

Using Real-Time Serializability and Optimistic Concurrency Control in Firm Real-Time Databases

Jan Lindström and Kimmo Raatikainen

University of Helsinki, Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland
E-mail: {jan.lindstrom,kimmo.raatikainen}@cs.Helsinki.FI

17th March 2000

Abstract

In a real-time database system, it is difficult to meet all timing constraints due to the consistency requirements of the underlying database. Real-time database transaction scheduling requires the development of efficient concurrency control protocols that try to maximize the number of transactions satisfying their real-time constraints. However, when the transactions in the system are heterogeneous, they are not of the same importance - some are of greater importance than others. Therefore, transaction importance should be taken into account in concurrency control. Although an optimistic approach has been shown to be better than locking protocols in real-time database systems, optimistic methods have the problems of unnecessary restarts and heavy restart overhead. In this paper, we propose a real-time serializability and a new optimistic concurrency control protocol called OCC-RTDATI (Optimistic Concurrency Control with Real-Time Serializability and Dynamic Adjustment of Serialization Order) which uses real-time serializability in the conflict resolution. In OCC-RTDATI the number of transaction restarts is minimized by dynamically adjusting the serialization order of the conflicting transactions. The need for dynamic serialization order adjustment is checked and the serialization order is updated in the validation phase. This provides more freedom to adjust the serialization order of conflicting transactions. These features are important and increase the probability that transactions of high importance will be completed before their deadlines.

Keywords: real-time databases, concurrency control, transaction processing.

1. Introduction

Concurrency control is one of the main issues in the studies of real-time database systems. With a strict consistency requirement defined by serializability [5], most real-time concurrency control schemes considered in the literature are based on two-phase locking (2PL) [9]. 2PL has been studied extensively in traditional database systems and is being widely used in commercial databases. However, 2PL has some inherent problems, such as the possibility of deadlocks as well as long and unpredictable blocking times. These problems appear to be serious in real-time transaction processing since real-time transactions need to meet their timing constraints, in addition to consistency requirements.

Many alternatives to two-phase locking in real-time systems have been proposed and studied [4, 11, 17]. Among them is a class of concurrency control schemes based on the optimistic approach [10, 19].

Optimistic concurrency control protocols have the nice properties of being non-blocking and deadlock-free. These properties make them especially attractive for real-time database systems (RTDBS). Because conflict resolution between the transactions is delayed until a transaction is near to its completion, there will be more information available in making the conflict resolution. Although optimistic approaches have been shown to be better than locking protocols for RTDBSs, they have the problem of unnecessary restarts and heavy restart overhead. This is due to the late conflict detection, which increases the restart overhead since some near-to-complete transactions have to be restarted. Thus, the major concern in designing real-time optimistic concurrency control protocols is not only to incorporate information about the importance of transactions for conflict resolution, but also to design methods that minimize the number of transactions to be restarted.

The deadlines in any real-time system can be divided into three types: *hard*, *soft*, and *firm* deadlines [1]. A hard deadline means that a task may cause a very high negative value to the system if the computation is not completed before the deadline. A soft deadline implies that, after the deadline, the computation has a decreasing value, which gradually goes down to zero. A firm deadline is between these poles; a task loses its value after the deadline but no negative consequence will occur. In this paper we concentrate on firm deadlines of real-time transactions. We propose a new optimistic concurrency control protocol called OCC-RTDATI (Optimistic Concurrency Control with Real-Time Serializability and Dynamic Adjustment of Serialization Order), which uses real-time serializability in the conflict resolution.

The rest of the paper is organized as follows. In Section 2 we discuss recent related work and introduce some notations for the rest of the paper. In Section 3 we introduce the basic mechanisms of the proposed optimistic concurrency control. In Section 4 we describe how the real-time serializability is taken into account in the OCC-RTDATI algorithm.

2. Using Priorities in Concurrency Control

In *Optimistic Concurrency Control* (OCC) [19] transactions are allowed to execute until they reach their commit point, at which time they are validated. The execution of a transaction consists of three phases, the *read phase*, the *validation phase*, and the *write phase*. The read phase is the normal execution of the transaction. Write operations are performed on private data copies in the local workspace of the transaction. This kind of operation is usually called *pre-write*. The validation phase ensures that all the committed transactions have executed in a correct (usually serializable) order. During the write phase, all changes made by the transaction are permanently stored into the database.

