A Concurrency Control Scheme for Nested Transactions

D.C. Shin and S.C. Moon

Department of Computer Science
Korea Advanced Institute of Science and Technology
Cheongryang P.O. Box 150, Seoul, Korea

Locking rules for concurrency control in nested transaction model, in which parent/child parallelism, sibling parallelism, and arbitrary commitment-dependence relationship between transactions are allowed, are presented. The locking rules are based on the lock transformation scheme that never allows the commit deadlock to occur, which could happen by permitting parent/child parallelism. In addition, two proposed locking rules are compared to each other with respect to the overhead involved in cascading abort, the degree of concurrency, and the frequency of deadlock occurrences.

1. INTRODUCTION

1.1. Background

The notion of transaction, which is viewed as a unit of consistency and recovery, has been used for many data processing applications to relieve the burden of application programmers in coping with concurrent execution and failures. In this respect, a transaction should be atomic, i.e., an entire transaction should be executed in whole or no effects of its execution should remain. The properties such as atomicity, consistency, isolated execution, and durability, which transactions should satisfy, are defined in [5]. Many techniques to preserve the properties of transaction have appeared in the literature [1, 12].

In conventional data processing applications, it is reasonable to claim that a simple and short transaction, which usually terminates its execution within a few seconds, should preserve the properties of transactions, since the overhead to preserve them are more or less low. For this kind of transactions, the overhead of waiting or restart that is incurred by sharing the same data items among transactions may not be significant, and recovery cost due to transaction failure may also not be significant. Execution of complex and long transactions, however, might induce relatively long waiting, more frequent restarts, and increased recovery cost. In other words, flat transactions (or single-level transactions) cannot provide enough flexibility and efficiency for complex applications. Since transactions in CAD applications usually have long duration and include decision-making of many different designers, a different transaction model is preferred [7]. Transaction processing mechanism for long-lived transactions, which could significantly cause the termination of shorter transactions to be delayed by holding data items for long period of time, was proposed in [2].

To overcome the deficiency of flat transactions, it has been recommended to have a finer grained concurrency control and recovery scheme by imposing hierarchical relationship between transactions [9, 10]. One of this kind of structure is the nested transaction. A nested transaction consists of not necessarily sequential, but concurrent subtransactions. In nested-transaction hierarchy, a transaction invokes subtransactions, each of which can then invoke other subtransactions, and so on.

Nested transactions have two major advantages over flat transactions: failure independence and intra-transaction parallelism [10]. Failure independence means that each subtransaction of a nested transaction may fail independently of each other or of the entire transaction, therefore it is not necessary to back out the entire transaction. Intra-transaction parallelism fully takes advantage of the potential concurrency between transactions, thereby increasing the performance.

1.2. Some Terminologies

In nested transaction model, the nesting hierarchy between transactions can be represented by a transaction tree, in which the nodes and the edges represent transactions and the nesting relationships between transactions, respectively. Since the relationship between transactions is represented by a transaction tree, we will strictly follow the terminologies used in the conventional tree structure.

A transaction that has its subtransactions are called parent, and its subtransactions are called children. The top-level transaction (TL-transaction) is a root transaction which does not have parent. Transactions which does not have child are called terminal, and otherwise they are called non-terminal. Note that a descendant and an ancestor of transaction T include T itself, i.e., the ancestor and descendant are defined to have reflexive property, whereas the superior and inferior are defined to have irreflexive property.

1.3. Motivation

The concurrency control algorithms for nested transactions have been proposed by a few researchers in the literature [6, 10, 11]. In [11], multi-version timestamp concurrency
control scheme was extended to make concurrent execution of nested transactions serializable. The problems in extending the two-phase locking protocol to nested transactions were discussed in [4]. To allow for early release of locks, a particular kind of nested transactions, multi-level transactions, was proposed in [14]. Locking-based algorithms for concurrency control in nested transactions are described in [6, 10]. In [10], two-phase locking scheme was extended to nested transaction model in which it is assumed that the actual operations on data items are performed only by terminal transactions and coordination and supervisory functions are performed by non-terminal transactions. In [6], the assumption used in [10] was relaxed and the locking rules of [10] were extended to the generalized nested environments. The work of [6, 10] are described in detail in the next section, Related Work.

