Speculation Based Nested Locking Protocol to Increase the Concurrency of Nested Transactions

P. Krishna Reddy and Masaru Kitsuregawa
Institute of Industrial Science, The University of Tokyo
7-22-1, Roppongi, Minato-ku, Tokyo 106-8558, Japan
{reddy, kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract

In this paper, we have proposed improved concurrency control protocol based on speculation for nested transactions and explained how it increases both intra- and inter-transaction concurrency as compared to Moss's nested locking protocol. In the proposed speculative nested locking (SNL) protocol, whenever a sub-transaction finishes work with a data object (produces after-image), it's parent inherits the lock. The waiting sub-transaction carries out speculative executions by accessing both before- and after-images of preceding sub-transaction and selects appropriate execution after the termination of preceding sub-transaction. In this way, SNL allows multiple executions to be carried out for a transaction by trading extra processing and main memory resources to increase concurrency.

Index terms Concurrency control, nested transactions, locking, serializability, transaction processing.

1 Introduction

The traditional transaction model, although suitable for conventional database applications such as banking and airline reservation systems, does not provide much flexibility and high performance when used for complex applications such as object oriented systems, long-lived transactions, or distributed systems. Nested transactions have been proposed [23] to overcome limitations of the flat transaction model. Nested transactions extend the notion that transactions are flat entities by allowing a transaction to invoke atomic transactions as well as atomic operations. They provide safe concurrency within transaction, allow potential internal parallelism to be exploited and offer an appropriate control structure to support their execution. Also, they provide finer control over failures by limiting the effects of failures to a small part of the transaction. This property is achieved by allowing transactions within a given transaction to fail independently of their invoking transaction. Nested transactions were implemented in system R [11], Argus [20], Clouds [6], Locus [24] and Eden [16], and are widely accepted as a suitable mechanism for reliable distributed transaction processing systems.

In nested locking (NL) concurrency control protocol proposed by Moss [23], each leaf-transaction follows two-phase locking (2PL) protocol [9, 10] for concurrency control. If a sub-transaction obtains a write lock, its parent inherits the lock only after its commit, as per 2PL rules. To access the locked data object, a (sub)transaction has to wait until termination of a lock holding transaction. Therefore, in nested transactions, data contention increases lock waiting time which decreases the throughput performance of the system. In this paper we propose speculative nested locking (SNL) protocol to increase concurrency by supporting multiple executions for a transaction with extra computing resources. In SNL whenever a sub-transaction T_i finishes work with a data object (produces after-image), it's parent inherits the lock. The waiting (sub)transaction accesses both before- and after-images of T_i and then carries out speculative executions. However, the order is maintained; i.e., the waiting transaction selects appropriate execution only after termination of T_i. As such, there is no limitation on the number of levels of speculation but this number depends on the system's resources, such as the size of main memory and processing power. In SNL, the number of speculative executions carried out by a transaction increases exponentially as data contention increases. The SNL approach trades available resources to increase concurrency.

As compared to NL, the SNL approach increases concurrency by allowing a sub-transaction to release the lock before termination without causing cascading aborts. (In this approach on termination of earlier transaction the waiting transaction drops invalid execution(s) and retains the valid one. However, this is different from aborting the entire transaction.) Further, if data objects accessed by a transaction are pre-declared, SNL increases both intra- as well as inter- transaction parallelism without violating

serializability criteria. Under simplified assumptions, we analyze the scope of SNL to increase concurrency among sub-transactions under limited resource environments.

The work is motivated by the fact that with the continual improvement in hardware technology, we now have systems with significant amounts of processing speed and main memory, but more time is spent by transaction waiting for data (both I/O and remote data) than performing actual computations. Consequently, a (sub)transaction keeps locks for longer times if Moss's NL is followed. As a result, throughput is decreased. Since the cost of both CPU and main memory is falling, we believe that extra processing power and memory could be added to the system at reasonable cost. The strength of SNL is that it offers the potential to increase concurrency by trading extra main memory and processing resources without violating seralizability as a correctness criteria. Also, the speculative processing is transparent to the user. (In this paper we are not considering extension of speculation to interactive transactions.) Since SNL is lock-based, it could be integrated with existing applications based on Moss's NL with little effort.

In the next section we discuss related work. In section 3 we explain nested transaction model and Moss's NL protocol. In section 4, we present the SNL approach, discuss its variants and explain the processing under limited resource environments. In section 5 we explain how SNL increases concurrency through an example. In section 6, we informally discuss the correctness of the SNL approach. In section 7, we perform concurrency analysis under simplified assumptions. In section 8, we discuss the performance issues concerning SNL. The last section consists of summary and conclusions.

