Dynamic Adjustment of Serialization Order using Timestamp Intervals in Real-Time Databases

Jan Lindström and Kimmo Raatikainen
Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland
fax: +358-9-70844441
Email: {Jan.Lindstrom,Kimmo.Raatikainen}@cs.Helsinki.FI

Abstract

Although an optimistic approach has been shown to be better than locking protocols for real-time database systems (RTDBS), it has the problems of unnecessary restarts and heavy restart overhead. In this paper, we propose a new optimistic concurrency control protocol called OCC-DATI. In OCC-DATI the number of transaction restarts is minimized by dynamic adjustment of the serialization order of the conflicting transactions. The need for dynamic adjustment of the serialization order is checked and the serialization order is updated in the validation phase. This provides more freedom to adjust the serialization order of conflicting transactions. OCC-DATI has several advantages over other optimistic concurrency control protocols. The protocol maintains all the nice properties with forward validation, a high degree of concurrency, freedom from deadlock, and early detection and resolution of conflicts, resulting in both less wasted resources and a smaller number of restarts. All of these are important to the performance of RTDBSs and contribute to greater chances of meeting transaction deadlines. Performance studies of our protocol have been carried out in RTDBS and the results confirm that the performance of the OCC-DATI is better than other well-known OCC protocols.

1 Introduction

A *Real-Time Database Systems* (RTDBS) processes transactions with timing constraints such as deadlines. Its primary performance criterion is timeliness, not average response time or throughput. The scheduling of transactions is driven by priority order. Given these challenges, considerable research has recently been devoted to to designing concurrency control algorithms for RTDBSs and to evaluating their performance [1, 8, 9, 10, 14, 15, 16, 18, 22]. Most of these algorithms are based on one of the two basic concurrency control mechanisms: *locking* [3] or *optimistic concurrency control* (OCC) [12].

Optimistic concurrency control protocols have the nice properties of being non-blocking and deadlock-free. These properties make them especially attractive for RTDBSs. As conflict resolution between the transactions is delayed until a transaction is near completion, there will be more information available for making the choice in resolving the conflict. However, the problem with these optimistic concurrency control protocols is the late conflict detection, which makes the restart overhead heavy as some near-to-complete transactions have to be restarted. Thus, the major concern in the design of real-time optimistic concurrency control protocols is not only to incorporate priority information for conflict resolution but also to design methods to minimize the number of transaction restarts.

In traditional databases, correctness is well defined as serializability and is the same for all transactions. However, strict serializability as the correctness criterion is not always the most suitable one in real-time databases, where correctness requirements may vary from one type of transactions to another. Some data may have temporal behavior, some data can be read although it is already written but not yet committed, and some data must be guarded by strict serializability. These conflicting requirements may be solved using a special purpose concurrency control scheme.

In this paper, we present a method to reduce the number of transaction restarts and we propose a new optimistic concurrency control protocol, called OCC-DATI. In this paper we concentrate on firm and non-real-time transaction models. OCC-DATI is a fully optimistic protocol and uses forward adjustment. With the new protocol, the number of transaction restarts is smaller than with OCC-BC, OPT-WAIT, or WAIT-50 [7, 8, 9], because the serialization order of the conflicting transactions is adjusted dynamically. On the other hand, the overhead for supporting dynamic adjustment is much smaller than the one in OCC-DA [15, 16]. We also present methods to relax serializability for telecommunication applications.

The rest of the paper is organized as follows. Section 2 introduces the basic mechanisms of our new optimistic CC protocol. Section 3 presents the OCC-DATI algorithm. In Section 4 we introduce a way to extend our optimistic protocol with relaxed serializability and semantic locking. Section 5 contains the performance study of the new protocol as compared with other well known optimistic methods. Finally, the conclusion of the paper is presented in Section 6.

2 Dynamic Adjustment of Serialization Order

In *Optimistic Concurrency Control* (OCC) [12], transactions are allowed to execute unhindered until they reach their commit point, at which time they are validated. The execution of a transaction consists of three phases, *read phase*, *validation phase*, and *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 called *pre-write*.

Let us first 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, and let v_i and c_i denote the validation and commit operations of the transaction T_i. $RS(T_i)$ denotes the set of data items read by the transaction T_i and $WS(T_i)$ denotes the set of data items written by the transaction T_i. $RTS(D_i)$ and $WTS(D_i)$ denote the largest timestamp of committed transactions that have read or written, respectively, the data object D_i. $TS(T_v)$ is the final timestamp of the validating transaction T_v and $TI(T_i)$ timestamp interval of the transaction T_i.

