Research Issues for a Real-Time Nested Transaction Model

Yu-Wei Chen Le Gruenwald

School of Computer Science
The University of Oklahoma
Norman, OK 73019

Abstract

Real-time database systems has been recently studied to meet stringent timing and reliability constraints that observed in many application areas. In some applications, many transactions must be not only executed correctly but also completed within their deadlines, and some data may become invalid after a certain duration of time. For some of these advanced applications, transactions are long and complicated. A transaction model beyond the conventional single-level transaction models is needed to manage the complexity. In order to extend current real-time transactions to a nested transaction model, several issues need to be explored or re-examined. In this paper, we discuss two important issues: how to propagate deadlines from a transaction to its subtransactions and how to control concurrent execution of nested transactions.

1. Introduction

In traditional real-time database systems, a transaction is considered as a flat single unit of task, which consists of a sequence of primitive actions (e.g., reads and writes of simple data objects), with a given deadline and/or a criticalness. The system usually schedules transactions according to the given characteristics of individual transactions. Due to database consistency requirements, transaction rollbacks and restarts are inevitable. Therefore, even if all transactions are initially schedulable, those rollbacks and restarts may cause transactions to miss their deadlines. In order to reduce the cost of those events, a transaction model beyond the flat model is desired to maximize system performance.

Nested transactions form a hierarchy of pieces of work. For a nested transaction, there is a top-level transaction controlling the whole transaction. Transactions nested within it are lower-level transactions, called subtransactions. Each subtransaction consists of either or both primitive actions and subtransactions. Unlike those in single-level transaction systems in which if any piece of a transaction fails the whole transaction fails, a subtransaction's failure only affects itself and its descendants. This provides failure independence and allows subtransactions to run concurrently. Nested transactions provide a powerful mechanism for both fine-tuning the scope of rollbacks and safe intra-transaction concurrency in applications with complex structures. These advantages make nested transaction model especially suitable for real-time complex and/or distributed environments.

Furthermore, it has been shown in [PLC92] that, for scheduling policies based on the earliest deadline principle, longer transactions are discriminated against shorter transactions. Pang et. al.[PLC92] developed a dynamic priority assignment scheme which improves the chances for long transactions to meet their time constraints by shortening deadlines of long transactions. Kao and Garcia-Molina [KaGM93] have also proposed a model to overcome the bias against long tasks. However, in order to avoid this kind of discrimination, a nested transaction approach, which may decompose a long transaction into several shorter transactions each of which has a shorter deadline is more reasonable than that in [PLC92] where transaction deadlines are altered. Comparing to the model in [KaGM93], a nested transaction model is more suitable for applications dealing with transactions.

Most of current real-time theories are done using the flat transaction model. To extend them to that of nested transaction model, several issues need to be explored or re-examined. In this paper, we discuss and present solutions for two important issues: how to propagate deadline from a transaction to its subtransactions and how to control concurrent execution of nested transactions. The paper is organized as follows. First, a nested transaction model is given in section 2. In section 3, a discussion of transaction deadline propagation is presented. In section 4, we deliberate conflict resolution problems. Finally, in section 5 we give our future research directions.

2. Nested Transaction Models

In the traditional transaction model, a transaction consists of a set of partially ordered atomic read and

0-8186-6375-8/94 $04.00 © 1994 IEEE

write operations. Flat transactions are those that have a single start point and a single termination point. Nested transactions extend the flat transactions by allowing a transaction to invoke atomic transactions as well as atomic operations. A nested transaction may contain any number of subtransactions, and every subtransaction, in turn, may consist of any number of subtransactions. The whole transaction therefore forms a tree, called a transaction tree, of (sub)transactions. In the following discussion, the terminologies defined in [Moss85] are used. The root transaction which is not enclosed in any transaction is called *top-level transaction* (TL-transaction). Transactions with no subtransactions are called *leaf transactions*. Transactions having subtransactions are called *parents*, and their subtransactions are their *children*. *Superiors* of a given subtransaction include all transactions on the path of the subtransaction to the root but not including itself. *Inferior* of a transaction are those transactions each of which is part of the subtransaction hierarchy spanned by the transaction and does not include itself. The *ancestors* (*descendants*) of a transaction are superiors (inferiors) of the transaction and the transaction itself. In this context, we will use the term 'transaction' to denote both TL-transactions and subtransactions.

