IDs of the set of active transactions whose can-follow relationships with T_i are of interest to T_i. (The can-follow set of each active transaction is initialized to be empty.) Although $T_i.can\text{-}follow$ can contain every active transaction that can follow T_i and arrives at the system earlier than T_i, $T_i.can\text{-}follow$ can also contain a subset of the active transactions that can follow T_i without harming the correctness. This nice property enables the DBMS to flexibly trade-off between the improved concurrency (due to can-follow concurrency control) and the corresponding (extra) overhead of the protocol.

In the two version algorithm, each item x has at most two versions. When x has two versions, one represents the last committed value of x, we call it the *old* version of x; the other represents the uncommitted value of x, we call it the *current* version of x. The two version algorithm works as follows:

(a) Except for the following specifications, every operation is scheduled according to strict 2PL, and the old version of each item, if any, can be neglected.

(b) When a transaction T_i arrives, each active transaction T_j whose can-follow relationship with T_i is of interest to T_i is checked: if T_j can follow T_i but T_i can not follow T_j then $T_i.can\text{-}follow = T_i.can\text{-}follow \cup \{T_j\}$. See that if T_j can follow T_i and vice versa then they do not conflict.

(c) When T_i wants to write an item x:

(c.1)If there are no locks on x, then T_i can lock x. And if $T_i.can\text{-}follow$ is not empty, then before T_i produces a new version of x, the old version must be kept.

(c.2) If there are only some read locks on x and each of these locks is held by a transaction in $T_i.can\text{-}follow$ then T_i can lock x. Before T_i produces a new version of x, the old version must be kept, and all the read locks associated with the old version must be kept too.

(d) When T_i wants to read an item x, if there is a transaction T_j holding a write lock on x and $T_i \in T_j.can\text{-}follow$, then T_i can be given the old version of x and a read lock is applied on the old version (See that the old version is always available in this situation).

(e) T_i can not release any write lock (thus can not commit) if $T_i.can\text{-}follow$ is not empty (this is referred as the *lock release* rule).

(f) When T_i commits, the ID of T_i is removed from every can-follow set with T_i in it. After T_i releases an write lock on item x, if the old version of x exists, it is discarded.

(g) When T_i aborts, the ID of T_i is removed from every can-follow set with T_i in it; In addition, for each of its write operation $w_i[x_i]$, if the old version of x is already there then the undo operation can be done by restoring the old version together with all the read locks associated with it. For each of its read operation, the read lock on either a current version or an old version (if any) is released.

Note that the two version algorithm may suffer from a deadlock while attempting to release locks. To illustrate, a transaction T_i can not release any write lock if $T_i.can\text{-}follow$ contains T_j according to the lock release rule. At this point, if T_j wants to acquire a write lock held by T_k and T_k wants to acquire a write lock held by T_i, then there is a deadlock. Fortunately, these deadlocks can be easily detected when T_i plans to commit by employing standard deadlock detection algorithms [6, 11]. Compared with (strict) 2PL the main difference is that transactions may be blocked when releasing locks. The correctness of the two version algorithm is specified by the following theorem.

Theorem 1 Every history H produced by the two version algorithm is one-copy serializable.

Proof: (*Sketch*) Since the two version algorithm is built on top of strict 2PL, for each item x $MVSG(H)$ can take the commit order as the total order $\ll_x$ on all transactions that write x. And we can prove as follows that all edges in $MVSG(H)$ are in the commit order: (i) 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. Since in the algorithm T_j can only read a committed value, so T_i commits before T_j. (ii) Consider a version order edge induced by $r_i[x_j]$, $w_j[x_j]$, and $w_k[x_k]$. If $T_j \ll_x T_k$, then if T_i has never been in $T_k.can\text{-}follow$ then T_i commits before T_k according to strict 2PL; otherwise, T_i commits before T_k according to the lock release rule. If $T_k \ll_x T_j$, then T_k commits before T_j and T_j commits before T_i. Since the commit order is acyclic by definition, $MVSG(H)$ is acyclic too, hence H is one-copy serializable. □

3.2 Multiversion Algorithm

In the two version algorithm, the lock release rule can cause concurrency loss in some situations. For example, assume there are several short update transactions, i.e., T_1, T_2, ... , T_n, running together with a long-duration read-only transaction T_r (assume T_r arrives first); Each update transaction writes a subset of items read by T_r, and the subsets may overlap with each other. According to the lock release rule, although these update transactions can write those items on which T_r holds read locks (since T_r can follow them), they can not release locks thus T_2 can not update what T_1 writes until T_r's termination.