The major performance problem with OCC protocols is the late restart. Therefore, one important mechanism to improve the performance of OCC protocols is to reduce the number of restarted transactions. In conventional OCC protocols many transaction restarts are unnecessary. For example, consider 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 [6], T_2 has to be restarted. However, this is not necessary. Because if T_2 is allowed to commit such as:

$H_2 : r_2[x]w_2[y]r_1[x]w_1[x]v_1c_1\mathbf{v_2c_2}$,

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 [2].

One way to reduce the number of transaction restarts is to dynamically adjust the serialization order of the conflicting transactions. This method we call *dynamic adjustment of the serialization order*. When data conflict between the validating transaction and active transaction is detected in the validation phase, there is no need to restart the conflicting active transaction immediately. Instead, a serialization order can be dynamically defined. To preserve serializability with OCC protocols, if the validating transaction T_v has to be serialized before the active transaction T_j, the following two conditions must be satisfied:

1. No overwriting. The writes of T_v should not overwrite the writes of T_j.
2. No read dependency. The writes of T_v should not affect the read phase of T_j.

Suppose we have a validating transaction T_v and a set of active transactions $T_j (j = 1, 2, ..., n)$. There are three possible types of a 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)
   Read-write conflict between T_v and T_j can be resolved by adjusting the serialization order between T_v and T_j as $T_v \rightarrow T_j$ so that the read of T_v cannot be affected by T_j's write. This type of serialization adjustment is called *forward ordering* or *forward adjustment*.

   Using timestamp intervals this is done by adjusting the timestamp interval of the active transaction **forward**, i.e.

   $$TI(T_j) = TI(T_j) \cap [TS(T_v) + 1, \infty[$$

2. $WS(T_v) \cap RS(T_j) \neq \emptyset$ (write-read conflict)
   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 write of T_v. This type of serialization adjustment is called *backward ordering* or *backward adjustment*.

   Using timestamp intervals this is done by adjusting the timestamp interval of the active transaction **backward**, i.e.

   $$TI(T_j) = TI(T_j) \cap [0, TS(T_v) - 1]$$

3. $WS(T_v) \cap WS(T_j) \neq \emptyset$ (write-write conflict)
   Write-write conflict between T_v and T_j can be resolved by adjusting the serialization order between T_v

and T_j as $T_v \rightarrow T_j$ such that the write of T_v cannot overwrite T_j's write (forward ordering).

Using timestamp intervals this is done by adjusting the timestamp interval of the active transaction **forward**, i.e.

$$TI(T_j) = TI(T_j) \cap [TS(T_v) + 1, \infty[$$

3 OCC-DATI

In this section we present a new optimistic concurrency control protocol named OCC-DATI. OCC-DATI is based on forward validation [6]. The number of transaction restarts is reduced by dynamic adjustment of the serialization order which is supported by similar timestamp intervals as in OCC-TI [18]. 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 in resolving the conflict. OCC-DATI also has a new final timestamp selection method compared to OCC-TI.

The problem with the OCC-TI algorithm is best described by the example given below. Let $RTS(x)$ and $WTS(x)$ be initialized as 100. Consider transactions T_1, T_2, and history H_1:

$$T_1 : r_1[x]w_1[x]v_1c_1$$
$$T_2 : r_2[x]v_2c_2$$
$$H_1 : r_1[x]r_2[x]w_1[x]v_1.$$

Transaction T_1 executes $r_1[x]$, which causes the time interval (TI) of the transaction to be forward adjusted to $TI(T_1) = [0, \infty[\cap [100, \infty[= [100, \infty[$. Transaction T_2 then executes a read operation on the same object, which causes the time interval of the transaction to be forward adjusted similarly. Transaction T_1 then executes $w_1[x]$, which causes the time interval of the transaction to be forward adjusted to $TI(T_1) = [100, \infty[\cap [100, \infty[\cap [100, \infty[= [100, \infty[$. Transaction T_1 starts its validation, and the final timestamp is selected as $TS(T_1) = min([100, \infty[) = 100$. Because we have one read-write conflict between the validating transaction T_1 and the active transaction T_2, the time interval of the active transaction must be adjusted: Thus $TI(T_2) = [100, \infty[\cap [0, 99] = []$. Thus the time interval is shut out, and T_2 must be restarted. However this restart is unnecessary, because history H_1 is acyclic, that is serializable. Taking the minimum as the commit timestamp ($TS(T_1)$) was not a good choice here.