The TL-transaction have all the properties: atomicity, consistency, isolated execution, and durability[HaRe83]. Therefore, TL-transactions must be isolated from each other, and, in case of failures, they must be rolled back without side-effects to other transactions. Subtransactions appear atomic to the surrounding transactions and may commit or abort independently. A transaction is not allowed to commit until all its children have terminated (committed or aborted). However, if a child is forcibly aborted or fails (e.g., misses deadline), its parent is not required to abort. Instead, the parent is allowed to perform its own recovery. The possible choices of the parent includes the following (1) retry the subtransaction, (2) initiate another subtransaction that implements an alternative action, (3) ignore conditions of failures, and (4) abort. The commit of a subtransaction depends on the outcome (commit or abort) of its superiors; even if it commits, aborting one of its superiors will undo its effects. All updates of a subtransaction become permanent only when the enclosing TL-transaction commits.

In the nested transaction model defined in [Moss85], actual work can only be done by the leaf-level transactions. Only they can directly manipulate objects. Higher-level transactions only organize the control flow and determine when to invoke which subtransaction. In order not to lose generality, he suggested that should one desire a parent to perform some actual work, a new child can be introduced to perform the action on the parent's behalf. Although this might solve the problem, it degrades system performance since the invocation of an additional level of nesting just for some simple primitive operations is expensive.

In [HaRo87], Harder and Rothermel proposed a more robust model, which allows actual work done at any level. They used client-server to model invocation methods. A client can invoke a server either synchronously or asynchronously. An invocation is said to be synchronous if a calling transaction (parent) is suspended until the invoked transaction finishes. For an asynchronous invocation, the calling transaction will not be blocked by the invoked transaction.

Both invocation mechanisms are desired in distributed real-time environments. The asynchronous invocation increases parallelism, which may, in turn, increase the chances that transactions meet their deadlines. For example, in a target tracing system, the application of looking for targets and registering them can be decomposed so that they can run asynchronously. A desired decomposition is described as follows. First, the possible direction of the object is estimated by extrapolating certain previously registered positions, and it spots the object's position. Then the transaction creates an asynchronous child transaction to register the object's position and continues to trace the object. The advantages of this decomposition are: (1) registering and tracing are performed in parallel, (2) the failure of registering would not affect the tracing. In other situations, synchronous invocation is necessary. For instance, if the validity of a data item expires which is currently needed by a transaction, instead of aborting the transaction, we may trigger a subtransaction to update the outdated data item. Since the original transaction needs up-to-date data and cannot proceed until the triggered transaction has been finished, this invocation should be synchronous. Due to the powerfulness of this approach, a nested transaction model similar to that of Harder and Rothermel will be used in our research in the context of distributed real-time database systems. Three basic rules of the nested transaction model are described as follows.

- Commit rule: when a subtransaction complete successfully, it is said to have committed, although such commitment is relative: any updates of the database become permanent only if the enclosing top-level transaction commits. The results of a committed subtransaction is accessible to its parent.
- Rollback rule: when a (sub)transaction is rolled back, all descendants of this transaction are rolled back, regardless of their commit status.
- Visibility rule: all changes made by a subtransaction become visible to its parent after the subtransaction commits. Siblings will not see the results made by each other. A subtransaction can access data held by its superior. After the subtransaction has accessed the data, the holder of the data is no longer the

original holder but the subtransaction. Thereafter, the previous visibility rules apply.

