A Concurrency Control Protocol for Nested Transactions

Ming-Ling Lo and C. V. Ravishankar
Computer Sciences Division, Dept. of EECS
University of Michigan–Ann Arbor
Ann Arbor, MI 48109-2122

Abstract

Nested transactions[6, 5] provide fine grain atomicity, efficient recovery control, and structural modularity. In distributed environments, they provide a natural and semantically clean way of modeling computations. However, the characteristics of nested transactions are sufficiently different from those of traditional single-level transactions that concurrency control for nested transactions should be reconsidered in order to exploit all its advantages.

In this paper, we investigate a new concurrency control protocol for nested transactions, and introduce the notion of a *request list* for that purpose. Our objectives are to provide shorter transaction turn-around times and better system throughput. These goals are accomplished by exploiting intra-transaction concurrency and by reducing the time a transaction has to wait for consistent data states.

1 Introduction

Nested transactions are introduced to model long or complex transactions by offering fine-grain fault atomicity, more efficient recovery control, and structural modularity[6]. Nested transactions are most suitable when the amount of computation in a transaction is large, when the data accessing pattern is complex, or when one unit of computation directly or indirectly invokes computations at remote sites. Computations invoked at the remote sites are usually modeled as subtransactions. These characteristics make nested transactions a suitable candidate for modeling transactions in systems such as object oriented database systems.

Because of the aforementioned characteristics, nested transactions used to their full advantages tend to be larger. They have the following properties in comparison to single level transactions: for a nested transaction, (1) the expected execution time is longer, (2) the expect number of data items accessed is larger, and (3) during the its life time, the expected number of transactions trying to access the same resources it has accessed is larger. Although all nested transactions are not large, a well-designed nested transaction management system should cope with situations in which the number of large transactions is significant. Three problems could arise in such situations:

1. Transaction execution times may be longer because of larger and more complicated computations. Communications to pass subtransaction status between application and system software modules and communications between parent and children subtransactions at different sites can also increase the life time of nested transactions. This may result in degraded response time for interactive users.
2. The effect of resource waiting may seriously prolong transaction execution times. A transaction waiting for resources may wait a long time simply because the waited nested transactions have a lot of work to do. However, the waited nested transactions may very well be waiting for resources accessed by other transactions, since nested transactions access more data items than single-level ones. The chain of resource waiting can become very long if the concurrency control protocol is not properly designed. When the system workload is heavy, the degradation in transaction execution time and system throughput due to resource waiting can be multiplicative, and the chance for deadlock can be high. This makes plain locking concurrency control unsuitable for nested transaction.
3. Optimistic concurrency control can be dangerous. If a nested transaction is aborted, the number of dependent transactions that need be aborted will be larger, which in turn will cause a even larger number of transactions to abort. That is, the fan-out of the *cascading abort*[1] will be greater. Consequently, aborting a long-lived nested trans-

action may potentially paralyze the whole system.

To avoid these problems, a concurrency control protocol for nested transaction must:

- Minimize execution times for individual nested transactions.
- Minimize the time a transaction has to wait for resources held by other transactions.
- Minimize the number of cases in which a transaction has to proceed on the assumption that some active transactions will commit.

The above objectives are interrelated. By minimizing transaction execution times we could also minimize the penalty of waiting and reduce the danger of optimistic assumptions. On the other hand, reducing resource wait times naturally reduces execution times of transactions. One obvious approach to shortening execution times for individual nested transactions is to exploit intra-transaction concurrency, i.e., to allow subtransactions from the same parent to execute concurrently.

In this paper, we investigate a new concurrency control protocol for nested transactions that addresses these design goals. We minimize transaction execution times by exploiting intra-transaction concurrency. We also introduced the concept of a *request list*, which is derived from the *history* mechanism[7] and is merged with the timestamp serialization protocol[4]. request list reduces the cases in which a nested transaction has to wait for resources held by others, and avoid altogether the assumption that same active transactions will commit when allowing operations to proceed. The problem of cascading abort is avoid as a result. And by using timestamp as the serialization protocol, deadlocks cannot occur, either. The primary costs in this method is paid at the time operations arrive at data items. We believe the costs is justifiable given the high chance of deadlocks and expensive cost of cascading aborts for nested transactions.

Exploiting intra-transaction concurrency turn out not to be as straightforward as we expected, due to the fact that existing concurrency control techniques are based on the assumption that transactions are single-leveled. Section 2 provides backgrounds for our work and discusses issues concerning intra-transaction concurrency. Section 3 provides a brief description of the history mechanism. Section 4 and 5 gives and outline and the details of our algorithm, respectively. In section 6, we discuss how our methods avoids deadlock and cascading abort, among others. Section 7 conclude this paper.

2 Background

Work on concurrency control for single-level transactions has focussed mainly on inter-transaction concurrency, in particular inter-transaction concurrency between transactions from different programs. In such systems, operations from a single-level transaction are executed sequentially. Moreover, it is assumed that a program will issue and execute its transactions sequentially and synchronously. That is, one transaction is executed after its preceding transactions in the program have completed. If one transaction must be executed after the effect of another transaction has registered into the database for the whole computation to be meaningful, the programmer is responsible for enforcing such a requirement.

2.1 The Objectives of Serialization for Nested Transactions

The concept of *Serializability*[2, 8] was first proposed in single-level transaction systems to preserve data consistency. Serializability ensures that the effect of executing two transactions is equivalent to that of executing them sequentially. But data consistency is not all we ask for. For the execution of a collection of (sub)transactions to be meaningful, it is sometimes necessary that these (sub)transactions be serialized in some particular order. For example, consider transactions A, B, C, and D that perform financial computation for a company. A, B, and C calculate and record salaries paid to three employees in a certain month. D calculates the total amount of salaries paid that month. A meaningful way to serialize these transactions will be A-B-C-D. If the transactions are serialized the order A-B-D-C, data consistency is still preserved, but the result returned by D does not satisfy its semantic requirement.