The nested transaction model of [10] is limited in the sense that all operations on data items are performed only by terminal transactions and only read/write locks are considered. However, to fully exploit the potential concurrency between transactions, relaxation of restrictions on relationships between transactions as well as on operations of transactions are necessary. In other words, the efficient concurrency control scheme for the generalized nested transaction model that allows high degree of parallelism between transactions is required. Therefore, in this paper parent/child parallelism and sibling parallelism are allowed to get high degree of parallelism.

In this paper, we propose new locking rules for nested transactions on basis of lock transformation scheme, which is also new. The lock transformation scheme is considered to be appropriate for the generalized nested transaction model which impose no restrictions on the relationships between transactions and on operations performed by transactions. Our model for nested transaction allows high parallelism between transactions, which exploits the potential concurrency of transactions including subtransactions in nested transactions. Finally, a brief comparison of the two proposed locking rules is made in terms of measures that affect the performance of concurrency control scheme.

The remainder of this paper is organized as follows. Section 2 describes the related work. The nested transaction model used in this study and the lock transformation scheme are described in Section 3. Two locking rules based on the lock transformation scheme are presented and compared in Section 4. Finally, conclusions appear in Section 5.

2. RELATED WORK

In this section, the locking-based algorithms for concurrency control of nested transactions are discussed. Among many related work, the locking protocols in [6, 10] are chosen to be presented here since they are the two major locking-based schemes that can be compared against our new schemes.

2.1. Moss's Upward Inheritance Scheme

A nested transaction model for a high-reliable, distributed computing environment was presented in [10]. In this section, only the synchronization technique for concurrency control will be addressed.

Moss distinguished two kinds of locks: hold lock and retain lock. If a lock on data item D is held by transaction T, only T has the right to access to D. If a lock on D is retained by T, a lock request on D can be granted only to T's inferiors regardless of the lock mode; if a transaction that is not the inferior of T requests D with a conflicting lock mode with T's retain lock, the request will be denied. When T commits, T's parent retains all locks held and/or retained by T. This lock movement from a child to its parent is so called upward inheritance. When T aborts, all locks that are currently held and/or retained by T are released. Moss assumed that every access to data item is made only by the terminal transactions and coordination and supervisory functions are performed only by non-terminal transactions. Furthermore, he did not consider the general lock modes i.e., he considered only read and write locks.

The above assumption on restricting execution of operations on data items only to terminal transactions is too restrictive and inefficient. Moss pointed out that a critical problem that must be solved, which will be defined as commit deadlock in Section 3, will arise if this assumption is relaxed, but he addressed no efficient solutions.

The deadlock detection algorithm in [10], of course, can deal with the commit deadlock. However, permitting commit deadlocks to occur and resolving them in the same way as the conventional deadlock might be inefficient, since the commit deadlock between the superior and the inferior always occurs whenever the inferior requests the lock held by the superior.

2.2. Haerder and Rothermel's Downward Inheritance Scheme

A concurrency control scheme that attempts to achieve a high degree of intra-transaction parallelism within nested transactions by using locking protocols has been presented in [6]. The scheme allowed concurrent execution between parent and its children, and ruled out the assumption made in [10], which is that terminal transactions can only perform operations on data items. He proposed a locking scheme that allows downward inheritance of locks. In downward inheritance scheme, a transaction may be inherited the locks of its superiors in the same lock mode, and further its children can also inherit its (inherited) locks, and so on. For example, a transaction holding a lock in mode M can inherit its lock by retaining a lock at itself in the same mode. Whether to do downward inheritance and when to do it are entirely dependent on explicit, arbitrary decision of the current lock holder.