In this section, we discuss recent related work, present some problems when using priority driven concurrency control, we develop the basic notation modified from [5] that will be used in the remainder of the paper. We present a new real-time model for histories and a new real-time model for serializability. Finally we present a dynamic adjustment method in optimistic concurrency control.

2.1. Related work

In recent years, various real-time concurrency control protocols have been proposed for single-site RTDBS by modifying some well-known non-real-time concurrency control protocols so that the priority of transactions affect the conflict resolution. These modifications can be divided into two basic categories of concurrency control protocols: locking [9], and optimistic [19, 10].

Most of the proposed modifications are based on *dynamic two phase locking* (D2PL). Most probably this is due to the simplicity of D2PL and its popularity for conventional database systems. One

of the earliest proposals in real-time transaction processing protocol was *Hight Priority Two Phase Locking* (H2PL) [2, 30]. In addition, to reduce blocking time, *priority inheritance* [15] and *conditional priority inheritance* [15] have been suggested. In these protocols a lower priority transaction inherits a priority of the higher priority transaction. *Hybrid Two Phase Locking* (HB2PL) [16] adopts the concept of cautious waiting to prevent locking induced trashing and to reduce the impact of priority inversion. Other ideas include *Two Phase Locking with Ordered Sharing* (2PL-OS) [3], *Two Phase Locking with Ordered Sharing and Before Images* (2PL-OS/BI) [3], *Two Phase Locking with Avoiding Cascaded Aborts* (ACA 2PL-OS) [3], and *Preemptive Two Phase Locking* (P2PL) [27]. In all these proposals the importance of transactions is the same as the scheduling priority.

Priority-driven concurrency control protocols based on the optimistic approach have not been widely studied. The exceptions include *Optimistic concurrency control using locking* (OCCL) [14] which is based on a locking mechanism, *Optimistic Concurrency Control with Adaptive Priority* (OCC-APR) [8], *Adaptive Priority Fan Out* (OCC-APFO), and [7] *Adaptive Priority Fan-out Sum* (OCC-APFS) [7]. OCC-APR and its variants attempt to reduce the number of restarts by exploiting and extending the capabilities of priority insensitive algorithms.

The protocol presented in this paper is related to two previously presented optimistic concurrency control protocols, namely OCC-TI and OCC-DA. OCC-TI (Optimistic Concurrency Control with Timestamp Intervals) [23] is based on the forward validation scheme. The number of transaction restarts is reduced by using dynamic adjustment of the serialization order. OCC-DA (Optimistic Concurrency Control with Dynamic Adjustment) [20, 21] is also based on the forward validation scheme and on the use of dynamic adjustment of the serialization order.

2.2. Problems in Priority driven Concurrency Control

In real-time database systems, the conflict resolution should take into account the importance of the transactions. This is especially true in the case of heterogeneous transactions. Some transactions are more important or valuable than others. Therefore, the goal of any real-time system should be to maximize the value or importance of the completed transactions. In most of the previous approaches the value or importance of a transaction has been equalized to the scheduling priority of a transaction. Unfortunately, that is a very serious restriction if the target is to maximize the value or importance of the completed transactions. Therefore, we use transaction importance in place of scheduling priority.

Below we characterize the four typified problems and we use priority to denote either scheduling priority or importance of the transaction:

- **wasted restart:** A wasted restart occurs if a higher priority transaction aborts a lower priority transaction and later the higher priority transaction is discarded when it misses its deadline.
- **wasted wait:** A wasted wait occurs if a lower priority transaction waits for the commit of a higher priority transaction and later the higher priority transaction is discarded when it misses its deadline.
- **wasted execution:** A wasted execution occurs when a lower priority transaction in the validation phase is restarted due to a conflicting higher priority transaction which has not finished yet.
- **unnecessary restart:** An unnecessary restart occurs when a transaction in the validation phase is restarted even when history would be serializable.

Traditional two-phase locking suffers from the problem of wasted restart and wasted wait. Optimistic protocols suffer the problems of wasted execution and unnecessary restart.