2 Related work

Several protocols exist to synchronize the execution of nested transactions. Reed developed a time-stamp based technique for nested transactions [26]. In [23], Moss presented a concurrency control algorithm using 2PL for a nested transaction environment. In [25] theoretical framework has been presented to prove the serializability of synchronization protocols for nested transactions. In [21], overview of research in the area of nested transactions is given. In [14], a concept of downward inheritance is introduced to improve the parallelism within a nested transaction. In [22] the pre-write operation is introduced to increase concurrency in a nested transaction processing environment. This model allows some particular sub-transactions to release their locks before their ancestor transaction's commit. This allows other sub-transactions to acquire required locks earlier. However, it is assumed that once the sub-transaction pre-writes the value, it will not abort.

In the context of flat transactions, speculation has been employed in [2] to increase the transaction processing performance for real-time centralized environments that employ optimistic algorithms for concurrency control. In [4], a branching transaction model has been proposed for parallel database systems where a transaction follows alternative paths of execution in case of a conflict.In that paper the operation in limited resource environments is not analyzed. In [15] a proclamation-based model is proposed for cooperative environments in which a cooperative transaction proclaims a set of values, one of which a transaction promises to write if it commits. The waiting transactions could access these proclaimed values and carry out multiple executions. This approach is mainly aimed at cooperative environments such as design databases and software engineering. In [17], a transaction processing approach has been proposed for distributed database systems where a transaction releases locks after completing execution by employing static 2PL. In [18], speculation is employed to increase concurrency in mobile environments, with the assumption that a mobile host could support a reasonable number of executions.

The SNL approach is a lock based approach and proposed for nested transactions. Also, in the SNL approach, a transaction releases locks before execution and cascading aborts do not occur. In [19] we have discussed the idea of extending speculation to nested transactions. In this paper we have improved the algorithm, included examples and concurrency analysis.

3 Nested transactions and locking protocol

3.1 Nested transaction model

We employ $X, Y, \ldots$ to represent data objects. Transactions are represented by $T_i, T_j, \ldots$; where, $i, j, \ldots$ are integer values. In nested transaction model [23] a transaction may contain any number of sub-transactions, which again may be composed of any number of sub-transactions-conceivably resulting in an arbitrary deep hierarchy of nested transactions. The root transaction which is not enclosed in any transaction is called the top-level transaction (TLT). Transactions having sub-transactions are called parent transactions (PTs), and their sub-transactions are their children. Leaf-transactions (LTs) are those transactions with no children. The ancestor (descendant) relation is the reflexive transitive closure of the parent (child) relation. We will use the term superior (inferior) for the non-reflexive version of the ancestor (descendant). The children of one parent are called siblings. The set of descendants of a transaction together with their parent/child relationships is called the transaction's hierarchy. In the following, we will use the term 'transaction' to denote TLT, PT, and LT. The hierarchy of a top-level transaction (TLT) can be represented by a transaction tree. The nodes of the tree represent transac-

tions, and the edges illustrate the parent/child relationships between the related transactions. In the transaction tree shown in Figure 1, T_1 represents TLT or root. A children of sub-transaction T_3 are T_4, T_6, and T_7, and the parent of T_3 is T_2.

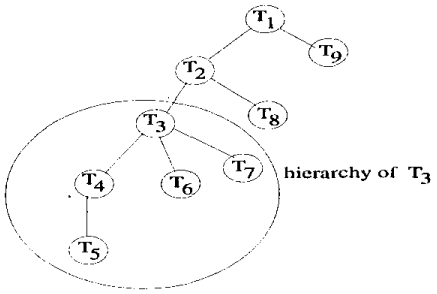

Figure 1. Example of a Transaction tree.

The properties defined for flat transactions are atomicity, consistency, isolated execution, and durability (ACID properties). In the nested transaction model, the ACID-properties are fulfilled for TLTs, while only a subset of them are defined for sub-transactions. A sub-transaction appears atomic to the other transactions and may commit and abort independently. Aborting a sub-transaction does not effect the outcome of the transactions not belonging to the sub-transaction's hierarchy, and hence sub-transactions act as fire-walls, shielding the outside world from internal failures. The durability of the effects of a committed sub-transaction depends on the outcome of its superiors. Even if a sub-transaction commits, aborting one of its superiors will undo its effects. A sub-transaction's effect becomes permanent only when its TLT commits.

Assumptions We assume only LTs perform data manipulation operations and issue lock requests to obtain locks and PTs act as a place holders for the locks[1]. An LT is a flat transaction as defined in [3]; i.e., it is a representation of execution that identifies Read and Write operations and indicates the order in which these operations are executed. It is assumed that no transaction reads or writes data objects more than once. Also, a transaction reads before it writes any data object.

Knowledge of after-image : Normally, an LT copies data objects through read operations into private working space and issues a series of update operations. For the SNL approach, we assume that for any data object X, write operation is issued whenever it completes work with the data object. This assumption is also adopted in [1, 27].

[1]This restriction does not reduce the generality of the model[23].