Traditional single-level transaction systems assume the programmer will arrange the transaction code in a correct order, for example, A-B-C-D. Since transactions from the same program are executed synchronously, semantical correctness is preserved. Moreover, concurrently running programs are considered semantically unrelated. Enforcing specific serialization order is not important to concurrency control protocols of single-level transaction systems. It needs only ensure that transactions be serialized and data consistency preserved.

For nested transactions management systems, a set of subtransactions from the same parent transaction may have serialization ordering constraints among

them, yet still offer opportunities for exploiting concurrency. We need concurrency control protocols that support concurrency between these subtransactions while preserving the serialization constraints.

2.2 Serialization Ordering Requirement

To make the discussions of serialization and concurrency control in the following sections precise, we define the following terms:

For a set of computation units(e.g. transactions or basic blocks) of a computation process, the *logical ordering requirement* is the minimal set of ordering requirements between these computational units for the process to bear its intended meaning. Two units of computation may or may not have a logical ordering requirement between them. The requirement is minimal in the sense that if a unit X can either precede or follow another unit Y without affecting the program's intended results, no ordering is defined between X and Y. Using the example in section 2.1, the logical ordering requirement will be $\{(A, D), (B, D), (C, D)\}$. There is no ordering requirement between A, B, and C, since they can be executed in arbitrary order. Implementation details of a particular system may require B to be executed before C to guarantee correctness, but these issues are not relevant at this level; the requirement is purely at the logical level.

The *serialization ordering requirement* refers specifically to transactions. It is the order in which two (sub)transactions *must* be serialized for the computation to preserve its intended meaning. If A and B are two (sub)transactions, the serialization requirement are formally defined as the collection of the following three mutually exclusive relations:

1. NS: $(A, B) \in$ NS, if efforts need not be made to serialize the two subtransactions. For instance, the two subtransaction may never access common data items directly or indirectly.
2. AS: $(A, B) \in$ AS, if A and B must be serialized, but the order is immaterial.
3. SS: $(A, B) \in$ SS, if A and B must be serialized in some specific order to preserve the meaning of the computation.

Note that NS and AS can hold only between subtransactions without any logical ordering requirements between them, whereas SS must hold between subtransactions constrained by logical ordering requirements. We want the the relation SS to be minimal and compatible with the logical ordering requirement. We also need a way of specifying and communicating information about serialization ordering requirements to the serialization mechanism.

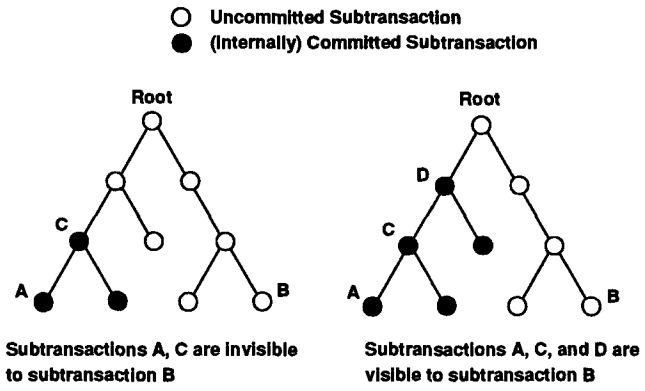

Figure 1: Visibility of subtransactions

2.3 The Visibility of Internal Commit

A subtransaction can be in one of two kinds of committed states. If a subtransaction is committed while its enclosing top-level transaction is not, the subtransaction is in the state of *internal commit*. If a top-level transaction commits, the top-level transaction and all its descendent subtransactions are in the state of *external commit* or *top-level commit*. The (sub)transactions in such a state is also said to be *committed to the top level*.

In a single-level transaction system, an operation is either visible or invisible to all other transactions. For nested transactions, because of their the hierarchical structure, the following *visibility principles for commitment* of operations should be observed[6]:

1. Top-level commit is visible to every operation. That is, an operation from a committed top-level transaction is considered committed by all other operations in the system.
2. Internal commit of an operation A is visible to operation B if and only if:
   (a) A and B are from the same top-level transaction, and
   (b) There is an ancestor of B which is a sibling of some internally committed ancestor of A.

The visibility of commit is illustrated in Figure 1. If the operation A is either committed to the top-level, or is internally committed and the internal commit is visible to operation B, we say A is *visibly committed with regard to* B.

Concurrency control protocols for nested transactions must enforce the visibility of internal commits. For locking methods, shadow copies can be used to enforce correct visibility[6]. However, this involves the costs of copying and inheriting the shadows. The copying cost would be quite expensive if multiple subtransactions from the same root transaction are to execute concurrently.

2.4 Related Concurrency Control Models

The easiest way to schedule subtransactions in a nested transaction system is to execute sequentially all subtransactions belonging to the same root transaction while interleaving subtransactions from different root transactions. This approach is not satisfactory because the active periods of individual transactions becomes long and the overall performance of the system might be poor as a result.

A slightly better method is to sequentially execute subtransactions from the same root, but with the following proviso: a group of subtransactions are executed concurrently, if they are adjacent to each other in program code, and it can be determined that relation NS or AS holds between each pair of them and that concurrent execution will not cause deadlock or contention. This is basically the way Argus [5] handles its intra-transaction concurrency control. However, this improvement is still unsatisfactory for the following reasons:

1. A subtransaction sees only its immediate children and no other descendants, hence it may not be possible to determine whether NS, AS, or SS holds between two of its children without violating modularity and analyzing the descendants' code. In distributed systems where a subtransaction can be an invocation on a remote object, such analysis may be impossible.

2. Analyzing the serialization ordering requirements between subtransactions are difficult and error-prone. In some systems this information may not be available until run-time. For example, it may depend on the parameters of program invocation or user input.

3. Even when the serialization ordering requirements are available, performance will still not be satisfactory if two subtransactions of relation SS are executed sequentially.

3 History Mechanism for Single-level Transactions

The abstraction of history[7] has been designed in the context of single-level transactions to support the implementation of atomic objects[5, 11] and meet the demand of systems with long transactions or with localized concurrency bottlenecks (hot-spots). It provides the basis for our concurrency control mechanism for nested transactions.