2.3. Real-Time History and Serializability

Let us develop the basic notation that will be used in the remainder of the paper. Let $r_i[x]$ and $w_i[x]$ denote read and write operations, respectively, on data object x by transaction T_i . Let v_i and c_i denote the validation and commit operations of the transaction T_i . Let $RS(T_i)$ denote the set of data items read by the transaction T_i and $WS(T_i)$ denote the set of data items written by the transaction T_i . Let $RTS(D_i)$ and $WTS(D_i)$ denote the largest timestamp of committed transactions that have read or written, respectively, the data object D_i . Let $TS(T_v)$ be the final timestamp of the validating transaction T_v and $TI(T_i)$ timestamp interval of the transaction T_i . Let $imp(T_i)$ be the importance of transaction T_i .

A transaction is a sequence of operations resulting from the execution of a **transaction program**. A transaction program is usually written in a high-level programming language with assignments, loops, conditional statements and other complex control structures. Execution of a transaction program starting at different database states may result in different transactions. Formally:

DEFINITION 2..1 A **transaction** $T_i = (O_{T_i}, \prec_{T_i})$, where $O_{T_i} = \{o_1, o_2, \dots, o_n\}$ is a set of operations and \prec_{T_i} is a total order on O_{T_i} . An operation o_i is a 5-tuple

$$(action(o_i), entity(o_i), value(o_i), RTS(o_i), WTS(o_i)),$$

where $action(o_i)$ denotes an operation type, which is either a read (r) or a write (w) operation. $entity(o_i)$ is the data item on which the operation is performed. If the operation is a read operation, $value(o_i)$ is the value returned by the read operation for the data item read. For a write operation $value(o_i)$ is the value assigned to the data item by the write operation. We have added two additional attributes to transactions. $RTS(o_i)$ largest timestamp of committed transactions that have read object o_i . $WTS(o_i)$ largest timestamp of committed transactions that have written object o_i . For simplicity of exposition, we assume that each transaction reads and writes a data item once at the most.

In real-time database systems, the conflict resolution should take into account the importance of the transactions. In most of the previous approaches the value or importance of a transaction has been equalized to the scheduling priority of a transaction. Unfortunately, it is a very serious restriction if the target to maximize the value or importance of the completed transactions. Thus, the major concern in designing real-time optimistic concurrency control protocols is not only to incorporate information about the importance of transactions for conflict resolution but also to design methods that minimize the number of transaction to be restarted. Therefore we define a real-time version of complete history. Formally, let $T = \{T_1, T_2, \dots, T_n\}$ be a set of transactions.

DEFINITION 2..2 A complete real-time history H over T is a partial order with ordering relation \prec_H where

1. $H = \bigcup_{i=1}^n T_i$;

2. $\prec_H \supseteq \bigcup_{i=1}^n \prec_i$; and

3. for any two conflicting operations $p, q \in H$, either:

- if $imp(p) > imp(q)$ then $p \prec_H q$;
- if $imp(p) < imp(q)$ then $q \prec_H p$; or
- if $imp(p) = imp(q)$ then either $p \prec_H q$ or $q \prec_H p$.

Condition (1) says that the execution represented by H involves precisely the operations submitted by T_1, T_2, \dots, T_n . Condition (2) says that the execution honors all operation orderings specified within each transaction. Finally, condition (3) says that the ordering of every pair of conflicting operations is determined by the importance of the transactions. Condition (3) is a new element in complete real-time history when compared to traditional history. We only allow histories where conflicting transactions are ordered based on their importance attributed. This ordering creates another partial order based on importance attribute. A *real-time history* is simply a prefix of a complete real-time history.

A complete real-time history H is *serial* if, for every two transactions T_i and T_j that appear in H , either all operations of T_i appear before all operations of T_j or vice versa. Thus, a serial history represents an execution in which there is no interleaving of the operations of different transactions. A history H is *real-time serializable* if its committed projection, $C(H)$, is equivalent to a serial history H . $C(H)$ is a complete real-time history and it is not an arbitrarily chosen complete real-time history. If H represents the execution so far, it is really only the committed transactions whose execution the database management system has unconditionally guaranteed. All other transactions may be aborted.