3.2 Nested locking protocol

In this section we will summarize the NL protocol proposed by Moss[23]. Conventional locking protocols offer two modes of synchronization - **Read**, which permits multiple transactions to share an object at a time, and **Write**, which gives the right to a single transaction for exclusively accessing an object. Possible lock modes on an object are NL-, R-, and W-mode. The null mode (NL) represents the absence of a lock request for or a lock on the object. A transaction can acquire a lock on object X in some mode M; then it holds lock in mode M until its termination. Besides holding a lock a transaction can *retain* a lock. When a sub-transaction commits, its PT inherits its locks and then retains them. If a transaction holds a lock, it has the right to access the locked object (in the corresponding mode), which is not true for retained locks. A retained lock is only a place holder. A retained W-lock, indicates that transactions outside the hierarchy of the retainer can not acquire the lock, but that descendants of the retainer potentially can. That is, if a transaction T_i retains an W-lock, then all non descendants of T_i can not hold the lock in either W- or in R-mode. If T_i is a retainer of an R-lock, it is guaranteed that a non-descendant of T_i can not hold the lock in W-mode, but potentially can in R-mode. As soon as a transaction becomes a retainer of a lock, it remains a retainer for that lock until it terminates.

The NL rules for a transaction T_i are as follows.

- **NL1 :** T_i may acquire a lock in R-mode if
  - -- no other transaction holds the lock in W-mode, and
  - -- all transactions that retain the lock in W-mode are its ancestors.
- **NL2 :** T_i may acquire a lock in W-mode if
  - -- no other transaction holds the lock in W- or R-mode, and
  - -- all transactions that retain the lock in W- or R-mode are its ancestors.
- **NL3 :** When T_i commits, its parent inherits its (held or retained) locks. After that, T_i's parent retains the locks in the same mode (W or R) in which T_i held or retained the locks previously.
- **NL4 :** When T_i aborts, it releases all locks it holds or retains. If any of its superiors holds or retains any of these locks they continue to do so.

Note that the inheritance mechanism (Rule NL3) may cause a transaction to retain several locks on the same object. In such a case, a transaction retains a most restrictive lock.

4 Speculative nested locking

4.1 Lock modes and commit dependency

In the SNL approach, the duration of lock in W-mode is partitioned into three modes, EW- (**E**xecutive **W**rite)- , PSW(**P**assive **S**peculative **W**rite)- and ASW (**A**ctive **S**peculative **W**rite)-mode. The LTs request only R- or EW-mode lock. Also, note that an LT holds a lock, and a PT (or TLT) retains a lock.

An LT requests a lock in R-mode to read a data object and in EW-mode both to read and write a data object. Lock conversion from R- to EW-mode is not allowed.[2] An LT converts lock from EW-mode to PSW-mode whenever it produces after-image and holds the lock in the same mode until its termination. Whenever an LT holds lock in PSW-mode, its parent inherits and retains a lock in an ASW-mode. Let T_j be a PT and retained a lock in ASW-mode on a data object. As per SNL rules (explained in the section 4.2), T_j converts lock from ASW-mode to PSW-mode and retains in the same mode. Whenever T_j retains a lock in PSW-mode its parent inherits and retains lock in ASW-mode.

For X, a retained ASW-lock indicates that descendants of the retainer potentially can acquire lock in EW-mode, but all non-descendants of the retainer can acquire a lock only after it converts the lock from ASW-mode to PSW-mode. Similarly, a hold/ retained lock in PSW-mode indicates that any other transaction which obtains lock in R- or EW-mode should form commit dependency with lock holding transaction. If T_i forms a **commit dependency** with T_j, then T_i is committed only after termination of T_j. Let T_i be an LT, T_j be any sub-transaction (an LT, PT or TLT) such that T_j is non-ancestor of T_i. In the SNL approach, T_i forms commit dependency with T_j under following situations.

Commit dependency rules :

- If T_i obtains the lock in R-mode while T_j holds/retains a lock in PSW-mode on a data object, T_i forms a commit dependency with T_j.
- If T_i obtains the lock in EW-mode while T_j holds/retains a lock in R-mode or PSW-mode on a data object, T_i forms a commit dependency with T_j.

(Note that as per nested rule a parent (ancestor) commits only after termination of transactions in its hierarchy. Therefore, even though an LT obtains a lock in R- or EW-mode while its parent (ancestor) retains a lock in PSW-mode (as per SNL rules in section 4.2), we do not form commit dependency with ancestor transactions.)

[2]However one can observe that lock conversion can be easily incorporated.

4.2 Speculative nested locking protocol

We first explain the data structures used in SNL protocol.

- $tree_X$: We employ a tree data structure to organize the uncommitted versions of a data object produced by speculative executions. The notation $X_q (q \geq 1)$ is used to represent the q^{th} version of X. For a data object X, its tree is denoted by $tree_X$. It is a tree with committed version as the root and uncommitted versions as the rest of the nodes.
- $Depend\_set_i$: $Depend\_set_i$ is a set of transactions with which T_i has formed commit dependencies for all the data objects it has accessed.

We now present SNL synchronization rules. Each data object X is organized as a tree with X_1 as a root. We use the notation T_{im} to represent the m^{th} $(m \geq 1)$ speculative execution of T_i. Note that deadlock handling [23] algorithms needs to be initiated whenever a deadlock occurs.