In order for a transaction to be able to control the lock mode that will be inherited downwards to its inferiors, they also extended their scheme to allow upgrading as well as downgrading locks. After transaction T downgraded a lock from mode $M_i$ to a weaker mode $M_j$, T retains the lock in mode $M_j$ and at the same time it still holds the lock in mode $M_i$. Since T still holds the lock in mode $M_i$, T prevents its inferiors from holding the lock in a mode conflicting with $M_i$. 
3. LOCK TRANSFORMATION SCHEME

3.1. Nested Transaction Model

This section describes the nested transaction model used in our study. Nested transaction model in [10] are restrictive in the sense that operations on data items are performed only by terminal transactions, excluding concurrency between a parent and its children. On the contrast, our model of nested transactions is generalized to fully take advantage of the potential concurrency inherent in the application by allowing parent/child parallelism as well as sibling parallelism. Therefore, it is possible that all descendants of a TL-transaction may execute concurrently. Allowing concurrent execution of transactions including subtransactions may cause a commit deadlock to occur, which must be resolved. We now define the commit deadlock clearly.

Definition 1: In the nested transaction model that allows concurrent execution between transactions including subtransactions, we say that commit deadlock occurs if an inferior requests an access to the data item that its superior has already locked.

Since the inferior cannot acquire the requested lock until its superior commits and subsequently releases the lock and the commitment of the superior is possible only after the inferior has committed, a deadlock really can occur. The lock transformation scheme described in Section 3.2 illustrates how to prevent the commit deadlock.

In addition to the intra-transaction parallelism, our model allows the arbitrary commitment-dependence relationship between a parent and its children, which was not considered in [6]. The idea of introducing this is that it is useful for a child to commit independent of its parent for some applications. For example, in a particular system such as naming system, a name lookup may cause information associated with the name to be copied from one location to another to speed up subsequent lookups for that name. It may be useful that the information copied from one location to another by a child remains in effect by not aborting the child even if its parent happens to abort [8]. In this case, the child can be considered as another TL-transaction, and the commitment-dependence relationship between the child and its parent no more exists. The concept related to this commitment relationship was also discussed in [13].

Definition 2: If transaction T is allowed to commit only after all children of T have already committed, T's commit sphere consists of T itself and the transactions in the children's commit spheres. T's commit sphere consists of the set of inferiors which should commit in order for T to commit. The commit sphere of a parent is determined by a particular commitment-dependence relationship between the parent and its children. Note that the commit sphere of T does not always include all its children. In other words, if transaction T is allowed to commit independent of the fate of its particular child, this child is excluded from T's commit sphere.

Example 1: Consider a transaction tree shown in Figure 1. TL-transaction A has its children B and C, B in turn has its children D and E, and E in turn has its children F, G, and H. If the commitment of A depends on B but is independent of C, A's commit sphere is composed of A itself and transactions in B's commit sphere; in this case, C is excluded from A's commit sphere. Further, suppose that E's commit sphere is composed of E, F, G, and H, and commitment of B depends on D and E. Then, B's commit sphere is composed of B, D, and transactions in E's commit sphere.

3.2. Lock Transformation Scheme

Since decomposition of transactions into a set of cooperating subtransactions or synthesis of a number of previously existing transactions into a new transaction is often made in nested transaction systems, the time gap between termination of execution of a transaction and beginning of its commitment may exist. We say that transaction T terminates its execution when T finishes its execution successfully; thereafter T can begin its commitment after all transactions included in T's commit sphere have already terminated. Example 2 explains the time gap.

Example 2: Suppose that transaction A's commit sphere includes transactions B and C (Figure 2). Further suppose that A, B, and C start (terminate) their executions at \( t_1 \), \( t_2 \), and \( t_3 \) (\( t_4 \), \( t_5 \), and \( t_6 \)), respectively. Parent A terminates its execution at \( t_4 \), but A's commitment can begin only after B and C, which are included in A's commit sphere, have already terminated, i.e., after \( t_5 \).
inheritance but also upward lock inheritance, since any child(parent) is allowed to acquire the locks released by its parent(child). To show that the commit deadlock can be prevented by early release of locks, consider Example 2. Assume that B requests a lock held by its parent A before \( t_1 \). In this case, A's lock release at \( t_1 \) obviously prevents the commit deadlock, which will occur without the lock release. In this case, no problem arises because A requires no additional locks after terminating its execution, hence the serializability is preserved.