We can determine whether a real-time history is real-time serializable by analyzing a graph derived from the real-time history called a serialization graph. The *serialization graph* for H , denoted $SG(H)$, is a directed graph whose edges are all $T_i \rightarrow T_j (i \neq j)$ such that one of T_i 's operations precedes and conflicts with one of T_j 's operations in H and T_i has higher importance. Therefore a history H is real-time serializable if and only if $SG(H)$ is acyclic.

We have only restricted correct histories. Therefore every real-time serializable history is also serializable history as defined in [5].

2.4. Dynamic Adjustment in OCC

The major performance problem with OCC protocols is the late restart. Therefore, one important mechanism to improve the performance of an OCC protocol is to reduce the number of restarted transactions. In conventional OCC protocols many transactions are unnecessarily restarted. To give an example, let us consider the following transactions T_1, T_2 and history H_1 :

$$\begin{aligned} T_1 &: r_1[x]w_1[x]v_1c_1 \\ T_2 &: r_2[x]w_2[y]v_2c_2 \\ H_1 &: r_2[x]w_2[y]r_1[x]w_1[x]v_1 \end{aligned}$$

Based on the OCC-FV algorithm [10], T_2 has to be restarted. However, this is not necessary, because when T_2 is allowed to commit such as:

$$H_2 : r_2[x]w_2[y]r_1[x]w_1[x]v_1c_1v_2c_2,$$

then the schedule of H_2 is equivalent to the serialization order $T_2 \rightarrow T_1$ as the actual write of T_1 is performed after its validation and after the read of T_2 . There is no cycle in their serialization graph and H_2 is serializable [5].

One way to reduce the number of transaction restarts is to dynamically adjust the serialization order of the conflicting transactions. Such methods are called *dynamic adjustment of the serialization*

order. When real-time serializability is used, the method can be called *real-time dynamic adjustment of the serialization order*. When data conflicts between the validating transaction and active transactions are detected in the validation phase, there is no need to restart conflicting active transactions immediately. Instead, a serialization order can be dynamically defined. However, transactions of higher importance must always precede the transactions of lower importance.

2.5. Real-Time Transactions

There are three problems that any real-time database system must deal with: 1) resolving resource contention, 2) resolving data contention, and 3) enforcing timing constraints. In the *Real-Time Object-Oriented Database Architecture for Intelligent Networks* (RODAIN) ¹ [18, 24, 28, 29, 31] database transactions to be executed are classified according to their deadline and to their importance. Real-time transactions differ from traditional transactions in many ways. In traditional database systems the main goal of transaction scheduling is to maximize the throughput of transactions. In real-time database systems the goal is to maximize the number of transactions that are completed before their deadlines. Predictability of transaction lengths is one of the most important prerequisites for meeting the deadlines in real-time systems.

In real-time database systems transactions must be controlled by a real-time scheduler. A scheduling priority for each transaction is automatically evaluated and modified during the run time. The evaluation process is based on the characteristics of the transaction, the validity of the accessed data object, and the dynamic behavior of the system.

The RODAIN scheduling algorithm, called *FN-EDF*, is designed to support simultaneous execution of both firm real-time and non-real-time transactions. In RODAIN firm deadline transactions are scheduled according to the EDF scheduling policy [13, 12, 26]. The FN-EDF algorithm guarantees that non-real-time transactions receive a respecified amount of execution time. The FN-EDF algorithm periodically samples the execution times of all transactions. The operating system scheduling priority of a non-real-time transaction is adjusted if its fraction of execution time is either above or below the respecified target value. For each class of non-real-time transactions the target value is given as a system parameter, which can be changed while the system is running. Transaction scheduling is based on attributes in transaction objects.

The object model used in RODAIN should be OMG-compliant and at the same time offer enough flexibility to support alternative interfaces. We have chosen the ODMG 2.0 object model [6] as the core object model in our database architecture. Since ODMG does not support real-time transactions, we have designed real-time extensions. Our extension is based on two assumptions: 1) a real-time extension should include the original model completely, and 2) any application that does not want to use real-time extensions should see the database management system as an ordinary ODMG database. In the model we define real-time objects and real-time characteristics that fit into the model. The additional characteristics are designed so that the original ODMG model is a true subset of the extended model. The essential feature of our extensions is to guarantee that the real-time scheduler and the concurrency controller have enough knowledge to overcome the problems due to heterogeneous transaction durations.