- **SNL1 : Lock acquisition** Note that during lock acquisition whenever T_i forms a commit dependency (as per commit dependency rules) with T_j, the identity of T_j is included in $depend\_set_i$. (Rules 1.b and 2.b increase intra-transaction concurrency whereas rules 1.c and 2.c increase inter-transaction concurrency.)
  1. Transaction T_i may acquire a lock in R-mode if
     (a) no other transaction holds the lock in EW-mode, and
     (b) all transactions that retain the lock in ASW-mode are ancestors of T_i and
     (c) no other transaction retains the lock in ASW-mode and for each transaction that retains/holds a lock in PSW-mode, its TLT retains a lock in PSW-mode.
  2. Transaction T_i may acquire a lock in EW-mode if
     (a) no other transaction holds the lock in R- or EW-mode and
     (b) all transactions that retain the lock in R- or ASW-mode are ancestors of T_i and
     (c) no other transaction retains the lock in ASW-mode and for each transaction that retains/holds a lock in R-/PSW-mode, its TLT retains a lock in R-/PSW-mode.
- **SNL2 : Execution and inheritance**
  1. **Execution** Suppose T_i be an LT, and is carrying out m speculative executions and obtains a

lock in EW-mode on X. Let $tree_X$ contains n versions. Then, each T_{iq} (q=1 ... m) splits into n speculative executions (one for each version of $tree_X$).

Lock conversion by an LT Whenever an LT (T_i) produces after-images during its execution, after including each after-image of X as a child to the corresponding before-image of X's tree, it converts the lock from EW-mode to PSW-mode and holds in the same mode.

2. **Inheritance** The inheritance process can be separated into two types: LT to PT and PT to PT.

   (a) **LT to PT** Whenever an LT holds a lock in R-/PSW-mode, its parent (T_j)inherits and retains the lock in R-/ASW-mode. (To avoid inconsistency, the two actions, lock conversion from EW- to PSW-mode by LT and lock inheritance by its PT should be carried out atomically. To be safe, an LT converts EW- to PSW-mode only after its parent inherits in ASW-mode)
   Lock conversion by a PT When all sub-transactions of a PT (T_j) finish work on X, if only one LT in T_j's hierarchy holds the lock in PSW-mode, T_j converts the lock from ASW- to PSW-mode and retains in the same mode without waiting for the commit of other transactions in its hierarchy.
   Otherwise, if more than one T_j's LTs hold the locks on X in PSW-mode, then T_j converts the lock from ASW- to PSW-mode only after all LTs which have accessed X have been committed.[3]

   (b) **PT to PT** Whenever a sub-transaction retains a lock in R-/PSW-mode, its parent inherits and retains a lock in R-/ASW-mode.

- **SNL3 : Termination**

1. **Commit** A transaction T_i commits by selecting appropriate execution only after termination of all transactions in $depend\_set_i$. Each locked data object is updated with after-image produced by T_i as the root. The T_i's identity is removed from $depend\_set$ of all remaining transactions. Also, the waiting transactions drop speculative executions carried out by reading before-images of T_i.
2. **Abort** When T_i aborts, it releases all the locks it holds or retains. If any of its superiors holds or retains any of these locks they continue to do so. Also, each tree of a data object (accessed by T_i) is updated by removing after-images (with sub-trees) which were included by T_i. Its identity is removed from the $depend\_set$ of all waiting transactions. The waiting transactions drop speculative executions carried out by reading after-images of T_i.

[3]In this protocol we assume that a sub-transaction either commits or aborts. If aborts, it releases all the locks both hold/retained. Next, it is resubmitted. This process repeats until it commits. However, an abort of a nonessential LT is allowed in nested environments [23]. We are not considering such option here. However, one can observe that SNL could be extended under such environments.

4.3 SNLnp and SNLp approaccches

In SNL, after inheriting a lock from a sub-transaction (as per rule SNL2), a PT can not donate the locks in turn to its PT, unless all sub-transactions in its hierarchy finish the work with corresponding data object. Without having knowledge of data objects accessed by its sub-transactions, a lock is held by a PT until termination of all transactions in its hierarchy. Therefore, based on the prior knowledge of data objects accessed by a transaction, SNL can adaptively operate in two modes: SNLnp (SNL-no-predeclaration) and SNLp (SNL-predeclaration).

In SNLnp mode, a lock is held by a PT till termination of all transactions in its hierarchy. Therefore, SNLnp increases only intra-transaction parallelism (up to only one level in the nested hierarchy).

On the other hand, in SNLp-mode, once inherited the speculative locks from an LT, its PT (T_i) checks if any of its other sub-transactions requires access to corresponding data object. If none, then T_i's PT inherits the locks on the corresponding data object. In this way, speculative locks can be donated outside nested transaction before its termination (under rule 1.c. and 2.c. of SNL1). As a result, SNLp increases both intra- as well as inter-transaction concurrency.