3. Deadline Propagation of Real-Time Nested Transactions

Since each subtransaction acts as a unit to compete for resources, it should possess its own deadline. Applications may or may not assign a deadline to each subtransaction. If deadlines of some subtransactions are not assigned, a propagation routine is needed to fill them up. Even if they are all assigned, it is still necessary to examine them to make sure they are consistent with each other; for instance, deadline of a child transaction should not be longer than that of its parent. We will discuss deadlines propagation schemes in the following subsections.

3.1 Absolute Deadline Propagation

A simplest way of deadline propagation is that a whole family have the same deadline which is associated with the top-level transaction. We call this deadline absolute deadline.

Absolute Deadline = Deadline of Top-Level Trans. - Adjust.

where *Adjust* includes communication (if any) and commit delay.

Apparently, this approach has problems. Assume a parent invokes a child synchronously. Since the child has the same deadline as it parent, it may finish within its deadline but after it has finished there may not be enough time left for the parent to finish before the parent's deadline expires.

3.2 Normal Deadline Propagation

This approach of deadline propagation can be described as follows.

$$D(T_{child}) = D(T_{parent}) - CAdj - Adjust$$

where $D(T)$ denotes the deadline of transaction T; *Adjust* includes communication (if any) and commit delay; and *Cadj* is conflict adjust time which will be described in the following.

- A parent and its child run parallelly and they have no conflicts after the child's invocation: since they would not block each other due to data conflicts, the child would not cause the parent to miss its deadline if the child finishes before the parent's deadline. Therefore, we have
$$CAdj = 0$$
- (1) A parent and its child run parallelly but they may have conflicts after the child's invocation, or (2) A parent invokes its child synchronously: since the parent would be blocked by the child or the parent has to wait for the child to finish and then resumes its work, in order to let the parent have enough time to finish before its deadline, the difference between the child's latest finish time and the parent's deadline should be greater than or equal to the remaining runtime estimate. Depending on the types of parent and its child, the assignment of *Cadj* is as follows.
  - Both parent and child are either soft or firm: they will share the parent's slack time.
$$CAdj = ES(s) + parent's\ slack\ time \times \frac{ES(s)}{ES(s) + ES(child)}$$
where *ES(s)* is the runtime estimate of the parent's remaining portion s after invoking the child, and *ES(child)* is the runtime estimate of the child. Note that the runtime estimate of portion s and child's runtime should include that of all subtransactions invoked in portion s and the child, respectively.
  - Parent is soft and child is firm: the whole slack time is given to the child, since the child will be aborted if it cannot meet the deadline.
$$CAdj = ES(s)$$
  - Parent is firm and child is soft: the slack time is given to the parent, since (1) the child's missing deadline is not fatal, (2) the child's shorter deadline may increase its priority and possibly finish earlier, and thus would increase it's parent's chance to finish in time.
$$CAdj = ES(s) + parent's\ slack\ time$$

The final deadline would be either the one from the above calculation or the one specified by the application depending on which is earlier.

3.3 Average Deadline Propagation

When the normal deadline propagation is used, some undesired situations might occur because the relative order of the priorities for different transaction families is not consistent. For example, there are two transactions, T1 and T2, which have subtransactions, T11 and T21, respectively. T1 and T21 execute on Node A, while T11 and T2 execute on Node B. If a priority assignment procedure uses deadline as the criterion, under a unified propagation, which means a whole transaction family have the same deadline, either the priority of transaction family T1 is higher than that of transaction family T2 or the priority of transaction family T2 is higher than that of transaction family T1. On the other hand, under the

normal propagation scheme, it is possible that on Node A the priority of T21 is higher than of T1 and on Node B the priority of T12 is higher than that of T2. Assuming T1 and T21 have data conflicts, T11 and T2 have data conflicts, and a priority abort conflict resolution scheme is employed, mutual aborts may happen. That is T11 aborting T2 and T21 aborting T1, and thus results in both transaction families T1 and T2 being aborted. It is also possible that on Node A the priority of T1 is higher than that of T21 and on Node B the priority of T2 is higher than that of T11. Assuming T1 and T21 have data conflicts as well as T11 and T2 have data conflicts, T21 has to wait for T1 to release the conflict data, and T11 has to wait for T2 to release the conflict data. But T1 and t2 cannot release the conflict data because their inferiors need more data (two phase locking). Therefore, a deadlock occurs. The outcomes which are caused by this priority reversal are not desired in real-time database environments. The average deadline propagation avoids this priority reversal problem by assigning every transaction in the same family the same deadline.