The Real-time transaction object includes the attributes *priority*, *deadline*, and *importance*. Priority and deadline attributes are normal object attributes. Therefore, the values of priority and deadline attributes can be different in every instance of the transaction class. Additionally, deadline and priority attributes can be the same for several transaction class instances. However, when the transactions

¹RODAIN is funded by Nokia Telecommunications, Solid Information Technology Ltd., and the Finnish Technology Development Center (TEKES).

in the system are heterogeneous, they are not of the same importance - some are of greater importance than others. The importance attribute is a class attribute, therefore it is the same for all instances of the same transaction class. The importance of the transaction does not depend on the arrival time of the transaction as the deadline attribute does. Therefore, we use importance of the transactions in place of the priority in conflict resolution of optimistic concurrency control. These attributes are used in the real-time scheduler and the concurrency controller in our new proposed algorithm. These extensions were not used in our earlier OCC-DATI [25] algorithm.

3. OCC-DATI

In this section, we briefly present our earlier optimistic concurrency control protocol named OCC-DATI (Optimistic Concurrency Control with Dynamic Adjustment of Serialization Order) [25]. OCC-DATI uses similar dynamic adjustment of the serialization order as in OCC-TI [23]. Unlike the OCC-TI protocol, all checking is performed at the validation phase of each transaction. There is no need to check for conflicts while a transaction is still in its read phase. As the conflict resolution between the transactions in OCC-DATI is delayed until a transaction is near completion, there will be more information available for making the choice to resolve the conflict. OCC-DATI also has a new final timestamp selection method compared with OCC-TI.

OCC-DATI differs from OCC-DA [20, 21] in several ways. We have adopted time intervals as the method to implement dynamic adjustment of the serialization order instead of a dynamic timestamp assignment as used in OCC-DA. We have also used a *deferred dynamic adjustment of the serialization order*. In the deferred dynamic adjustment of the serialization order, all adjustments of the timestamp interval are done to temporal variables. The timestamp intervals of all conflicting active transactions are adjusted after the validating transaction is guaranteed to commit. If the validating transaction is aborted, no adjustments are performed. Adjustment of the conflicting transaction would be unnecessary since no conflict is present in the history after the abortion of the validating transaction. Unnecessary adjustments may later cause unnecessary restarts.

The OCC-DATI protocol resolves conflicts using the *time intervals* (TI) [22] of the transactions. Every transaction must be executed within a specific time interval. When an access conflict occurs, it is resolved using the read and write sets of the conflicting transactions together with the allocated time interval. Time intervals are adjusted when a transaction validates. This timestamp interval is used to record a temporary serialization order during the validation of the transaction. At the beginning of the validation (Figure 1), the final timestamp of the validating transaction $TS(T_v)$ is determined from the timestamp interval allocated to the transaction T_v . The timestamp intervals of all other concurrently running and conflicting transactions must be adjusted to reflect the serialization order. We set the final validation timestamp $TS(T_v)$ of the validating transaction T_v to be the current timestamp, if it belongs to the timestamp interval $TI(T_v)$, otherwise $TS(T_v)$ is set to be the maximum value of $TI(T_v)$.

The adjustment of timestamp intervals iterates through the read set (RS) and write set (WS) of the validating transaction. First we check that the validating transaction has read from committed transactions. This is done by checking the object's read and write timestamp. These values are fetched when the first access to the current object is made. Then we iterate the set of active conflicting transactions. When access has been made to the same object both in the validating transaction and in the active transaction, the temporal time interval of the active transaction is adjusted. Thus we use deferred dynamic adjustment of the serialization order. Time intervals of all conflicting active transactions are adjusted after the validating transaction is guaranteed to commit. If the validating transaction is aborted, no adjustments are done. Non-serializable execution is detected when the