4.4 SNL under Limited resource environments

In the SNL approach, the number of speculative executions of a transaction increases exponentially as data contention increases. Since each speculative execution needs separate work space, the size of main memory available in the system limits the number of speculative executions that can be carried out. With this limitation, processing cost may not be considered as a considerable overhead as current technology provides high speed parallel computers at low cost. Under limited resource environments the number of speculative executions of a transaction could be limited as follows. Let amount of memory to carry out single execution is one unit. Based on the available memory units, we

decide the feasible number of speculative executions that could be carried out by a transaction. During processing if the number of executions crosses the fixed value, the transaction is either put to wait or aborted.

5 Example

Consider follwing two transactions T_1(T_2 (T_4 : $\{V, X\}$ T_5 : $\{X, Y\}$), T_3 : $\{U, V\}$) and T_6(T_7 : $\{U\}$, T_8 : $\{Z\}$) which are simulataneously entered into the system (see Figure 2). Consider that all request locks in EW-mode. The processing employing NL and SNLp is as follows.

- **NL** Figure 3(a) depicts the processing employing NL. T_4 obtains lock on X only after termination of T_5. Similary T_3 obtains lock on V only after the abort of T_4 or the commit of both T_4 and T_2. Similarly, T_7 obtains lock on U only after the abort of T_3 or the commit of both T_3 and T_1.

- **SNL** Figure 3(b) depicts the processing with SNLp. At first, T_5, T_4, T_3, and T_8 obtain locks in EW-mode on X, V, U, and Z respectively. Whenever T_5 and T_4 produces after-images of X and V, respectively, T_2 inherits the lock in ASW-mode and whenever T_3 produces after-images of U, T_1 inherits the lock on ASW-mode. Next, T_4 obtains lock in EW-mode and carries out two speculative executions by accessing both before- and after-images of X. Due to pre-declared assumption (since T_5 will not access V), T_2 decides that it has finished work with V and therefore changes lock on V from ASW- to PSW-mode. Then, T_1 inherits lock in ASW-mode on V and retains in the same mode. Next, T_3 obtains lock on V in EW-mode and carries out two speculative executions. Due to pre-declaration assumption (no other sub-transaction of T_1 will access U), T_1 decides that it has finished work on U, and converts lock from ASW- to PSW-mode. T_7 onbtains lock in EW-mode (under rule 2.c of SNL1) carries out two executions by accessing before- and after-images of U.

In this way SNLp increases both intra- and inter-transaction parallelism of nested transactions.

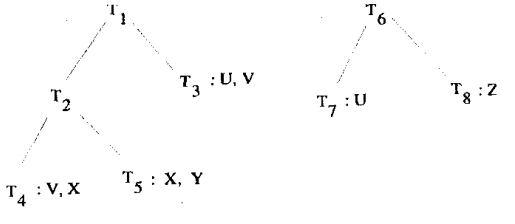

Figure 2. Depiction of T_1 and T_6.

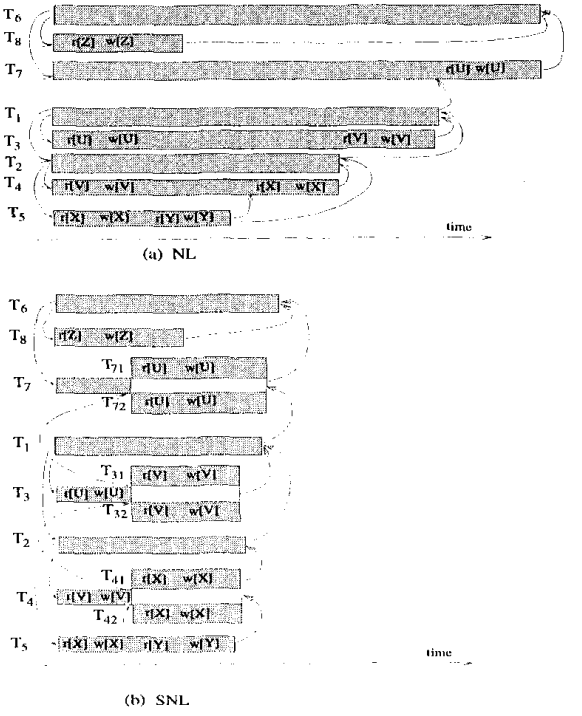

Figure 3. Depiction of processing (a) NL (b) SNLp (In this figure, an arrow from from a to b, indicates b happens after a.)

6 Correctness

In this section we informally argue that the histories produced by SNL are serializable like histories produced by NL[23].

In nested transactions ACID-properties are fulfilled for TLTs. Therefore, the execution of a group of TLTs is correct if it is equivalent to a serial execution of same TLTs. However, within TLT, each set of sibling transactions runs as if all the transactions that have committed ran in a serial order and all the transactions that aborted did not run at all. Since we assume all sub-transactions essentially commit (after re-submissions) the correctness criteria can be stated as follows. Let T_i be a TLT or a PT with 'k' siblings (children). T_i's execution is correct, iff it is equivalent to a serial execution of 'k' sibling transactions.