Before presenting the lock transformation scheme, let's define a meaningful terminology for the lock transformation scheme.

**Definition 3**: Let MODES be a set of lock modes. For \( X \) and \( Y \in \text{MODES} \), assume that \( X \) and \( Y \) are compatible with any lock modes \( Z_1 \) and \( Z_2 \) in MODES, respectively. Let \( M_1 \) and \( M_2 \) be a subset of MODES such that \( M_1 \) and \( M_2 \) are just composed of a set of any such \( Z_1 \)'s and a set of any such \( Z_2 \)'s, respectively. Then, \( Y \) is more restrictive than \( X \) if \( M_1 \) is a proper subset of \( M_2 \).

According to the above definition, \( \text{write(W)} \)-lock is more restrictive than \( \text{read(R)} \)-lock, since an \( R \)-lock is compatible with other concurrent \( R \)-lock, whereas a \( W \)-lock is neither compatible with an \( R \)-lock nor compatible with other concurrent \( W \)-lock.

When transaction \( T \) terminates its execution, the lock on data item \( D \) held by \( T \) is released if 1) \( T \) is not the unique holder of the lock and 2) lock mode on \( D \) of another transaction is more restrictive than the mode of \( T \)'s hold lock. Otherwise, the lock is transformed into offer lock with the same previous mode. Then, it is necessary to compute the new offer lock mode on \( D \) if another offer lock on \( D \) currently exists. From the newly-formed offer lock and the current offer lock, the more restrictive lock mode between the two is taken as the new offer lock mode on \( D \).

To explain the principle of lock transformation scheme, consider Example 3 which shows the simple case with \( R \)-lock and \( W \)-lock.

**Example 3**: Assume that \( T_1 \) that held \( R \)-lock on data item \( D \) terminates its execution. If transaction \( T_2 \) also currently holds \( R \)-lock on \( D \), \( T_2 \) releases the lock without transforming into offer lock since \( T_1 \)'s hold lock is no more restrictive than \( T_2 \)'s hold lock. In this case, \( T_2 \) will later transform the lock into offer lock when \( T_2 \) terminates its execution. On the contrast, if \( T_1 \) is the unique holder of lock on \( D \), the lock is transformed into offer lock with the same previous mode. Assume that another offer lock with \( W \)-lock mode on \( D \) currently exists. Then, since the newly-formed offer lock is \( R \)-lock and the current offer lock is \( W \)-lock, the new offer lock mode becomes \( W \)-lock.

**Definition 4**: The lock sphere of transaction \( T \) is composed of the set of transactions that are included in the commit sphere of \( T \)'s parent.

The lock sphere is used to determine to what extent offer locks are available. The offer locks of transaction \( T \) are always available to the transactions that are in the lock sphere of \( T \). Note that if \( T \) is TL-transaction, \( T \)'s lock sphere is just composed of the set of descendants of \( T \) included in \( T \)'s commit sphere. Consider, for example, Figure 1 again. \( E \)'s lock sphere is composed of \( B \), \( D \), \( E \), \( F \), \( G \), and \( H \). \( B \)'s lock sphere consists of all transactions in \( A \)'s commit sphere.

4. LOCKING RULES

A variety of applications have used locking scheme as the standard technique for concurrency control. In this section, new locking rules that are developed in this paper for the nested transaction model, which allow parent/child parallelism, sibling parallelism, and arbitrary commitment-dependence relationship between transactions, are presented. Our locking rules use the lock transformation scheme described in the previous section; they never use the lock inheritance scheme used in [6, 10].

Recall that offer locks of transaction \( T \) are always available to transactions that are included in \( T \)'s lock sphere. The degree of concurrency and recovery cost due to failure of \( T \) are mainly affected by the time when \( T \)'s offer locks become available to other transactions that are not in the lock sphere of \( T \) but are in the lock sphere of TL-transaction of \( T \); hereafter, we will call these transactions as rest transactions. Note that the lock sphere of TL-transaction includes all descendants in its commit sphere.