timestamp interval of an active transaction becomes empty. If the timestamp interval is empty the transaction is restarted. Finally current read and write timestamps of accessed objects are updated and changes to the database are committed.

```

occdati_validate( $T_v$ )
{
  // Select final timestamp for the transaction
   $TS(T_v) = \min(\text{validation\_time}, \max(TI(T_v)))$ ;
  // Iterate for all objects read/written
  for (  $\forall D_i \in (RS(T_v) \cup WS(T_v))$  )
  {
    if ( $D_i \in RS(T_v)$ )
       $TI(T_v) = TI(T_v) \cap [WTS(D_i), \infty[$  ;
    if ( $D_i \in WS(T_v)$ )
       $TI(T_v) = TI(T_v) \cap [WTS(D_i), \infty[ \cap [RTS(D_i), \infty[$  ;
    if ( $TI(T_v) == []$ ) restart( $T_v$ );

    // Conflict checking and TI calculation
    for (  $\forall T_a \in \text{active\_conflicting\_transactions}()$  )
    {
      if ( $D_i \in (RS(T_v) \cap WS(T_a))$ )
        forward_adjustment( $T_a, T_v, adjusted$ );

      if ( $D_i \in (WS(T_v) \cap RS(T_a))$ )
        backward_adjustment( $T_a, T_v, adjusted$ );

      if ( $D_i \in (WS(T_v) \cap WS(T_a))$ )
        forward_adjustment( $T_a, T_v, adjusted$ );
    }
  }

  // Adjust conflicting transactions
  for (  $\forall T_a \in adjusted$  )
  {
     $TI(T_a) = adjusted.pop(T_a)$ ;

    if ( $TI(T_a) == []$ ) restart( $T_a$ );
  }

  // Update object timestamps
  for (  $\forall D_i \in (RS(T_v) \cup WS(T_v))$  )
  {
    if ( $D_i \in RS(T_v)$ )
       $RTS(D_i) = \max(RTS(D_i), TS(T_v))$ ;
    if ( $D_i \in WS(T_v)$ )
       $WTS(D_i) = \max(WTS(D_i), TS(T_v))$ ;
  }

  commit  $T_v$  to database;
}

```

Figure 1. Validation algorithm.

4. OCC-RTDATI

We have developed an optimistic concurrency control protocol called OCC-RTDATI, which uses real-time serializability. OCC-RTDATI is based on forward validation [10] and the earlier optimistic OCC-DATI [25] protocol. The validation protocol is the same in OCC-DATI and OCC-RTDATI. The difference is in the conflict resolution. The conflict resolution of OCC-RTDATI uses real-time serializability and transaction object attributes. This section outlines new parts of OCC-RTDATI when compared to OCC-DATI. Suppose we have a validating transaction T_v and a set of active transactions $T_j (j = 1, 2, \dots, n)$. There are three possible types of data conflicts which can cause a serialization order between T_v and T_j :

1. $RS(T_v) \cap WS(T_j) \neq \emptyset$ (read-write conflict)

A read-write conflict between T_v and T_j can be resolved by adjusting the serialization order between T_v and T_j . When $T_v \rightarrow T_j$, then the reads in T_v cannot be affected by writes in T_j . This type of serialization adjustment is called *forward ordering* or *forward adjustment*. Using information about the importance of the transactions and timestamp intervals this is done by adjusting the timestamp interval of the active transaction **forward**, i.e.

```

if ( imp( $T_v$ ) >= imp( $T_j$ ) )
     $TI = TI(T_j) \cap [TS(T_v) + 1, \infty[;$ 
     $TI(T_j) = TI$ 
else
    restart( $T_v$ )

```

The order of the conflicting transactions should be based on the importance of the transaction. A transaction of higher importance should precede a transaction of lower importance in real-time history. Therefore, the transaction of higher importance should not be forward adjusted after a transaction of lower importance. Thus, if this is the case a transaction of lower importance is restarted. This is a wasted execution, but it is required to ensure the execution of the transaction of higher importance. If a validating transaction has higher importance we make normal deferred dynamic adjustment of the serialization order.

2. $WS(T_v) \cap RS(T_j) \neq \emptyset$ (write-read conflict)

A write-read conflict between T_v and T_j can be resolved by adjusting the serialization order between T_v and T_j as $T_j \rightarrow T_v$. It means that the read phase of T_j is placed before the writes of T_v . This type of serialization adjustment is called *backward ordering* or *backward adjustment*. Using information about importance and timestamp intervals this can be done by adjusting the timestamp interval of the active transaction **backward**, i.e.

```

if ( imp( $T_v$ ) >= imp( $T_j$ ) )
     $TI = TI(T_j) \cap [0, TS(T_v) - 1];$ 
     $TI(T_j) = TI$ 
else
    restart( $T_v$ )

```

Again, we must ensure real-time serializable execution. Therefore, transactions of high importance should not be backward adjusted; instead, conflicting transactions having lower importance should be restarted. This is wasted execution and unnecessary restart, which must be acceptable when we favor transactions of high importance.