We briefly argue that the commit dependency rules and SNL rules preserve the correctness. Consider T_i and T_j are LTS that conflict (write-write) on X under same TLT. Suppose T_i first obtains a lock in EW-mode on X. When T_i converts its lock into PSW-mode, its parent inherits in ASW-mode. As per inheritance rules, the lock propagates to least common ancestor (LCA) of both T_i and T_j. The T_j obtains lock and carries out speculative executions by accessing before and after-images. Also, T_j forms commit

dependency with T_i and its ancestors which are in LCA's hierarchy. When T_j forms a commit dependency, it commits only after termination of T_i and its ancestors which are in LCA's hierarchy. If T_i commits, it selects appropriate speculative execution that ensures the order $T_i \prec T_j$. In this way a conflict among any two LTs within a TLT, forces a serial order among siblings of their's LCA. In this way, the SNL approach forces a serial order between transactions through corresponding LCA of conflicting transactions.

Similarly, TLTs execute in a serial order by forcing the commit dependency among TLTs.

7 Concurrency analysis

In this section we provide a simple analysis to demonstrate how NSL increases concurrency by considering a set of transactions under the same parent.

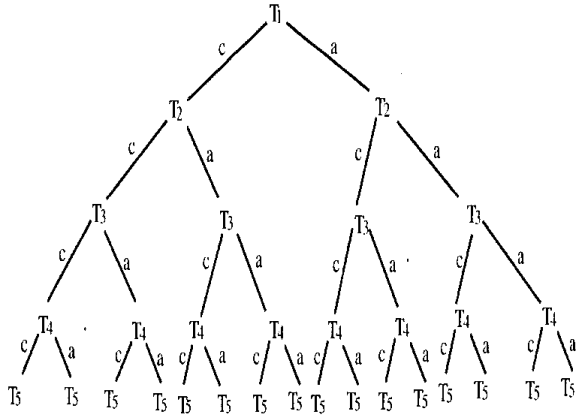

Figure 4. Depiction of processing under SNL when five transaction conflict on single data object (c denotes commit and a denotes abort).

Consider that n transactions under the same parent conflict on X. As per the proposed algorithm, a transaction reads k ($k \geq 1$) versions of X, carries out k executions and then includes n new versions to the tree of X (We assume that each execution produces a distinct version of X). Therefore for any transaction if it conflicts with n transactions, it has to carry out 2^n executions. Figure 4 depicts the processing when five subtransctions conflict on a single data object. To cover all possibilities of termination (commit and abort) of preceding transactions NSL carries out 2^n executions if it conflicts with n transactions.

In the SNL approach, since each speculative execution needs separate work space, the size of main memory available in the system limits the number of speculative executions carried out by a transaction. With this limitation, processing cost may not be considered as a significant overhead as current technology provides high speed parallel computers at low cost. We assume that when we have memory space, sufficient processing power is available to carry out speculative executions in parallel, so we do not specify amount of CPU power explicitly. We assume that all transactions require the same amount of main memory for execution. Also, for a transaction, all its speculative executions require the same amount of main memory. We consider that each execution requires one unit of main memory. We denote the amount of main memory available in the system by $memory\_units$ (mus).

Database systems vary with respect to available resources and data contention. Now we analyze how SNL increases the concurrency in such environments.

7.1 Single conflict environments (Hot spots)

In this section we analyze how SNL increases concurrency by considering an hot-spot situation. Consider that n sub-transactions conflict on X. If n transactions serially conflict on data object X, the total number of executions become equal to $\sum 2^k$, where, k=0 to n. Therefore to support these executions system requires main memory equal to $\sum 2^k$ mus.

Figure 5 shows the total number of executions (or mus) at different values of n, for 2PL and SNL. For example if three transactions conflict on data object X, 2PL requires 3 mus and SNL requires 7 mus ($= 2^2 + 2^1 + 1$). It also gives the ratio of mus required for SNL and 2PL. From this figure we can observe that, by doubling the memory size, each pair conflicting transctions can be processed in parallel. If we increase the memory three times, each group of three transactions can be processed in parallel. In this way SNL increases concurrency by trading extra memory and processing resources.

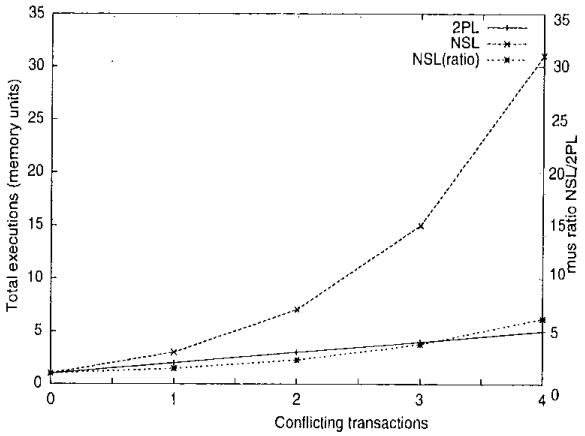

Figure 5. Conflicting transactions versus speculative executions (mus)

7.2 Multiple conflicts (long transactions)

In multiple conflict environments, the object dependency among the data objects accessed by a transaction affects the number of speculative executions. We first explain the object dependency through example.