The history mechanism uses application semantics to increase concurrency. Introducing application semantics into transactions has drawbacks. Not only writing applications becomes more complicated, it also becomes difficult to write an application program without unnecessarily exposing details of the underlying concurrency control algorithm. The history mechanism alleviates these problems by classifying the operations on data types into mutators and observers, and hiding the serialization protocol from the applications. Classifying operations into mutators and observers provides a way of systematically and efficiently exploiting application semantics without exposing too many details. This method is particularly interesting in the context of nested transactions because it greatly reduces the number of cases in which an operation has to wait for others.

3.1 Data Objects with History Abstraction

Conceptually, an atomic data object can be viewed as a state machine with four components: a set of possible states, an initial state, a set of possible transitions, and a set of rules that determine how the states of the atomic object are changed by the transitions. A *transition* is an ordered pair consisting of an operation invocation on the data object and its result. We will use the terms operation and transition interchangeably when there is no ambiguity.

The state of an atomic object is represented as a list, or history of previously executed transitions. The execution of an operation is implemented as an addition to this history of transitions. Formally, the knowledge possessed by an atomic object with the history abstraction can be expressed as a triple (T, C, O), where:

- T is the set of operations from all transactions that have arrived at the data object.
- C is the set of operations from committed transactions. $C \subseteq T$.

- O is a relation between elements of T. $(t_1, t_2) \in O$, if the object knows t_1 is serialized before t_2.

How the serialization ordering becomes known to the object depends on the serialization protocol. If timestamp ordering is used, the data object discovers the ordering of two operations by comparing their timestamps. If the serialization order is determined by the order of commitment, as is usually the case when locking is used for concurrency control, the data object obtain more information on ordering when it is informed of the completion of some subtransactions.

The notion of *possible serialized sequences* [7] is a central idea used in the history mechanism to synchronize operations performed on the same data object. A possible serialized sequence of an object can be viewed as a *possible development* of the history of that object. Each possible serialized sequence represents a possible state of the object in the future when all operations in the sequence are committed in the described order, and all other transitions in the history are aborted.

Formally, given a history (T, C, O), a sequence S of operations is a possible serialized sequence if

1. S includes all committed operations, i.e., $S \supseteq C$.
2. S preserves known ordering of operations, i.e., for $t_1, t_2 \in S$, if $(t_1, t_2) \in O$, then t_1 is ordered before t_2 in S.

There may be several possible serialized sequences for with a given history. For instance, consider the following history, produced possibly under commit-order serialization protocol:

$T = \{t_1, t_2, t_3\}$,
$C = \{t_1\}$,
and $O = \{(t_1, t_2), (t_1, t_3)\}$.

All possible serialized sequences for this history are:

$(t_1, t_2, t_3), (t_1, t_3, t_2), (t_1, t_2), (t_1, t_3), (t_1)$

If timestamp ordering is used, the ordering between any two transitions are known, and O might become $\{(t_1, t_2), (t_1, t_3), (t_2, t_3)\}$. The possible serialized sequences in this case are:

$(t_1, t_2, t_3), (t_1, t_2), (t_1, t_3), (t_1)$

3.2 Mutators and Observers

The history mechanism classifies operations on a data object into *mutators* and *observers*. A mutator is an operation that changes the object state. A mutator differs from a write in that its may change the objects state relative the object's previous state, whereas a write determines new object state based solely on its parameters. An observer is an operation that deduces information from the object state. Conceptually, an observer derives its result by observing the possible serialized sequences, if there are more than one such sequences, information deduced may be inconsistent. The observer fails in this case. An operation is both an observer and a mutator when it both deduces information from and changes the object state.

With the above definitions, the algorithm to synchronize operations arriving at a data object can be summarized as follows:

1. Before adding a new observer into the history, determine whether it observes different results in different possible serialized sequences. If so, either the new observer are delayed or aborted, or some action needs be taken to make the observations, such as aborting some active mutators.
2. Before adding a new mutator, we should determine whether some observers serialized after the new mutator would observe inconsistent result if it is inserted into the history. The mutator is inserted only when no such observers exist. Otherwise, the it is aborted.
3. When a transaction commits, all operations it issued are marked as committed, i.e. included into the set C. If a transaction aborts, the records of the operations it issued are erased from the history.

3.3 Advantages of the History Mechanism

The history mechanism potentially allows more concurrency in a single-level transaction system than its counterparts using read/write locking or timestamp concurrency control[7].

For instance, suppose transaction A accesses data object X, and then B accesses X. With read/write locking, unless both A and B are reads, B would be blocked until A completes. In the history mechanism, B would be only if A is a mutator, B is an observe, and B observe inconsistent result.

With single version timestamp concurrency control, if A read X and then B write X with a smaller timestamp, we would either have to abort A or B. In the corresponding situation for a history mechanism, if A issues an observer to X and then B issues a mutator with smaller timestamp, we need not abort either A

or B unless an observer would be invalidated if B is inserted into the history.

The history mechanism is comparable to multi-version timestamp protocols[10]. However, it does not have the problem of cascaded abort. When a mutator is about to be insert into the history, we make sure no existing observers would observe differently because of the new inserting. Similarly, an observer is allowed into the history only when no abortion of existing mutators can affect the information it derived. When a mutator needs be aborted, it can simply be delete from the record, affecting no other operations.

4 The Algorithm for Nested Transaction Currency Control

In this section, we described a concurrency control protocol for nested transaction that is derived from the history mechanism and is merged with the timestamp serialization protocol. The concept of a *request list* is introduced to support the protocol.

4.1 The Data Object Model

In our model, a data object knows (1) all operations invoked upon the object, (2) the ordering among these operations, (2) the originating transactions of these operations, and (3) the commit status of the originating transactions. This knowledge is recorded in the request list. When an operation try to deduce information from a data object, an *object state* is derived for it from the request list depending on its serialization ordering with respect to other operations.