$$Average\ Deadline = \frac{1}{n}\sum_{i=1}^{n} deadline(T_i)$$

where n is the total number of potential subtransactions in the transaction family T, and T_i 's are substransactions of T. The deadlines of subtransactions are derived from the normal deadline propagation scheme, and are called virtual deadlines in order to distinguish with actual deadlines.

4. Conflict Resolutions

The serializability of nested transactions can be ensured via locking protocols. The nested transaction model we adopted here is from the work done by Harder and Rothermel [HaRo93]. They extended the locking rules proposed in [Moss85] to include downward inheritance. Readers are referred to [HaRo93] for detailed locking rules.

Lock-based concurrency control would cause problems under the environment of real-time nested transaction model. Some of the problems and their solutions are discussed in the following sections.

4.1 Priority Inversion Problem

Here a transaction's priority is said to be the transaction's position in the ready queue, which is unique and resulted from a priority assignment scheme. In the nested transaction model, a parent and its children can run concurrently. Therefore, they should have different priorities. Based on different priority assignment schemes, a parent may have either a higher or lower priority than those of its children.

Most real-time systems utilize priorities to maintain predictability. The fact that higher priority transactions must be executed before lower priority transactions is essential for the correctness of real-time systems. In many database systems, the two-phase locking protocol has been used to guarantee serializability. However, when a priority-driven scheduling scheme and the two-phase locking protocol are integrated together in a real-time database system, a potential problem of creating priority inversion arises in which a higher priority transaction is blocked by lower priority ones. The situation contradicts the purpose of priority assignment which ensures that higher priority transactions execute before lower priority ones. Even worse, this kind of blocking delay may be unbounded [SoCh90].

Similar to that in a flat transaction model, the unbounded priority inversion problem also happens in a nested transaction model and can be illustrated by the example given in Figure 1 in which T denotes a transaction or subtransaction, $R[d]$ and $W[d]$ read lock and write lock on data item d, respectively.

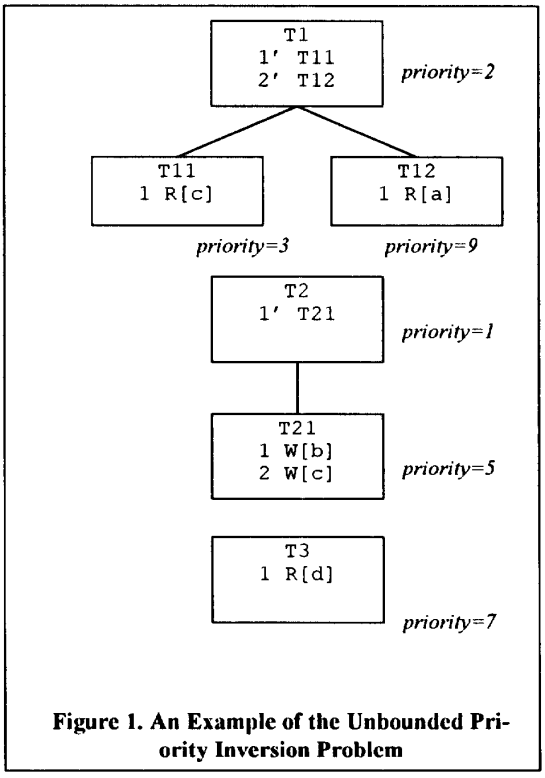

Figure 1. An Example of the Unbounded Priority Inversion Problem

In this example, T2 invokes T21 with priority 5 and waits for the later to commit. Now, T1 with priority