Consider three transactions T_1, T_2 and T_3 arrive to access (X), (X,Y) and (Y) respectively, in that order. T_1 accesses X_1 and then produces X_2. When T_2 accesses both X it splits into two executions (with X_1 and X_2) and then each execution reads Y_1. If computation on the Y is dependent on the value (read or written by T_2) of X, the number of new versions added to Y's tree in this case is two. Next, when T_3 accesses Y, it has to carry out three executions since Y's tree contains three versions, because T_3's termination is indirectly depends on the on the termination mode of T_1.

We formally define the object dependency as follows.

Definition (depend(X,Y)) Given transaction T_i that accesses n objects ($n \geq 2$). Let X and Y be any data objects accessed by T_i. depend(X,Y)=true, if $w_i[Y]$ is an arbitrary function of $r_i[X]$. Otherwise, depend(X,Y)=false. We assume that for a data object X, $depend(X, X) = true$ always holds. That is, $w_i[X]$ is always an arbitrary function of $r_i[X]$.

If there is no object dependency exists among the data objects accessed by a transaction, 2^n number of executions are carried out if it conflicts with n transactions. So the analysis is same as single conflict environments.

However, if object dependency exists among the data objects accessed by a transaction, the SNL approach has to take into account the indirect dependencies. Let a transaction conflicts with n transactions directly and m transactions indirectly on a data object. Then, in the worst-case, the transaction has to carry out 2^{n+m} executions under SNL.

However, for any two data objects X and Y accessed by a transaction, depend(X,Y)=false is valid for majority of transaction processing applications. For transactions in real applications (such as banking and airline reservations), the computation on particular object is independent of value of other objects. If we observe the transaction profiles of TPC-A, TPC-B [12]and TPC-C [13] benchmarks, one can observe that there is no object dependency exists among data objects.

We assume that by compiling transaction (or SQL) code, we can identify the existence of object dependency among data objects accessed by a transaction. In [28] chopping algorithm is proposed which chops a transaction different pieces by checking the dependency among data objects. If higher degree object dependency exists among data objects (for example read the set of data objects and update the summary object) NL is preferred.

8 Performance issues

In this section we explain the performance issues concerning SNL.

- **No additional disk I/O**
  The SNL approach involves no additional disk write cost for recovery. In centralized environments, the system force-wrires after-images only after selecting appropriate final version. In distributed environment also, the coordinator force-writes corresponding after-images only after selecting the appropriate speculative execution. The participant force-writes corresponding after-images only after receiving final commit message from the coordinator. Therefore, even though a transaction produces a number of versions for a data object during its execution, force-writes only one version, after selecting the speculative execution to be confirmed. Therefore, no extra disk write cost is involved.

- **No additional logging overhead by employing delayed logging** In SNL the logging process is delayed until a transaction confirms the single version. So even though many versions are created all are kept in the main memory. However due to delayed logging, the already completed work may be wasted in case of a failure. We believe that the effect of such failures on the performance is negligible.

- **Message overheads: negligible**
  In distributed environment we separate the message overhead issue into two parts: execution and two-phase commit (2PC). During execution, after images of a data object are sent to object sites whenever a transaction produces them. This will increase the number of messages. If no transaction is waiting for corresponding data object at the object site, this message is wasted. This can be reduced in two ways. One, after-images of local data objects are included to respective trees whenever these are produced, however, remote data object values are sent with the ''prepare'' message of 2PC. Second, after-images are transferred only for remote hot spot data objects. Thus, with extra messages, concurrency can be increased by processing waiting lock requests.

  The number of messages during commit 2PC is not increased. After-images produced by the transaction are piggy-backed on commit processing messages. However, the size of the message increases but this will have negligible effect on the performance.

9 Summary and conclusions

In this paper we have proposed concurrency control approach based on speculation for nested transactions. In the SNL approach, a (sub)transaction releases a lock on the data object when it produces after-image.In this approach a transaction carries out multiple executions by reading both before- and after-images of preceding transaction. By trading extra resources SNL increases concurrency without violating serializability criteria.

As a part of future work, we evaluate the performance through simulation experiments and formally prove correctness.

Normally, a nested transaction consists of both essential and non-essential sub-transactions. If an essential sub-transaction aborts, the rest of the transaction has to be aborted. However an abort of non-essential sub-transaction is allowed in nested environment. However, it should be noted that a non-essential sub-transaction could block the essential sub-transaction by holding the crucial lock. We will investigate the performance by selectively processing essential sub-transactions with speculation.

We believe that our approach should work for distributed electronic commerce transaction environments. Even though electronic commerce attracts strong attention not so many systems are working yet. We are currently collaborating with industry to pick up some of real applications and verify the effectiveness of our approach using real database configuration data. We believe that the simulation study in this paper is useful to understand the behavior of the SNL approach. But simulation study using real business database applications data will give us further valuable insights.