Timestamps are used to order all top-level transactions. A precedence number, unique with the enclosing top-level transaction, is associated with each subtransaction and used to order the subtransactions. The precedence number must be compatible with the logical ordering requirement, i.e. if subtransaction A is logically required to be ordered before subtransaction B, A will be assigned a smaller precedence number. Precedence numbers can be assigned either by examining the location of subtransactions in the transaction code or by specification by the programmer.

Since subtransactions may be invoked dynamically and in a hierarchical fashion, we do not assume precedence numbers to be either consecutive or integer. Figure 2 shows one way of assigning precedence numbers, with precedence determined by lexicographical order. For example, precedence number `1.1` is smaller than `2`, and `2.1.2` is greater than `1.2.2`.

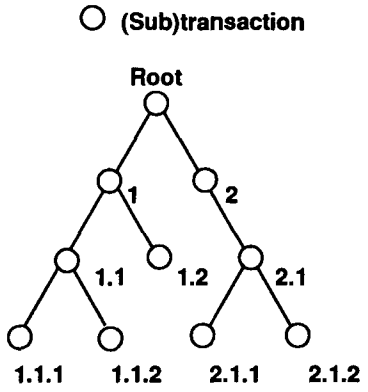

Figure 2: Example of precedence number assignment for subtransactions

Formally, the request list of an data object D is defined as a tuple (T, C) where:

- T is a sequence constituted of all operations invoked on the data object. The ordering of the operations within the sequence reflects their serialization order. Operation x appears before operation y in T if and only if x should be serialized before y.
- C is the set of all committed operations in T.

As in the history mechanism[7] operations are classified into observers and mutators. An operation both changes and derives information from the object's state will assume the roles of both a mutator and an observer.

An *epoch* is defined as the interval between two adjacent mutators, or the interval from the start to the first mutator in the request list. An epoch whose beginning and end mutators are M_i and M_j is denoted (M_i, M_j). An observer O_l in the request list is considered to be in the epoch (M_i, M_j), if the ordering of the operations is $M_i < O_l < M_j$. An operation before an epoch (M_i, M_j) is either an operation serialized before M_i or M_i itself. An operation after (M_i, M_j) is either an operation serialized after M_j or M_j itself.

There is a set of *candidate states* associated with each epoch, which represents the set of possible states the data object could assume if and only if all operations before that epoch have completed, i.e., either committed or aborted. The candidate states associated with epoch (M_i, M_j) are the set of object states an observer located in the epoch could possibly see. Figure 3 gives examples of candidate states. When a mutator is committed or aborted, all epoch serialized after this mutator must adjust their candidate states

accordingly. If all mutators before an epoch have committed, the set of candidate states associated with this epoch is a singleton.

A main difference between the history mechanism[7] and request list is that the ordering between two operations in a history may be unknown, while that of two operation in a request list is always known. History avoid this requirement because it tries to separate the serializing protocol from its concurrency control protocol and support both timestamp ordering and commit order ordering.

4.2 The Basic Algorithm

According to criteria described below, an operation P_l that arrives at a data object can be rejected or inserted into the request list. If it is inserted, it will be inserted into an epoch (M_i, M_j) where $M_i < P_l < M_j$. If the new operation is a mutator, the epoch is split into two epochs: (M_i, P_l) and (P_l, M_j). When we need abort an operation from an internally committed subtransaction, we have to abort the youngest uncommitted ancestor. So simplicity, we sometime also refer to such a scenario as aborting the operation.

The concurrency control algorithm using the request list is as follows:

1. *Operation ordering.*
   (a) Each top-level transaction is associated with a timestamp. Each subtransaction is assigned a precedence number, unique within its enclosing top-level transaction.
   (b) The data object determines the precedence of operations according to the following criteria:
   - Two operations with different timestamps, i.e. two operation comes from different root transactions, are ordered by comparing their timestamps.
   - Two operations with the same timestamp are ordered by comparing the precedence numbers of the subtransactions they come from.
2. *Processing of observers.* When a new observer arrives at a data object, the observer derives its observation from the set of candidate states associated with its enclosing epoch as follows:
   - If the observer derives the same result from all candidate states in the period, the observer is inserted into the request list.
   - If the observer derives different results from different candidate states, the observer is blocked and retried later when it can observe a consistent result from all candidate states. The issue of retry will be discussed in the next section.
3. *Processing of mutators.* When a new mutator arrives at a data object, we must decide whether or not to insert the mutator into the request list. If the mutator is inserted, the candidate states for the all epochs located after the new mutator might be changed. An observer which previously saw consistent results in one of these epoch may now see inconsistent results. If this happens, the observer is said to be *invalidated* by the mutator. The issue of checking invalidation is discussed further in the next section. The criterion for deciding whether to insert the mutator is as follows:
   (a) If no observer in the request list is invalidated, the mutator is inserted into the request list.
   (b) If some observers are invalidated, then:
   - If some of the invalidated observers are from different top-level transactions, then:
     - If any of the invalidated observers has committed to the top level, the mutator is aborted.
     - Otherwise,
       we call an algorithm `ChooseAbort` to determine whether to abort the offending mutator or the invalidated observers. `ChooseAbort` can be designed according to the number and commit status of invalidated observers (uncommitted or internally committed), the relative cost of abort mutators and observers, etc. It can be tuned to optimize system throughput, or simple heuristic can be used. When an internally committed observer needs to be aborted, we have to abort the youngest uncommitted ancestor of the observer.
   - If the mutator and all invalidated observers come from the same top-level transaction as the mutator, i.e., they have a common ancestor, then the observers must be aborted regardless of whether any of them has internally committed, since these observers must have

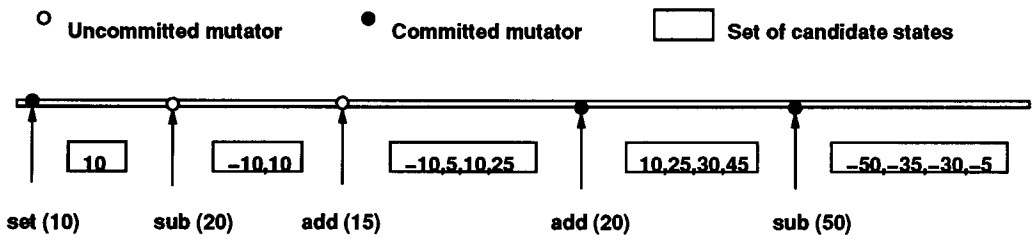

Figure 3: Examples of candidate states

been ordered after the mutator within the top-level transaction and some of them might logically depend on the result of the mutator. None of the observers in this case could have committed to the top-level, since there is at least one operation, the mutator still uncommitted in this top-level transaction.