The concurrency loss in the above example can be saved by allowing one item to have more than two versions. At this situation, T_1 can release locks before T_r terminates and T_2 can then update what T_1 writes. Moreover, if later T_r wants to read an item x updated by both T_1 and T_2, the value before x was updated by T_1 can be given. In summary, a DBMS keeping at most N versions for an item x allows up to N-1 transactions to update x during the execution of a transaction T_i which can follow these transactions.

We maintain multiple versions of an item in a stack similar to the stack used in [7]. Each substructure (or entry) of the stack contains the following information of a version: (1)**TRN**, the ID of the transaction which created this version, and (2)**PTR**, a pointer to the data item. A TRN value of 0 indicates that the item has not been updated yet in the current execution, i.e., the initial value of the item.

The multiversion algorithm works as follows. For brevity, only the modifications to the two version algorithm are specified. Note that the correctness can be proved in a way similar to the two version algorithm.
(a) When T_i wants to write an item x, and x has more than two versions, assume the top version of the version stack was produced by T_j, then if there are less than N versions kept in the stack, $T_j.can\text{-}follow \subseteq T_i.can\text{-}follow$, and there are no read locks on the top version held by other transactions, then T_i can write x, and the version produced by T_i is pushed into the stack. Otherwise, the write is blocked.
(b) When T_i wants to read an item x and x has more than two versions:
(b.1) If T_i can follow every active transaction whose version is in the stack, the bottom version is given. And a read lock is applied on it.
(b.2) If T_i reads x in order to update x, then if $T_j.can\text{-}follow \subseteq T_i.can\text{-}follow$ where T_j is the TRN of the top version, and there are no read locks on the top version, then the top version is given and a read lock is applied on it.
(b.3)Otherwise, the read is blocked.
(c) T_i can not commit if $T_i.can\text{-}follow$ is not empty or there exists an item x such that in the version stack of x there is a version produced by a transaction T_j which is below the version produced by T_i and T_j is still active. When T_i commits, the old version below the version produced by T_i in any version stack, if any, is discarded.
(d) When T_i aborts, when a version produced by T_i is removed from a version stack, appropriate cascading aborts may need be enforced.

The multiversion algorithm may also suffer from deadlock in two situations (and both can be detected by standard deadlock detection algorithms): (1) a deadlock can be caused when a transaction wants to commit. To illustrate, assume in the version stack of x the version produced by transaction T_j is above that of T_i, and assume T_i wants to read an item y write-locked by T_j, then there is a cycle of transactions waiting for each other to terminate. As a result, T_i or T_j has to be aborted. These cycles, however, can easily be tested when T_j wants to commits by employing standard deadlock detection algorithms[6, 11]. (2) A deadlock can be caused when a transaction whats to write an item. To illustrate, when T_i wants to write x and x has more than two versions, if T_i can write x without the condition $T_j.can\text{-}follow \subseteq T_i.can\text{-}follow$ held, i.e., there is a transaction T_k in $T_j.can\text{-}follow$ but not in T_i's, then there can be a deadlock when T_k wants to read x. At this point, T_k waits for T_i's termination in order to read x, T_i waits for T_j's termination in order to commit, and T_j waits for T_k's termination in order to commit.