Acknowledgments

This work is supported by "Research for the future" (in Japanese Mirai Kaitaku) under the program of Japan Society for the Promotion of Science, Japan.

References

[1] D.Agrawal, A.El Abbadi, and A.E.Lang, The performance of protocols based on locks with ordered sharing, *IEEE Transactions on Knowledge and Data Engineering*, vol.6, no.5, October 1994, pp. 805-818.

[2] Azer Bestavros and Spyridon Braoudakis, Value-cognizant speculative concurrency control, *proc. of the 21th VLDB Conference, 1995*, pp. 122-133.

[3] P.A.Bernstein, V.Hadzilacos and N.Goodman, *Concurrency control and recovery in database systems*(Addison-Wesley, 1987).

[4] A.Burger and P.Thanisch, Branching transactions: a transaction model for parallel database systems, *Lecture Notes in Computer Science 826.*

[5] P.K.Chrysanthis and K. Ramamritam. A formalism for extended transaction models. In proc. of 17th VLDB conference, 1991.

[6] P.Dasgupta, R.Liblanc Jr, and W.Appelbe. The clouds distributed operating system. In proceedings of 8th International Conference on Distributed Computing Systems, San Jose, CA, 1988.

[7] C.T. Davies, Data processing spheres of control, IBM Systems Journal. 17(2), pp. 179-198, 1978.

[8] A.K.Elmagarmid (ed.), Database transaction models for advanced applications, Morgan Kaufmann, 1992.

[9] K.R.Eswaran, J.N.Gray, R.A.Lorie, and I.L. Traiger, The notions of consistency and predicate locks in a database system, Communications of ACM, November 1976.

[10] J.N.Gray, Notes on database operating systems: in operating systems an advanced course, *Volume 60 of Lecture Notes in Computer Science*, 1978, pp. 393-481.

[11] J.Gray. et all. The recovery manager of the system R database manager, ACM Computing Surveys, 13, pp.223-244, 1981.

[12] Jim Gray (ed.) The benchmark handbook for database and transaction processing systems, Morgan Kaufmann, 1991.

[13] TPC BENCHMARK-C, Standard specification, Transaction processing council, October 1999.

[14] T.Harder and K.Rothermel, Concurrency control issues in nested transactions, The VLDB Journal, vol.2, no.1, pp.39-74, 1993.

[15] H.V.Jagadish, and O.Shmueli, A proclamation-based model for cooperation transactions, proceedings of the 18th VLDB Conference, Canada, 1992.

[16] W.H.Jessop, D.M.Jackobson, J.Baer, and C.Pu. An introduction to the Eden transactional file system. In proceedings of 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, PA, 1982.

[17] P.Krishna Reddy and Masaru Kitsuregawa, Improving performance in distributed database systems using speculative transaction processing,*in proceedings of The Second European Parallel and Distributed Systems conference (Euro-PDS'98)*, 1998, Vienna, Austria.

[18] P.Krishna Reddy and Masaru Kitsuregawa, Speculative lock management to increase concurrency in mobile environments, First International Conference, MDA'99, Hong Kong, December 1999, Lecture Notes in Computer Science, vol. 1748., pp. 81-95.

[19] P.Krishna Reddy and Masaru Kitsuregawa, Speculation to Increase the Concurrency of Nested Transactions, proceedings of International Conference on Information Technology (CIT'99) 1999, India.

[20] B.Liskov, Distributed computing in Argus, Communications of ACM, 31, pp.300-312, 1988.

[21] S.K.Madria, A study of the concurrency control and recovery algorithms in nested transaction environment, The Computer Journal, vol. 40, no.10, pp.630-639, 1997.

[22] S.K.Madira, S.N.Maheswari, B.Chandra and B. Bhargawa, Crash Recovery algorithm in open and safe nested transaction model, *Lecture Notes in Computer Science*, vol. 1308, 1997, pp. 440-451.

[23] J.E.B.Moss, Nested transactions: An approach to reliable distributed computing. Cambridge, mass, MIT Press, 1985.

[24] E.T. Mueller, J.D.Moore, and G.Popek. A nested transaction mechanism for Lotus. In proc. of 9th ACM Symp. on Operating Systems Principles, 1983.

[25] R.F.Resende, Synchronization in nested transactions, Ph.D thesis, University of California, Santa Barbara, 1994.

[26] D.P.Reed, Naming and synchronization in a decentralized computer system. Ph.D thesis. Technocal report MIT/LCS/TR-205, MIT Laboratory for Computer Science, MA.

[27] K.Salem, H.Garciamolina and J.Shands, *Altruistic locking*, ACM Transactions on Database Systems, vol. 19, no.1, March 1994, pp. 117-165.

[28] D. Shasha, F. Llirbat, E. Simon and P.Valduriez, Transaction Chopping : algorithms and performance studies, ACM Transactions on Database Systems, 20(3), Sept. 1995, pp. 325-363.