3. $WS(T_v) \cap WS(T_j) \neq \emptyset$ (write-write conflict)

A write-write conflict between T_v and T_j can be resolved by adjusting the serialization order between T_v and T_j . When $T_v \rightarrow T_j$, then the writes of T_v cannot overwrite the writes of T_j . Using information about the importance of transactions and timestamp intervals this is done by adjusting the timestamp interval of the active transaction **forward**, i.e.

```

if ( imp( $T_v$ ) >= imp( $T_j$ ) )
     $TI = TI(T_j) \cap [TS(T_v) + 1, \infty[$ ;
     $TI(T_j) = TI$ 
else
    restart( $T_v$ )

```

This case is the same as in the read-write conflict.

Figure 2 depicts an implementation outline of a real-time dynamic adjustment of the serialization order using timestamp intervals and information about the importance of the transactions.

```

forward_adjustment( $T_a, T_v, adjusted$ )
{
    if ( imp( $T_v$ ) < imp( $T_a$ ) )
        restart( $T_v$ ); /* Validation ends here */

    if ( $T_a \in adjusted$ )
         $TI = adjusted.pop(T_a)$ ;
    else
         $TI = TI(T_a)$ ;

     $TI = TI(T_a) \cap [TS(T_v) + 1, \infty[$ ;
     $adjusted.push(\{T_a, TI\})$ ;
}

backward_adjustment( $T_a, T_v, adjusted$ )
{
    if ( imp( $T_v$ ) < imp( $T_a$ ) )
        restart( $T_v$ ); /* Validation ends here */

    if ( $T_a \in adjusted$ )
         $TI = adjusted.pop(T_a)$ ;
    else
         $TI = TI(T_a)$ ;

     $TI = TI(T_a) \cap [0, TS(T_v) - 1]$ 
     $adjusted.push(\{T_a, TI\})$ ;
}

```

Figure 2. Backward and Forward adjustment for OCC-RTDATI.

5. Conclusions

Although the optimistic approach has been shown to have a better performance than locking protocols in firm real-time database systems, it has problems of unnecessary restarts and high restart overhead.

In this paper, we have proposed an optimistic concurrency control protocol called OCC-RTDATI that takes into account the importance of transactions. It has several advantages over the other concurrency control protocols. The protocol maintains all the nice properties of forward validation, a high degree of concurrency, freedom from deadlock, and early detection and resolution of conflicts, resulting in less waste of resources as well as a smaller number of restarts. All of these are important to the performance of RTDBSs and contribute to greater chances of meeting transaction deadlines.

We have presented a theoretical model for real-time histories and real-time serializability. These ideas are implemented in conflict resolution of the OCC-RDATI. These features are important and increase the probability that transactions of high importance will be completed before their deadlines.

References

- [1] Abbott, R. and Garcia-Molina, H. (1988). Scheduling real-time transactions. *ACM SIGMOD Record*, 17(1):71–81.
- [2] Abbott, R. and Garcia-Molina, H. (1992). Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560.
- [3] Agrawal, D., Abadi, A. E., and Jeffers, R. (1992). Using delayed commitment in locking protocols for real-time databases. In *ACM SIGMOD’92*.
- [4] Anderson, J. H., Ramamurthy, S., Moir, M., and Jeffay, K. (1996). Lock-free transactions for real-time systems. In *Proc. of the First International Workshop on Real-Time Databases: Issues and Applications*, March 7-8.
- [5] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.
- [6] Cattell, R. G. G., editor (1997). *The Object Database Standard: ODMG 2.0*. Morgan Kauffmann, San Francisco, Calif.
- [7] Datta, A. and Son, S. H. (1996). A study of concurrency control in real-time active database systems. Tech. report, Department of MIS, University of Arizona, Tucson.
- [8] Datta, A., Viguier, I. R., Son, S. H., and Kumar, V. (1997). A study of priority cognizance in conflict resolution for firm real time database systems. In *Proc. of the Second International Workshop on Real-Time Databases: Issues and Applications*, March 7-8.
- [9] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. (1976). The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633.
- [10] Härder, T. (1984). Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120.
- [11] Haritsa, J. R., Carey, M. J., and Livny, M. (1990). Dynamic real-time optimistic concurrency control. In *Proc. of the 11th Real-Time Symposium*, pages 94–103, IEEE Computer Society Press.
- [12] Haritsa, J. R., Carey, M. J., and Livny, M. (1992). Data access scheduling in firm real-time database systems. *The Journal of Real-Time Systems*, 4(2):203–241.