3.3 Distributed Algorithm

We consider a distributed transaction processing system where strict 2PL is enforced at each site[‡]; two phase commit (2PC) is used to ensure the atomicity of distributed transactions; and data are not replicated. Our methods can be extended to such a system by applying the following mechanisms. In this paper, detailed presentation is omitted in favor of the following sketch.
(a) If transaction T_i can follow T_j, then each subtransaction of T_i (denoted T_{ik}) can follow every subtransaction of T_j (denoted T_{jl}), no matter how these subtransactions are generated.

[‡]Strict 2PL is necessary in distributed transaction processing to ensure global serializability.

(b) When a global transaction T_i arrives at the system, a timestamp $t(T_i)$ is generated in order to determine the arriving order of transactions.
(c) At each site, when a subtransaction T_{ik} arrives every active subtransaction T_{jl} at the site is checked: if T_j can follow T_i but not vice versa and $t(T_j)$ is earlier than $t(T_i)$ [§], then $T_{ik}.can\text{-}follow = T_{ik}.can\text{-}follow \cup \{T_{jl}\}$. Note that the can-follow set of a subtransaction at this site has nothing to do with the active transactions which have no subtransactions running at the site.
(d) The scheduler at each site can schedule subtransactions as if they were transactions in centralized systems except that strict 2PL need be enforced globally. As a result, in the two version algorithm T_{ij} can not release locks if there is a subtransaction T_{ik} such that $T_{ik}.can\text{-}follow$ is not empty.

Combining the above mechanisms with either the two version algorithm or the multiversion algorithm can ensure that the histories generated by the distributed algorithm are globally one-copy serializable. The correctness can be proved in a way similar to centralized systems.

4 Performance Analysis

We compare the throughput of can-follow concurrency control and the throughput of 2PL. We adopt the transaction model, the database access model, and the performance model described in [9]. [9] shows that these (to some extent simplified) models are quite useful in comparing the performance of concurrency control methods. In particular, we assume all transactions have the same length (i.e., they all access k items); locks are requested on demand and released upon termination. (For brevity, other description of these models are omitted.) We assume that there are D items in the database; and M transactions in the system (active or blocked). The throughputs of both our protocol and 2PL are: $T(M) = M/R(M)$, where $R(M) = (k+1)s(\bar{M}_a) + kP_cW$. Here $R(M)$ is the mean transaction response time; $\bar{M}_a$ is the mean number of active transactions in the system; $s(\bar{M}_a)$ is the mean processing time of a transaction step with $\bar{M}_a$ active transactions in the system; P_c is the probability of lock conflict per lock request; and W is the mean transaction waiting time due to a lock conflict. Note that although some commitment delay could be caused by the lock release rule, since every transaction has the same length and every transaction in $T_i.can\text{-}follow$ arrives at the system earlier than T_i, the impact of the lock release rule on $R(M)$ can be omitted in this performance analysis model where transactions in $T_i.can\text{-}follow$ should commit before T_i (in most cases).

It is easy to see that $\bar{M}_a$ (thus $s(\bar{M}_a)$) of our protocol is smaller than that of 2PL since our protocol can make some transactions blocked in 2PL not blocked any more. In the following, we show that the P_c value of our protocol is also smaller than that of 2PL. As a result, the throughput of our protocol should be strictly higher than that of 2PL (based on the assumption that the values of W are roughly the same in both protocols).

(a) Let $\bar{L}$ be the mean number of locks held per transaction, and N_L be the total number of locks held in the system, then $N_L = M\bar{L}$. According to [9], when all locks are exclusive and $k \ll N_L$ the probability of lock conflict (p_c) per lock request is: $p_c \simeq N_L/D$. When there are some shared locks, let the probability that a lock request is for a write-lock be b, then the number of items write-locked is $N_L^w = bN_L$, and the number of items read-locked is: $N_L^r = D(1-(1-\frac{1}{D})^{(1-b)N_L}) \simeq (1-b)N_L$. Here we make the approximation that $((1-b)N_L/D)^2 \simeq 0$. Therefore on average each read-locked item has approximately 1 lock associated with it. According to[9], in this situation, $p_c \simeq (N_L^w + bN_L^r)/D$.