2 invokes T11 and T12 which are assigned priorities 3 and 9, respectively. Then, T1 waits for T11 and T12 to commit. After T11 commits, the read lock of data item *c* is inherited by T1. T21 gains the CPU but it finds out that it has a conflict with T1 on data item *c* which is retained by T1; therefore, it waits for T1 to release *c*. Now T12 with priority 9 gains the CPU. At this moment, T3 arrives and is assigned priority 7. Since there are no conflicts between T3 and T12, and T3 has a higher priority than T12 does, T3 preempts T12. Thus, T2 can be indirectly blocked by some lower-priority transactions indefinitely even if they have no conflicts with it. Some approaches to avoid this problem are discussed in the following subsections.

4.2 Priority Inheritance Protocol

In the above example, T2 has to wait for its child transaction T21 to commit; which causes a priority inversion situation to occur. To avoid this, when a transaction is waiting for its children to commit and the parent has a higher priority, all its inferiors are promoted to higher priorities similar to its own priority. We do not use the "same priority" because a priority is unique in our system. The relative order among those inferiors will be kept the same. In case its inferiors have higher priorities than it, the inferiors run on their own priorities.

For other situations which also occur in flat transaction models such as a higher priority transaction is blocked by a lower priority one due to data conflicts, a scheme similar to the priority propagation scheme proposed in [Haqu93] can be used to avoid unbounded priority inversion. In this scheme, the lower-priority one inherits the higher-priority and immediately propagates that priority to all its family; the entire family then would execute at the higher priority. Thus, the relative priority order will be preserved. The transitive property of the priority inheritance protocol in flat transaction models also applies in the nested transaction model.

4.3 Priority Abort Protocol

Whereas an abort in flat transaction models will not cause cascading aborts if the strict two-phase locking protocol is employed, in nested transaction models aborting a subtransaction may lead to aborting several subtransactions. These aborts may not only involve the subtransaction's descendants but also its ancestors and siblings. The following example illustrates the problem.

- T11 has already completed. T12 completes and a read lock of data item *d* and write lock of data item *a* are inherited by T1. Then, T1 keeps those read/write locks in the retain mode.
- T1 requests and is granted a read lock on data item *a*. Now, T1 keeps the read lock on data *a* in the hold mode.
- T1 invokes subtransaction T13. T13 requests and is granted a read lock on data item *b*.
- T2 with priority equal to 3 arrives and obtains the CPU. T2 requests a write lock on data item *b* which is held by T12.

Assuming T11 does not include step 2, if T2 aborts T13, then T13 is the only subtransaction needed to be aborted and restarted. If T11 does include the 2nd operation (write *b*), then T11 and T13 should be aborted (conflict on data item *b*). Due to T12's reading data item *d* which has been modified by T11, T12 should also be aborted. T12's abort finally will cause T1 to be aborted because of T1 reading data item *a* which was written by T12.

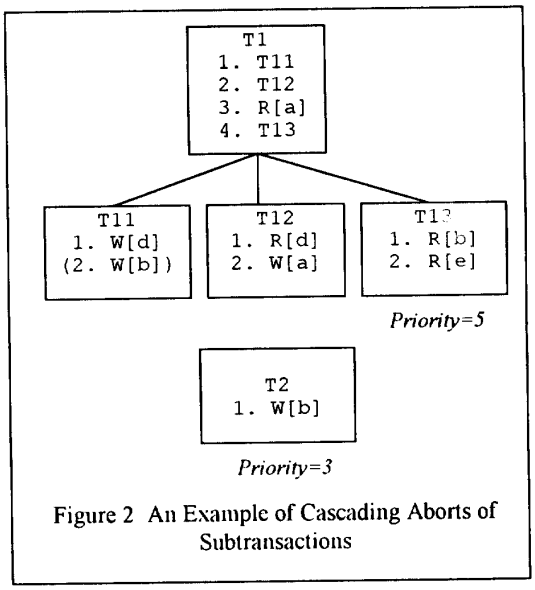

Figure 2 An Example of Cascading Aborts of Subtransactions

From the above example, aborting subtransactions can cause other subtransactions of the same family to be aborted, and even though some of these subtransactions may have committed. Therefore, an efficient search method needs to navigate through a transaction tree to determine which subtransactions should be aborted when a subtransaction aborts.