4.1. Locking Rules I

Before describing locking rules, let's define some fundamental terminologies used in this section. We say that a transaction precommits when all descendants that are included in its commit sphere terminate their executions. When a transaction precommits, the updates made by it are stored in stable storage so that the updates can survive in face of failures.

**Definition 5**: If transaction \( T_i \) is neither superior nor inferior of transaction \( T_j \), the least common ancestor (LCA) of \( T_i \) to \( T_j \) is defined to be the oldest one of \( T_i \)'s ancestors that is not one of the ancestors of \( T_j \). By oldest transaction we mean the highest possible transaction with respect to the level in transaction tree.

LCA of \( T_i \) to \( T_j \) is used to determine whether \( T_i \)'s offer locks are available to \( T_j \) or not. Note that LCA is defined between two transactions that are descendants of TL-transaction. For example, in Figure 1 LCA of \( D \) to \( F \) is \( D \) itself, whereas LCA of \( F \) to \( D \) is \( E \).

In Locking Rules I, granting the offer locks of transaction \( T_i \) to one of rest transactions, say \( T_j \), is delayed until LCA of \( T_i \) to \( T_j \) precommitted. That is, the offer locks of \( T_i \) can be available to \( T_j \) only after LCA of \( T_i \) to \( T_j \) precommitted. Example 4 explains the principle of Locking Rules I.

**Example 4**: Consider Figure 1. Suppose that \( F \) has the offer lock on data item \( R \) and \( D \) requests the lock on \( R \). Since \( F \) has the offer lock, \( E \), \( G \), and \( H \) can get the lock at any time. However, \( D \) will not get the lock unless LCA of \( F \) to \( D \), i.e. \( E \) precommitted.

In Example 4, the degree of concurrency may be decreased since \( D \) can get the lock only after \( E \) precommitted. However, abort of \( F \) due to some reason
such as abort of E never affects the fate of D, therefore cascading aborts do not occur in this case. Suppose D finally got the lock requested before. Then, abort of A due to abort of E after D got the lock will cause D to be aborted; note that in this case the abort of E might have been forced affected by abort of B after E precommitted. However, this abort of D incurs no severe penalty because D should be subject to be aborted according to the commitment-dependence relationship that might exist between B and D. Hence, in Locking Rules I unnecessary cascading aborts can be prevented, at the expense of sacrificing the degree of concurrency.

The locking rules are summarized as follows.

R1. A transaction T may acquire a lock on data item D with mode M, if 1) no other transaction holds a lock on D with incompatible mode with M and 2) T is included in the lock sphere of transaction T, that has the offer lock on D, if such T exists, or LCA of T to T has precommitted, if such LCA exists.

R2. When some transaction T except TL-transaction terminates its execution, the locks held by T are transformed into offer locks, according to the lock transformation scheme.

R3. When a transaction aborts, it releases all locks it currently holds and/or its offer locks, if any.

Note that only after all transactions that belong to the commit sphere of TL-transaction terminated their executions, TL-transaction can commit. After TL-transaction committed, all locks, i.e. hold locks and/or offer locks, that are currently possessed by the transactions in the commit sphere of TL-transaction are then released. Commitment of each transaction in the commit sphere of TL-transaction is performed by the two-phase commit protocol to ensure transaction atomicity [3].

4.2. Locking Rules II

As pointed out in Section 4.1, the time when offer locks of transaction T are granted to the rest transactions affects the degree of concurrency and recovery cost due to transaction failure. To increase the degree of concurrency, the offer locks of transaction T can be granted to rest transaction, say T, regardless of whether the least common ancestor of T to T precommitted or not. This is the main idea of Locking Rules II. Example 5 explains the principle of Locking Rules II.

Example 5: Consider Figure 1 again. Suppose that F has the offer lock on data item R and D requests the lock on R. With Locking Rules II, E, G, or H can get F’s offer lock. Unlike Locking Rules I, D can also get F’s offer lock. In other words, F’s offer lock can be granted to A, B, D, E, G, or H that are in the lock sphere of TL-transaction of F, i.e. A. In this case, unfortunately, abort of F may induce many cascading aborts.