4. *Commit order constraint.* Subtransactions under the same parent must commit in the order of their precedence number. That is, a subtransaction can commit only when all subtransactions under the same root with smaller precedence numbers are committed. A top-level transaction can commit when all its children have internally committed.
5. *Processing commits and aborts.* When a (sub)transaction aborts, the operations issued from this (sub)transaction are simply deleted from the request list. If a deleted operation is a mutator, conceptually the immediate epochs before and after the mutator will be merged into one. The sets of candidate states associated with the epochs after the aborted mutator may change as a result. When a (sub)transaction commits, the candidate states of the epochs after its mutators may also need be updated. In both cases, the cardinalities of the changed candidate state sets are likely to become smaller.

The model above is interesting in two ways. First, the serialization protocols for top-level transactions and for subtransactions are decoupled. Timestamps are different from precedence numbers in that when a top-level transaction is aborted and retried it has a new timestamp, whereas if only an operation within a transaction is retried, its precedence number is still the same. This subtle difference allow us to exploit intra-transaction concurrency with minimal overhead. Potentially other serialization protocol can be used for top-level transaction or subtransactions to explore other possibilities. Second, the issuing and commitment of subtransactions are decoupled. In the above discussion, we made no assumptions about the way we issue the subtransactions. In principle, all of the subtransactions under the same parent can be issued and executed concurrently, as long as we force their commit order to conform to the serialization ordering requirement.

4.3 Commit and Abort Issues

When a subtransaction is aborted, its parent could either proceed without the child, re-issue and retry the child, or abort itself. In our method, the concurrency control algorithm will initiate an abort only when a mutator newly arrived at a data object invalidates some existing observers in the request list. In this case, either the offending mutator or the invalidated observers may be aborted.

If the offending mutator M_i is aborted, there is no point for its parent P_j to retry it again, because M_i will come with the same time-stamp and the same precedence number, be inserted into the same place, and invalidate the same and newer observers. If P_j is programmed in such a way that it keeps retrying M_i, eventually P_j's parent will see it as time-out and abort it. Therefore, when an offending mutator is aborted, we abort it with the hint **NO_REINSTANTIATE**. The parent P_j could use this hint to make an appropriate decision. If this mutator is essential for the whole nested transaction, the nested transaction will have to abort itself eventually and a new nested transaction may be started with a new timestamp.

An observer to be aborted could be either totally uncommitted or internally committed but not externally committed. Suppose an uncommitted observer O_i is aborted. Since new nested transactions will be assigned new timestamps, and old transactions will keep completing, the number of transactions with timestamps smaller than O_i will decrease with time. Eventually there will be no mutator that could possibly invalidate O_i. Therefore, an uncommitted observer is aborted with the hint **REINSTANTIATE** to in-

dicate that if O_i is retried, it will have a better chance to succeed. On the other hand, it is dis-advantageous for O_i's parent to abort itself in this case. Aborting O_i's parent would waste system resources and introduce the possibility of *infinite retires* if the abortion leads to the abortion of the whole nested transaction. When a new nested transaction is instantiated with a larger timestamp, it faces an increased risk of being invalidated by mutators in the systems.

When an internally committed observer O_i must be aborted, we cannot simply abort the observer itself, since the O_i's parent have already see it as committed. Instead, we must abort the youngest ancestor A_j of O_i which has not committed internally. A_j has not promised its parent anything yet.

Re-instantiating A_j in this case can be either useless or helpful. If many of A_j's descendants are mutators, retrying A_j may be useless, because these mutators may be invoked with different parameters depending on the new result of O_i, and thus invalidate other observers in the system. On the other hand, if nearly all descendants of A_j are observers, then the retry is very likely to succeed. Heuristics can be devised to return the right hint to the program to improve system performance.

When a newly arrived mutator invalidates existing observers, and none of the invalidated observers have committed to the top level, we use `ChooseAbort` to decide whether to abort the observers or the mutator, as described before. The idea is to choose whichever side will result in less overhead. If we abort the mutator, it is very likely the whole top-level transaction will be aborted and everything done so far would be wasted. If the observers are aborted, we also need to abort all subtransactions that depend directly or indirectly on the result of these observers. There are ways to associate the costs or predicted costs of algorithm-initiated abort with each mutator and observer, and `ChooseAbort` may decide which side to abort by comparing these costs of the observer and the mutator. On the other hand, we can of course always choose to abort the mutator or the observer, and avoid overhead in `ChooseAbort`.

5 Details of the Algorithm

The primary costs characteristic to this algorithm can be divided into three main categories: (1) the cost of deciding what a newly arrived observer will see, (2) the cost of deciding whether a newly arrived mutator will invalidate some existing observers, and (3) the cost of retries. In this section, we discuss the details in our algorithm that minimize these costs.

5.1 Classification of Operators

In order to use operator characteristics to improve performance, we further classify operations as follows:

- *Relative mutators and absolute mutators*: A relative mutator changes the data object to a new state relative to the original object state. Adding an amount to an account object and withdrawing an amount from an account object are examples of relative mutators. Absolute mutators change the states of data objects without referencing the original states. That is, the new states depends on the parameters of these mutators only. Examples are: resetting an integer to zero or setting an integer to 100.

- *Value observer and criterion observer*: A value observer reports the value or part of the value of the data object. Criterion observers ask yes-or-no questions and return boolean values as answers. Examples are: asking whether an account is greater than zero or less than a certain amount.

5.2 Handling Observers

To simplify the discussion below, we assume all data objects involved are scalar objects. We have extended these methods to composite objects.