OCC-DA [15, 16] is based on the forward validation scheme. The number of transaction restarts is reduced by using dynamic adjustment of the serialization order. This is supported with the use of a dynamic timestamp assignment scheme. Conflict checking is performed at the validation phase of a transaction. OCC-DATI differs from OCC-DA in several ways. We have adopted time intervals as the method to implement dynamic adjustment of the serialization order instead of dynamic timestamp assignment as used in OCC-DA. The validation procedure in OCC-DA consists of four parts compared to one in OCC-DATI (we have presented the algorithm in three parts here to save space). Thus we claim that validation in OCC-DA wastes more resources than validation in OCC-DATI.

We have also used a *deferred dynamic adjustment of serialization order*. In the deferred dynamic adjustment of serialization order all adjustments of timestamp interval are done to temporal variables. The timestamp interval of all conflicting active transactions are adjusted after the validating transaction is guaranteed to commit. If a validating transaction is aborted no adjustments are done. Adjustment of the conflicting transaction would be unnecessary since no conflict is present in the history after abortion of the validating transaction. Unnecessary adjustments may later cause unnecessary restarts. OCC-TI and OCC-DA both use dynamic adjustment but they make unnecessary adjustments when the validating transaction is aborted. OCC-TI makes unnecessary adjustments even in the read phase.

OCC-DATI offers greater changes to successfully validate transactions resulting in both less wasted resources and a smaller number of restarts. All of these are important to the performance of RTDBSs and contribute to greater chances of meeting transaction deadlines.

The OCC-DATI protocol resolves conflicts using the time intervals [17] 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. In OCC-DATI every transaction is assigned a timestamp interval (TI). At the start of the transaction, the timestamp interval of the transaction is initialized as $[0, \infty[$, i.e., the entire range of timestamp space. 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 current timestamp, if it belongs to the timestamp interval

$TI(T_v)$, otherwise $TS(T_v)$ is set to be the maximum value of $TI(T_v)$.

```
occdati_validate($T_v$)
{
  // Select final timestamp for the transaction
  $TS(T_v) = min(validation\_time, max(TI(T_v)))$;
  // 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 ( $\forall T_a \in active\_conflicting\_transactions()$ )
    {
      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.

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 objects 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.

Figure 2 shows a sketch of implementing dynamic adjustment of serialization order using timestamp intervals.

```
forward_adjustment($T_a, T_v, adjusted$)
{
  if ($T_a \in adjusted$)
    $TI = adjusted.pop(T_a)$;
  else
    $TI = TI(T_a)$;

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

backward_adjustment($T_a, T_v, adjusted$)
{
  if ($T_a \in adjusted$)
    $TI = adjusted.pop(T_a)$;
  else
    $TI = TI(T_a)$;

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

Figure 2. Backward and Forward adjustment.

Using the example below we show how our algorithm works. Consider transactions T_1, T_2, and history H_1:

$$T_1 : r_1[x]w_1[x]v_1c_1$$
$$T_2 : r_2[x]v_2c_2$$
$$H_1 : r_1[x]r_2[x]w_1[x]v_1.$$

In this example T_1 reaches the validation phase first and has a write-read conflict with T_2. Therefore, T_2 must precede T_1 in the serialization history in order to avoid an unnecessary restart. Let $RTS(x)$, $WTS(x)$, $RTS(y)$, and $WTS(y)$ be initialized as 100. Assume that transaction T_1 arrives to its validation phase at time 1000. Our OCC-DATI algorithm sets $TS(T_1)$ as 1000. The validating transaction T_1 is first forward adjusted to $[1000, \infty[$. Transaction T_2

has read object x. Therefore T_2's time interval is adjusted to $[0, 999]$ using backward adjustment. Transaction T_1 updates object x timestamps and commits. Thus our OCC-DATI algorithm produces a serializable history as well as avoiding the unnecessary restart problem found in OCC-TI.

Lemma 1: Let T_1 and T_2 be transactions in a history H produced by the OCC-DATI algorithm and $SG(H)$ serialization graph. If there is an edge $T_1 \rightarrow T_2$ in $SG(H)$, then $TS(T_1) < TS(T_2)$.