The priority aborted protocol prevents priority inversion by aborting low priority transactions when conflicts occur[AbGM88]. In the nested transaction model, aborting a low-priority transaction by a median-priority transaction may cause high-priority transactions to be

cascadingly aborted. This violates the spirit of priority abort protocol. Following is a solution to this problem.

For every transaction, there are two priorities associated with it. One, called assigned priority, is given by a priority assignment procedure, and the other one, called virtual priority, is derived from the context of its family. The virtual priority of a transaction is the maximum assigned priority of its descendants, and when a subtransaction inherits a lock from one of its superiors, it also inherits that superior's virtual priority. The CPU dispatcher is based on the assigned priorities, and conflict resolution is based on virtual priorities. When two transactions have conflicts, if the lower-virtual-priority transaction holds the needed data, the transaction with a higher virtual priority would abort the one with a lower virtual priority. Otherwise, the transaction with a lower virtual priority would wait. The above scheme may have the priority inversion problem. For example, a low-assigned-priority but high-virtual-priority transaction may block a high-assigned-priority but low-virtual-priority transaction. In order to avoid the priority inversion problem, the priority inheritance protocol discussed in the previous section would be incorporated into this scheme.

5. Further Work

The normal deadline propagation scheme preserves the true urgency of transactions but suffers from the priority reversal problem. the absolute and average deadline propagation schemes avoid the priority reversal problem but cannot reflect transactions' true urgency. We are currently conducting simulations to compare the performances of different combinations of deadline propagation and conflict resolution schemes. The results may help us understand these effects better.

References

[AbGM88] R. Abbott, H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," 14th Int. Conf. on Very Large Data Bases, 1988, pp. 1-12.

[Haqu93] W. ul Haque, Transaction Processing in Real-Time Database Systems, Ph.D. Dissertation, Department of Computer Science, Iowa State University, 1993.

[HaRe83] T. Harder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," ACM Computing Surveys, vol. 15, no. 4, pp. 287-318, 1983.

[HaRo93] T. Haerder and K. Rothermel, "Concurrency Control Issues in Nested Transactions," VLDB Journal, vol. 2, no. 1, pp. 39-74, 1993.

[HSRT91] J. Huang, J. Stankovic, K. Ramamritham, D. Towsley, "On Using Priority Inheritance in Real-Time Database," 12th Real-Time Systems Symposium, 1991, pp. 210-221.

[KaGM93] B. Kao, H. Garcia-Molina, "Deadline Assignment in a Distributed Soft Real-Time System," In Proc. of the 13th Int'l Conf. on Distributed Computing Systems, pp. 428-437, 1993.

[Moss85] J.E.B. Moss, Nested Transactions: An Approach to Reliable Computing, MIT Press, 1985.

[PLC92] H. Pang, M. Livny, M.J. Carey, "Transaction Scheduling in Multiclass Real-Time Database Systems," In Proc. of the 13th Real-Time Systems Symposium, Dec. 1992.

[Rama92] K. Ramamritham, "Real-Time Databases," International Journal of Distributed and Parallel Databases.

[SoCh90] S.H. Son and C. Chang, "Performance Evaluation of Real-Time Locking Protocols Using a Distributed Software Prototyping Environment," 10th Int'l. Conf. on Distributed Computing Systems, June, 1990, pp. 124-131.

[SRL88] L. Sha, R. Rajkumar, J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," ACM SIGMOD Record, March 1988, pp. 82-98.

[UlBe92] O. Ulusoy, G. Belford, "Real-Time Lock-Based Concurrency Control in Distributed Database Systems," 1992, pp. 136-143.