Value Observers

Suppose O_v is a newly arrived value observer and O_v will be inserted into epoch (M_2, M_3) if accepted. O_v is handled according to the following principle:

> Let M_1 be the last visibly committed absolute mutator with respect to O_v. If all mutators between M_1 and M_2 including M_2 are *visibly committed* with respect to the O_v, then the observer can observe a consistent result and is inserted into the request list. Otherwise O_v is blocked and retried.

In an implementation, the above condition can be checked by a backward linear scan of the request list from M_2. The scan stops at the first uncommitted mutator, in which case O_v fail. If no uncommitted mutator has been encounter and a visibly committed absolute mutator is met, O_v is successful.

Criterion Observers

The handling of criterion observers can be optimized significantly. For each kind of criterion observers we define a *worst possible value* and a *best possible value* (BP and WP values hereafter). Each epoch has a BP and a WP value, each of which is en element of the set of candidate states. The question of whether a newly arrived criterion observer can succeed and the information the observer need can be answered by consulting the BP and WP values of the epoch enclosing the observer.

For example, Figure 4 shows a data object exporting the observer `IsGreaterThanZero`. For each epoch, we keep WP value **pmin**, which is the smallest possible value for all candidate states at that epoch, and a BP value **pmax**, which is the greatest possible value. When an instance of `IsGreaterThanZero` arrives at the data object, we know it must return `YES` if both **pmin** and **pmax** are greater than zero. No matter which active mutators are finally committed or aborted, the candidate states in the epoch would always give the answer `YES`. If both **pmin** and **pmax** are less than zero, the observer must return `NO`. If **pmin** is less than zero while **pmax** is greater than zero, we infer that the observer observes inconsistent results, and should be blocked and retried.

To summarize, we define a BP value and a WP value for each criterion observer type. When an instance of that observer type arrives at the data object, if both BP and WP values satisfy the criterion, then the observer succeeds with answer `YES`. If both BP and WP values don't satisfy the criterion, the observer succeeds with the answer `NO`. When BP satisfies the criterion and WP value doesn't, the observer could observe inconsistent results, and therefore should be blocked and retried.

Denote the BP value at a epoch h as BP_h, the nth mutator in the request list as M_n, and the result of executing M_n on a possible value BP_{n-1} as $M_n(BP_{n-1})$. Then BP_n is obtained in the following way:

1. BP_0 is the initial state of the data object.
2. If M_n is externally committed, then $BP_n = O_n(BP_{n-1})$.
3. If M_n is not externally committed yet, then compare the value BP_{n-1} and $O_n(BP_{n-1})$, whichever is *better* is chosen as BP_n.

The WP values can be obtained in a similar fashion, except that we choose whatever is worse when the operation in consideration is an uncommitted mutator. Figure 4 gives examples of calculating BP and WP values.

When a new mutator arrives at a data object, it may be serialized at the end or in the middle of the request list. In either case, the structure of the request list will change. If the mutator is appended to the end of the request list, only the BP and WP values in the new epoch need be calculated. If it is inserted in the middle of the request list, all of the BP and WP values associated with the epochs positioned after the new mutator will change. Similarly, when an operation commits or aborts, the BP and WP values associated with the epochs after the operation also change.

The BP and WP values can be shared among different types of criterion observers. For example, if there is another criterion observer `IsGreaterThanFifty`, the BP and WP values will be the same as those of `IsGreaterThanZero`. If there is a criterion observer `IslessThanZero`, then **pmin** becomes the BP value and **pmax** becomes the WP value.

Value observers can also be handled using BP and WP. A value observer derives a consistent result from an epoch if and only if the set of candidate states of the epoch is a singleton set, in which case the BP and WP of the epoch would be the same, and their value is the answer to the values observer. In actual implementation, if we can define BP and WP values for all *criterion* observers, we need not maintain the set of candidate states at all.

The key issue in this discussion is: what is the definition of *better* or *worse*? If the criterion observer is based on numerical number comparison such as the one in the above example, the definition of better or worse can easily be given without ambiguity. However, we are unable to exhaust all categories of criterion observers that may be used, and are unable to give a formal definition of being *better* or *worse*. Our contention is that in most practical cases, there are an unambiguous ways of obtaining the BP and WP values. If a clear definition of BP and WP values is impossible, we can simply reclassify the observer as a value observer.

Compensation for the Visibility of Internal Commit

One might notice that we did not take into account the visibility of internal commit in the above calculation of the BP and WP. That is, we treat as uncommitted all mutators has not committed to the top level. The result is that we are overly pessimistic.

Consider the example in Figure 5. The observer in question is `IsGreaterThanZero`. Let `add(15)`, de-

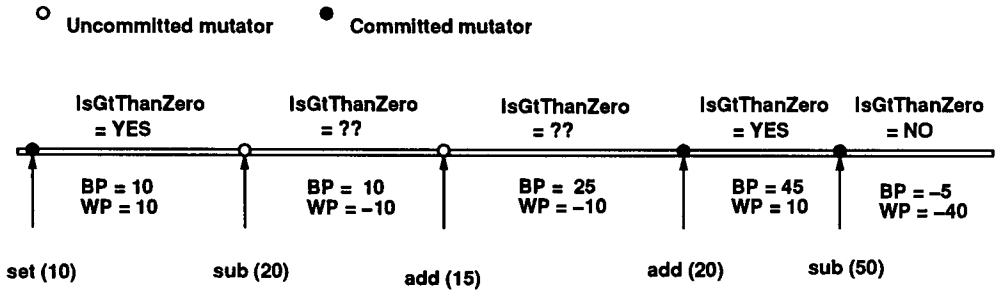

Figure 4: Maintaining best and worst possible values for `IsGreaterThanZero`

noted as M_3, be the mutator that divide epoch 2 and 3. The BP and WP values in the epoch 2, i.e., BP_2 and WP_2, are 10 and -10, respectively. According to the rules described above, BP_3 and WP_3 are 25 and -10. Therefore, a newly arrived observer O_c serialized into epoch 3 would not be able to derive any conclusion and would be blocked. However, if the new observers is from the same top-level transaction as M_3 and M_3 is visibly committed with respect to O_c, then the worse possible value O_c can see is 5 and O_c should in fact return the answer `YES`.