Proof: If there is an edge, $T_1 \rightarrow T_2$ in $SG(H)$, there must be one or more conflicting operations whose type is one of the following three:

1. $r_1[x] \rightarrow w_2[x]$: This case means that T_1 commits before T_2 reaches its validation phase since $r_1[x]$ is not affected by $w_2[x]$. For $w_2[x]$, OCC-DATI adjusts $TI(T_2)$ to follow $RTS(x)$ that is equal to or greater than $TS(T_1)$. Thus, $TS(T_1) \leq RTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.

2. $w_1[x] \rightarrow r_2[x]$: This case means that the write phase of T_1 finishes before $r_2[x]$ executes in T_2's read phase. For $r_2[x]$, OCC-DATI adjusts $TI(T_2)$ to follow $WTS(x)$, which is equal to or greater than $TS(T_1)$. Thus, $TS(T_1) \leq WTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.

3. $w_1[x] \rightarrow w_2[x]$: This case means that the write phase of T_1 finishes before $w_2[x]$ executes in T_2's write phase. For $w_2[x]$, OCC-DATI adjusts $TI(T_2)$ to follow $WTS(x)$, which is equal to or greater than $TS(T_1)$. Thus, $TS(T_1) \leq WTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$. □

Theorem 1: Every history generated by the OCC-DATI algorithm is serializable.

Proof: Let H denote any history generated by the OCC-DATI algorithm and $SG(H)$ its serialization graph. Suppose, by way of contradiction, that $SG(H)$ contains a cycle $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 1, we have $TS(T_1) < TS(T_2) < ... < TS(T_n) < TS(T_1)$. By induction we have $TS(T_1) < TS(T_1)$. This is a contradiction. Therefore no cycle can exist in $SG(H)$ and thus the OCC-DATI algorithm produces only serializable histories. □

4 Relaxing Serializability

Traditional concurrency control methods use serializability [3] as the correctness criterion when transactions are concurrently executed. However, in real-time database systems strict serializability is not always needed because it restricts simultaneous access and induces unnecessary overhead to the system. Real-time data is often temporal and neither serializing concurrency control nor full support for failure recovery is not required because of the overhead [29]. This has led to ideas such as *atomic set-wise serializability* [27], *external versus internal consistency* [19], *epsilon serializability* [26], and *similarity* [13]. Graham [4, 5] has argued that none are as obviously correct, nor as obviously implementable, as serializability.

Due to the semantic properties of telecommunications applications, the correctness criterion can be relaxed to so called *semantic based serializability*. We decompose the semantic based serializability into two parts. Firstly, we define a temporal serializability criterion called τ-*serializability* [25], which allows old data to be read unless the data is too old. Secondly, we use a semantic conflict resolution model that introduces explicit rules, which are then used to relax serializability of transactions. The first method reduces the number of read-write conflicts whereas the second one reduces the number of write-write conflicts.

We use the τ-serializability as a correctness criterion to reduce read-write conflicts. Suppose that transaction T_A updates the data item x and gets a write lock on x at time t_a. Later transaction T_B wants to read the data item x. Let t_b be the time when T_B requests the read lock on x. In τ-serializability the two locks do not conflict if $t_a + \min(\tau_b, \tau_x) > t_b$. The tolerance $\min(\tau_b, \tau_x)$ specifies how long the old value is useful, which may depend both on data semantics (τ_x) and on application semantics (τ_b).

We have adopted the model of semantic-based concurrency control presented earlier [20, 28]. We use two semantic levels for write operations: *update* and *replace* semantics. Update semantics is like a write operation in traditional databases. Replace semantics is used when the new value of an object does not depend on any value of any object in the database. This semantics is like a blind write operation but the transaction can perform a read operation before replacing the object. This semantic model is specified as an attribute of the transaction. Thus, all write operations in the same transaction have the same semantics.

These novel ideas are utilized with the optimistic concurrency control protocol designed for RTDBS. We implemented semantic conflict resolution and τ-serializability modifications to both OCC-DA and OCC-DATI algorithms. Modified algorithms are named OCC-τDA and OCC-τDATI respectively.

5 Results from Experiments

We have carried out a set of experiments in order to examine the feasibility of our algorithms in practice. *Real-Time Object-Oriented Database Architecture for Intelligent Networks* (RODAIN) [21, 11, 23, 24, 30] is an architecture for a real-time, object-oriented, fault-tolerant, and distributed database management systems. RODAIN consists of a main-memory database, priority based scheduling and optimistic concurrency control. All experiments were executed in the RODAIN prototype database running on a machine with one processor Pentium Pro 200MHz and 64 MB of main memory with the Chorus/ClassiX operating system. The test environment is a reduced subset of the RODAIN architecture.