- [13] Haritsa, J. R., Livny, M., and Carey, M. J. (1991). Earliest deadline scheduling for real-time database systems. In *Proceedings of the 12th Real-Time Symposium*, pages 232–242, IEEE Computer Society Press.
- [14] Huang, J., Stankovic, J. A., Ramamritham, K., and Towsley, D. (1991a). Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the 17th VLDB Conference*, pages 35–46.
- [15] Huang, J., Stankovic, J. A., Ramamritham, K., and Towsley, D. (1991b). On using priority inheritance in real-time databases. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 210–221, IEEE Computer Society Press.
- [16] Hung, S.-L. and Lam, K.-Y. (1992). Locking protocols for concurrency control in real-time database systems. *ACM SIGMOD Record*, 21(4):22–27.
- [17] Kataoka, R., Satoh, T., and Inoue, U. (1991). A multiversion concurrency control algorithm for concurrent execution of partial update and bulk retrieval transactions. In *Proceedings of the 10th Phoenix Conference on Computers and Communications*, pages 130–136.
- [18] Kiviniemi, J., Niklander, T., Porkka, P., and Raatikainen, K. (1997). Transaction processing in the RODAIN real-time database system. *Real-Time Databases and Information Systems*, pages 355–375, London. Kluwer Academic Publishers.
- [19] Kung, H. T. and Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226.
- [20] Lam, K.-W., Lam, K.-Y., and Hung, S. (1995a). An efficient real-time optimistic concurrency control protocol. In *Proceedings of the First International Workshop on Active and Real-Time Database Systems*, pages 209–225, Springer.
- [21] Lam, K.-W., Lam, K.-Y., and Hung, S. (1995b). Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *Proceedings of IEEE Real-Time Technology and Application Symposium*, pages 174–179.
- [22] Lee, J. and Son, S. H. (1993). Using dynamic adjustment of serialization order for real-time database systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 66–75, IEEE Computer Society Press.
- [23] Lee, J. and Son, S. H. (1996). Performance of concurrency control algorithms for real-time database systems. In Kumar, V., editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, pages 429–460. Prentice-Hall.
- [24] Lindström, J., Niklander, T., Porkka, P., and Raatikainen, K. (1999). A distributed real-time main-memory database for telecommunication. In *Proceedings of the Workshop on Databases in Telecommunications*, pages 1–10, Edinburgh, UK, Co-located with VLDB-99.
- [25] Lindström, J. and Raatikainen, K. (1999). Dynamic adjustment of serialization order using timestamp intervals in real-time databases. In *Proc. of 6th International Conference on Real-Time Computing Systems and Applications*.
- [26] Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61.

- [27] Marzullo, K. (1989). Concurrency control for transactions with priorities. Tech. Report TR 89-996, Department of Computer Science, Cornell University, Ithaca, NY.
- [28] Niklander, T., Kiviniemi, J., and Raatikainen, K. (1997). A real-time database for future telecommunication services. In Ga'ti, D., editor, *Intelligent Networks and Intelligence in Networks*, pages 413–430, Paris, France. Chapman & Hall.
- [29] Raatikainen, K. (1997). Real-time databases in telecommunications. In Bestavros, A., Lin, K.-J., and Son, S. H., editors, *Real-Time Database Systems: Issues and Applications*, pages 93–98. Kluwer.
- [30] Son, S. H., Park, S., and Lin, Y. (1992). An integrated real-time locking protocol. In *Proceedings of 8th International Conference on Data Engineering*, pages 527–534, IEEE Computer Society Press.
- [31] Taina, J. and Raatikainen, K. (1996). Experimental real-time object-oriented database architecture for intelligent networks. *Engineering Intelligent Systems*, 4(3):57–63.