We need to compensate the effects of internal commit:

> When a new observer arrives, if it cannot succeed with current BP and WP values, we check to see if there are mutators whose internal commits are visible to the observer. If such mutators exist, we calculate a temporary set of BP and WP values with the mutators regarded as committed. If both WP and BP values in the temporary set satisfy or fail the criterion, the observer succeeds. Otherwise, the observer is blocked.

We choose not to consider internal commits when deriving BP and WP values but compensate for them later, because considering internal commits when deriving BP and WP values will result in one set of BP and WP values for operations from each different top-level transaction. The cases in which compensation for internal commit is necessary are expected to be few. We need to compensate for internal commit only when an observer and a mutator from the same top-level transaction access the same data object, and the observer is serialize before the mutator in the top-level transaction. In other word, we use the simpler concept of one set of BP and WP values and compensate for internal commit when necessary so that normal processing will not be burdened by infrequent special cases.

Such compensation is not necessarily expensive. If the mutator is a commutative operation, the temporary set of BP and WP values can be obtained by applying the mutator on one of the old BP or WP values.

5.3 Handling Mutators

When a mutator M arrives at a data object, we need decide whether M will invalidate any existing observer. The following test answers this question for value observers:

> If there is a value observer O_v serialized after M and there is no externally committed absolute mutator serialized between them, then O_v is invalidated by M. Otherwise no value observers are invalidated.

To decide if a new mutator M invalidates any existing criterion observer, we need to calculate new WP and BP values for each epoch that is located after M and contains criterion observers. Then we conduct the following test on criterion observers located within epochs whose BP and WP values have changed:

> Is there is a criterion observer which returned `YES` but now fails to satisfy the criterion?
> Is there is a criterion observer which returned `NO` but now satisfies the criterion?

If the new mutator invalidate some existing criterion observers, it is processed as described in section 4. Otherwise, no observers are invalidated and the candidate states in each epoch located after the new mutators should be updated.

Note that we were not concerned about the visibility problem of internal commits when handling the new mutators. A mutator cannot arrive already internally committed.

We also did not consider the possibility that a previously blocked observer becomes able to observe a consistent result because of the newly arrived mutator.

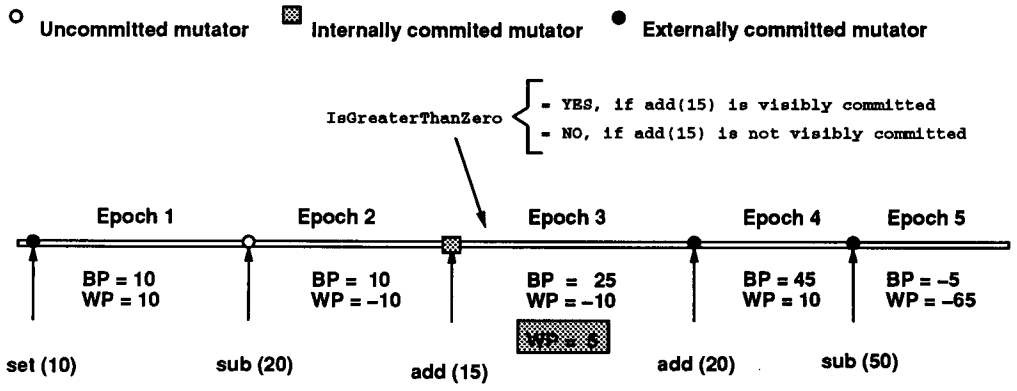

Figure 5: Compensation for visibility of internal commit

Inserting an uncommitted mutator can only make the request list more *uncertain* and reduce the chance that a consistent state can be observed.

5.4 The Handling of Retries

Another cost in our protocols is that cost of retries. This cost also exists in the history mechanism for single-level transactions. In the original model, when a new observer derives inconsistent results from different possible serialized sequences, the observer is rejected and will be retried later by the issuing transaction. The delay time before the next try is decided by the transaction. The transaction may also choose to abort the observer after some number of retries.

Retry is a serious problem for the original history mechanism, as stated in [7]. First, since retries are initiated by the issuing transactions, there are additional communication costs between the transaction software and the data object manager, both when an observer is *rejected* by the data object and when it is retried by the transaction later. The cost is even higher if they reside in different sites. Second, the transaction retries by guessing a suitable delay. It is possible that the operation still cannot proceed on the next try, or that it could have been successfully tried earlier.

For these reasons, we choose to block the operations at the data object sites and have the objects initiate the retries. A data object knows more about what happened inside itself and what is the best time to retry. Communication costs will be saved, and transaction code will be easier to write. The transaction may choose to abort the observer if it does not return after some period of time.

Retrying Value Observers

Ideally, a blocked observer O_b should be retried when the request list of the data object has changed in such a way that O_b can now observe a consistent result. However, such condition is expensive to detect.

Our solution is to avoid testing for this condition, and have the data object choose a time when it is highly possible that the retry will succeed.

The following is a sufficient condition for the success of observers:

> Suppose the blocked observer O_b is serialized in epoch (M_2, M_3). Let M_1 be the last logged absolute mutator. O_b will be able to observe a consistent result if that M_1 becomes visibly committed and all mutators between M_1 and M_2 in the request list including M_2 become either visibly committed or aborted.

If M_1 is aborted instead of being visibly committed, by definition, the last logged absolute mutator will become the M_1 in the test.

The condition described above depends on a sequence of consecutive mutators rather than a set of possible serialized sequences as in single-level history mechanism, and is therefore easier to evaluate.

In real implementations, we can implement heuristics for the test and pay even less price. An good heuristic would be to retry O_b when M_2 becomes visibly committed or aborted. The reason is that because M_2 is the mutator with the largest timestamp before O, it is highly probable that when O is aborted or visibly committed all mutators between M_1 and M_2 would have done so, too. Other good choices include retrying O when both M_1 and M_2 are completed, or retrying O when the last two mutators before O are completed.