All transactions arrive at the RODAIN prototype through a specific user subsystem. In these tests we have used a special user subsystem that receives the arriving transactions from an off-line generated test file. Every test session contains 10 000 transactions and is repeated at least 20 times. Reported values are means of the repetitions. In particular, we examined how well our OCC-DATI algorithm performs when compared to the OCC-TI [17] and OCC-DA [15] algorithms.

The test database represents a typical Intelligent Network (IN) service. The size of the database is 30 000 objects. We used three different transactions named R1, W1 and T1. Transaction R1 is a read-only service provision transaction that reads few objects and commits. Transaction W1 is an update service provision transaction that reads few objects, updates them and commits. Transactions R1 and W1 are firm real-time transactions. Transaction T1 is a service management transaction that reads and updates 300 objects and commits. Transaction T1 is a non-real-time transaction.

New transactions are accepted up to a prespecified limit, which is the number of installed Transaction Processes. If there is no Transaction Process available when a new transaction arrives, the transaction is aborted if the arriving transaction has lower priority. If the arriving transaction has higher priority, one of the lower priority active transactions is aborted. Transactions are validated atomically. If the deadline of a transaction expires, the transaction is always aborted. The workload in a test session consists of a variable mix of transactions. Fractions of each transaction type is a test parameter. Other test parameters include the arrival rate which is assumed to be exponentially distributed. The relative deadline of all firm real-time transactions is 100ms. The final deadline is calculated adding the relative deadline to the arrival time.

In the first series of experiments we have used a fixed arrival rate of transactions and varied the fraction of write transactions (W1) from 10% to 100%. The rest are read transactions (R1). In Figure 3 the arrival rate of transactions is 100 transactions per second, in Figure 4 the arrival rate of transactions is 333 transactions per second, in Figure 5 the arrival rate of transactions is 500 transactions per second.

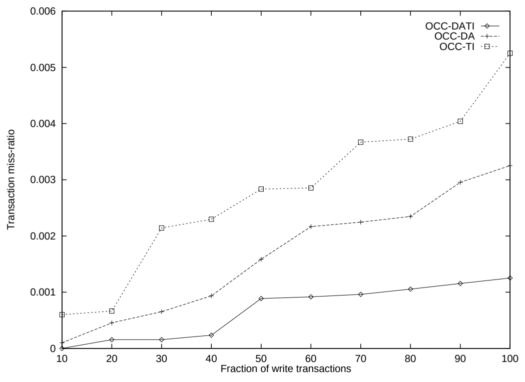

Figure 3. OCC algorithms with 100 tr/s.

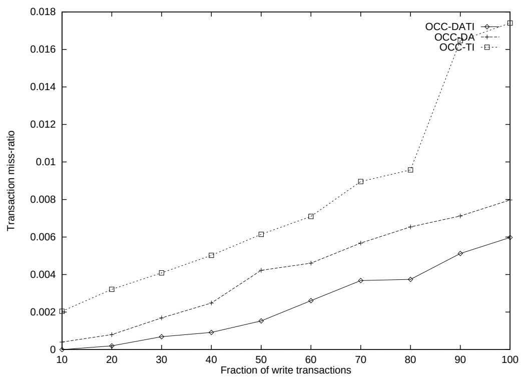

Figure 4. OCC algorithms with 333 tr/s.

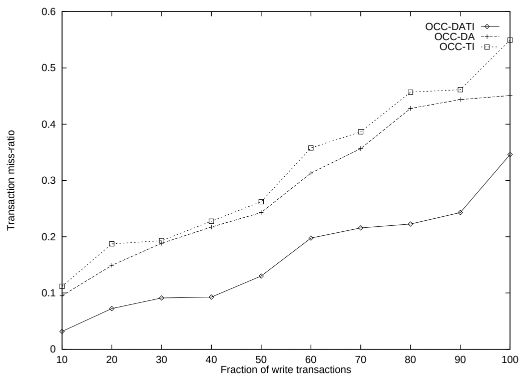

Figure 5. OCC algorithms with 500 tr/s.