In Locking Rules II, the degree of concurrency between transactions can be increased whereas recovery cost due to possible cascading aborts becomes high. To conform to the principle of Locking Rules II, R1 of Locking Rules I should be modified to R1', below, while R2 and R3 remain unchanged because they are irrelevant to the time when offer locks are granted to other transactions.

R1'. A transaction T may acquire a lock on data item D with mode M, if 1) no other transaction holds the lock on D with the incompatible mode with M and 2) a transaction T' in the lock sphere of T's TL-transaction has the offer lock on D, if such T' exists.

4.3. Comparison of the Two Locking Rules

In the methodological respect, the two locking rules can be distinguished by the choice of the time when offer locks of transaction T become available to the rest transactions. In Locking Rules I, by delaying the granting of T's offer locks to other transactions, unnecessary cascading aborts due to abort of T can be prevented. However, Locking Rules I cannot achieve as much concurrency as Locking Rules II can. On the other hand, in Locking Rules II, granting T's offer locks to other transactions without delay may induce cascading aborts in case T is aborted. However, Locking Rules II can achieve higher degree of concurrency than Locking Rules I.

In addition, two locking rules can also be compared in terms of frequency of deadlock occurrences. To compare them, consider Figure 1 again. Suppose that D waits for the lock held by F and G waits for the lock held by D. According to Locking Rules II, if the lock held by F is transformed into offer lock, D can acquire the lock immediately. Likewise, if the lock held by D is transformed into offer lock, G can also acquire the lock immediately. Hence, no deadlock occurs with Locking Rules II in this case. However, in Locking Rules I, in order for D to acquire the lock, the least common ancestor of F to D, i.e. E, should have already precommitted, which is impossible because G is still waiting for D and has not yet been terminated. Hence, deadlocks occurs among D, F, and G in this case with Locking Rules I. To resolve the deadlock among D, F, and G, it is necessary to force one of them abort.

4.4. Comparison to Past Work

As far as we know, Moss proposed locking rules for concurrency control in nested transactions for the first time in the literature. Moss's work was based on limited parent/child parallelism by allowing only terminal transactions to access data items. If this limitation is relaxed, Moss's locking rules become inefficient in that commit deadlock occurs whenever an inferior requests the lock held by its superior. Haerder and Rothermel's locking rules allow not only sibling parallelism but also parent/child parallelism. They also extended Moss's locking rules by introducing so called downward lock inheritance scheme.

Our locking rules used the lock transformation scheme with which the commit deadlock can be prevented, whereas Moss's work and Haerder and Rothermel's work used the lock inheritance scheme. Notice that the lock transformation scheme covers not only upward but also downward lock inheritance scheme. In this sense, our locking rules allow parent/child parallelism as well as sibling parallelism. In addition, the commitment-dependence relationship between parent and child was considered in our locking rules.

Except for inclusion of parent/child parallelism and commitment-dependence relationship between a parent and its children, the principle of Locking Rules I on granting locks may be considered the same as that of Moss's
locking rules. However, Locking Rules II used a totally new approach that fully exploits the properties of commit sphere.

5. CONCLUSIONS

We proposed two locking rules for concurrency control on basis of the lock transformation scheme in general nested transaction model, which allows a higher degree of parallelism than the previous work in the literature. The high degree of parallelism was achieved by allowing intra-transaction parallelism and arbitrary commitment-dependence relationship between a parent and its children.

To allow early release of locks is the basic principle of the lock transformation scheme. Use of the lock transformation scheme makes our locking rules immune from commit deadlock. Two proposed locking rules are based on the lock transformation scheme. Locking Rules I can avoid unnecessary cascading aborts, at the expense of sacrificing the degree of concurrency. Locking Rules II, however, may introduce cascading aborts by gaining the increased concurrency. Additionally, Locking Rules I is prone to deadlock more frequent than Locking Rules II.

REFERENCES