(b) In our protocol, for a transaction T_i, let Mr_c (r_c is called the *can-follow ratio*) be the mean number of transactions which T_i can follow but not vice versa; then the mean number of transactions which can follow T_i but not vice versa is also Mr_c. It should be noticed that for two transactions T_i and T_j, when T_j arrives first, even if T_i can follow T_j (but not vice versa) the lock contention between T_i and T_j can not be mitigated. Assume that in this situation the probability that T_i arrives first is 0.5, then only half of the Mr_c transactions that T_i can follow can have T_i in their can-follow sets; and only half of the Mr_c transactions that can-follow T_i can have their IDs in $T_i.can\text{-}follow$.

(c) For a transaction T_i let r_o be the fraction of the reads which will not later be upgraded to a write in all T_i's reads. And assume r_o is identical. Then in the two version algorithm when all can-follow relationships are used the probability of lock conflict (per lock request) is

$$
\begin{aligned}
& p_c = \\
& \text{prob(write-write conflict} \mid \text{write request) prob(write request)} \\
& + \text{prob(read-write conflict} \mid \text{write request) prob(write request)} \\
& + \text{prob(write-read conflict} \mid \text{read request) prob(read request)} \\
& \simeq \frac{bN_L^w}{D} + \frac{bN_L^r}{D}\Big(1 - \frac{\frac{Mr_c}{2}\bar{L}(1-b)r_o}{N_L^r}\Big) \\
& + \frac{(1-b)N_L^w}{D}\Big(1 - r_o \times \frac{\frac{Mr_c}{2}\bar{L}b}{N_L^w}\Big) \\
& = \frac{1}{D}\Big(N_L^w\Big(1 - \frac{(1-b)r_cr_o}{2}\Big) + bN_L^r\Big(1 - \frac{r_cr_o}{2}\Big)\Big)
\end{aligned}
$$

The above performance analysis of the two version algorithm can be easily extended to the multiversion algorithm which usually achieves more concurrency. In these two algorithms, the degree of concurrency improvement depends on the values of the parameters used above which usually vary from one system to another.

5 Discussion and Conclusion

In addition to improving concurrency, our protocol has the following properties: (1) our protocol enables the

[§] Note that $t(T_j)$ may be later than $t(T_i)$. In this situation, since T_{jl} may have already written an item which will later be read by T_{ik} without keeping the old value, T_{ik} can no longer be put into $T_{jl}.can\text{-}follow$.

DBMS to flexibly tradeoff between the improved concurrency and the corresponding (extra) overhead by allowing partial can-follow concurrency control. (2) The histories generated by our protocol are recoverable. In particular, the two version algorithm is not only recoverable but also can avoid cascading aborts [3], although the multiversion algorithm is only recoverable. Moreover, we can do logging in the same way as in single version systems to ensure that system crashes are recoverable. (3) Version selection in our protocol does not require the analysis of a dependency graph.

In distributed environments, since it is possible that two subtransactions at a site may have the can-follow relationship although their corresponding global transactions do not, we may be able to further improve concurrency by exploiting the can-follow relationships between subtransactions. This is a topic for future research.

References

[1] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 2001. To appear.

[2] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. *ACM Transactions on Database Systems*, 5(2):139–156, June 1980.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[4] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notion of consistency and predicate locks in database systems. *Communications of the ACM*, 19(11):624–633, 1976.

[5] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[6] R.C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, 1972.

[7] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 124–133, San Diego, CA, June 1992.

[8] R.E. Stearns and D.J. Rosenkrantz. Distributed database concurrency controls using before-values. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, Ann Arbor, April 1981.

[9] Y.C. Tay, N. Goodman, and R. Suri. Locking performance in centralized databases. *ACM Transactions on Database Systems*, 10(4):415–462, December 1985.

[10] A. Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, March 1998.

[11] M. Yannakakis, C.H. Papadimitriou, and H.T. Kung. Locking protocols: Safety and freedom from deadlock. In *Proceedings of the Twentieth IEEE Symposium on Foundations of Computer Science*, 1979.