Retrying Criterion Observers

Since it is easier for criterion observers to make consistent observations than for value observers, the above condition may be too conservative for criterion observers. As noted earlier, when a mutator commits, we have to update the possible values for all epochs located after it. When updating possible values for these epochs, we can also decide whether a blocked criterion observer can now succeed.

> When an observer is blocked, it is blocked on a BP/WP type and an epoch. The observer is retried when the BP/WP values in the epoch has changed.

The test above is optimistic. In real implementation, heuristic attempting fewer tries can be used. For example, we can use the test for value observer together with a background process retrying observers that have blocked for too long.

5.5 Remarks

The discussion in this section has relied on the classification of operations, such as classifying observers into value and criterion ones. One might suggest that the operations on data objects do not always allow such classifications, or that it is not always possible to define the WP value and BP value for a criterion observer. However, the classification and the WP and BP values only serve to optimize performance. If we found that a clear classification or or a definition WP or BP values is not possible for an observer, we can simply classify the observer as a value observer, and the algorithm can proceed without any difficulty. Similarly, if there is any ambiguity, we can classify the mutator as a relative mutator.

The concepts discussed in these sections, such as sets of candidate states, need not correspond to physical entities. We have provided methods to simplify the protocol at the conceptual level, but various implementation issues still remain to be investigated.

6 Performance Failures

We use timestamps to order top-level transactions in our model. As timestamp ordering is free from deadlock[3, 9], deadlocks cannot occur across two top-level transactions. Subtransactions are assigned precedence numbers which are unique within a top-level transaction. For similar reason, deadlock cannot occur between subtransactions within the same top-level transaction.

Cascading abort, or the domino effect cannot occur, either. Suppose operations A and B are both active and access the same data object; B is serialized after A. Aborting A can affect B only when A is a mutator and B is an observer. Since the set of candidate states in B's epoch has included all possible object states in which A can be aborted or committed, if B was allowed into the request list in the first place, it means B can derive a consistent result whether A commits or not. Aborting A would not change B's observation. Therefore, not only cascading abort cannot occur, aborting a top-level transaction will not affect other top-level transactions at all.

Infinite retry cannot occur within a nested transaction because the structure of any transaction is assumed to be finite, and in conflict we always resolve in favor of the operation with a smaller timestamp. Therefore, a subtransaction cannot be preempted by other subtransactions from the same root indefinitely.

Infinite retry between the top-level transactions, however, is a problem. It is possible that a top-level transaction might be aborted and retried indefinitely, each time failing because some new operation with a larger timestamp has show up in the data objects it accesses. This problem is intrinsic to any timestamp-based protocol, and is not aggravated in our case. Most timestamp-based systems assume it to be negligible. How serious this problem is for our method can only be measured in a real implementation, and is remained for further investigation.

7 Conclusions

In this paper, we propose a new concurrency control protocol for nested transactions. Our method try to address three design goals: to minimize execution times of individual transactions, the time a transaction has to spend on resource waiting, and the number of cases in which a transaction proceeds based on the assumption that some other transaction will commit. These goals will improve both program response times and system throughput. We meet these goals by exploiting intra-transaction concurrency, and using the concept of request lists, which is derived from the history mechanism for single-level transactions and merged with a timestamp serialization protocol. It preserve the benefits or both methods. Problems in locking and timestamp concurrency control protocols, such as deadlocks and cascading aborts, are avoid altogether. Most notably, our method decouples the serialization protocol for top-level transaction from that for subtransaction within the root, and decouples the

issuing of subtransactions from the synchronization of commitment. That is, we need not be concerned with how subtransactions are issued as long as they are committed in the required order.

References

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

[2] P. A. Bernstein, D. W. Shipman, and D. W. Wong, "Aspects of serializability in database concurrency conctrol," *IEEE Trans. on Software Engineering*, vol. 5, no. 3, pp. 203–216, May 1979.

[3] W. Cellary, E. Gelenbe, and T. Morzy, "Concurrency control in distributed database systems, chap 1-7 and chap 11-12," *North-Holland*, 1988.

[4] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. on Database Systems*, vol. 6, no. 2, pp. 213–226, June 1981.

[5] B. Liskov and R. Schiefer, "Guardians and actions: Linguistic support for robust, distributed programs," *Transactions on Programming Languages and Systems, Vol. 5, No. 3*, pp. 381–404, July 1983.

[6] J. Moss, "Nested transactions: An approach to reliable distributed computing," *MIT Press*, 1985.

[7] T. Ng, "Using history to implement atomic objects," *ACM, TOCS*, Nov 1989.

[8] C. H. Papadimitriou, *The Theory of Concurrency Control*, Computer Science Press, Rockville, MD, 1986.

[9] C. Papadimitriou, "Database concurrency control," *Computer science press*, 1990.

[10] D. P. Reed, "Implementing atomic object on decentralized data," *ACM, Trans. on Computer Systems*, vol. 1, no. 1, pp. 3–23, Feb. 1983.

[11] W. Weihl and B.Liskov, "Implementation of resilient, atomic data types," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 2, pp. 244–269, April 1985.

About the authors

Ming-Ling Lo is a Ph. D. candidate in Computer Science at the University of Michigan. His research interests include parallel and spatial databases, transaction processing, and distributed systems.

Ming-Ling received his B. S. degree in Electrical Engineering from National Taiwan University in 1985. He can be reached at the EECS department, University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109-2122, or by e-mail at mingling@eecs.umich.edu.

Chinya V. Ravishankar is presently Assistant Professor in the Electrical Engineering and Computer Sciences Department at the University of Michigan. His research interests include distributed systems, programming language design and software tools.

Ravishankar received his B. Tech. degree in Chemical Engineering form the Indian Institute of Technology-Bombay, in 1975, and his M.S. and Ph.D. degrees in Computer science from the University of Wisconsin-Madison in 1986 and 1987, respectively. He can be reached at the EECS department, 1301 Beal Avenue, Ann Arbor, MI 48109-2122, or by e-mail at ravi@eecs.umich.edu.