OCC-DATI clearly offers the best performance in all tests. This confirms that the overhead for supporting dynamic adjustment in OCC-DATI is smaller than the one in OCC-DA. This also confirms that the number of transaction restarts is smaller in OCC-DATI than OCC-TI or OCC-DA. The results clearly show how unnecessary restarts affect the performance of the OCC-TI. Our results also confirm results in [15] that OCC-DA outperform OCC-TI. The results [17] have already confirmed that OCC-TI outperforms OCC-BC, OPT-WAIT and WAIT-50 algorithms. These results confirm that OCC-DATI also outperforms OCC-DA and OCC-TI when the arrival rate of the transactions is increased.

A high data contention introduced by massive service management transactions often increases the number of concurrency control aborts in traditional optimistic concurrency control protocols. In addition, the validation time of transaction must be bounded in order to guarantee that real-time transactions will meet their deadlines. In the second series of experiments (Figure 6), we examined how the execution of non-real time service management transactions affect the database throughput. The fractions of transactions were: R1 89.8%, W1 10.0%, and T1 0.02%. Experiment results confirm that OCC-DATI outperforms OCC-DA.

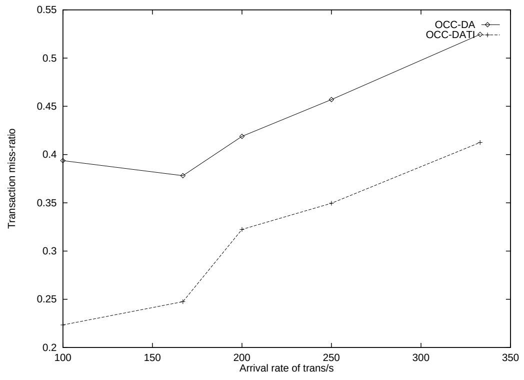

Figure 6. Service management transactions compared.

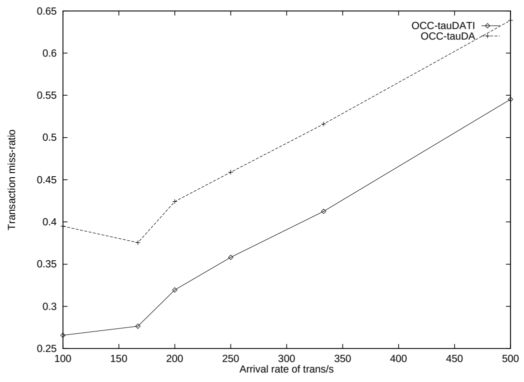

Figure 7. Relaxed serializability compared.

In the third series of experiments (Figure 7) we examined the execution of non-real time service management transactions. We implemented semantic conflict resolution and τ-serializability modifications to both OCC-DA and OCC-DATI algorithms. Modified algorithms are named OCC-τDA and OCC-τDATI respectively. The results show that OCC-τDATI performs better than OCC-τDA.

6 Conclusions

Although the optimistic approach has been shown to be better than locking protocols for RTDBSs, it has the problems of unnecessary restarts and heavy restart overhead. In this paper, we propose a new optimistic concurrency control protocol called OCC-DATI. It has several advantages over the other concurrency control protocols. The protocol maintains all the nice properties with forward validation, a high degree of concurrency, freedom from deadlock, and early detection and resolution of conflicts, resulting in both less wasted resources and a smaller number of restarts. All of these are important to the performance of RTDBSs and contribute to greater chances of meeting transaction deadlines.

An efficient method was designed to adjust the serialization order dynamically amongst the conflicting transactions to reduce the number of transaction restarts. Compared with other OCC protocols that use dynamic serialization order adjustment, our method OCC-DATI is much more efficient and its overhead is smaller. There is no need to check for conflicts while a transaction is still in its read phase. All the checking is performed in the validation phase. As 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 in resolving the conflict. OCC-DATI has a new final timestamp selection method compared to OCC-TI. We have also used the deferred dynamic adjustment of the serialization order, which further minimizes unnecessary restarts. Our performance studies presented here confirm that OCC-DATI outperforms both OCC-TI and OCC-DA.

We have also presented methods to relax the correctness criterion to semantic based serializability using τ-serializability and semantic locking. These methods are used to create OCC-τDATI and OCC-τDA algorithms whose performance was also studied. The results confirmed that OCC-τDATI outperforms OCC-τDA.

Acknowledgments

We would like to thank Pasi Porkka, Matti Luukkainen, Jaakko Kurhila, and Tiina Niklander from the University of Helsinki for fruitful discussion during this research.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Proc. of the*

14th VLDB Conf., pp 1–12, San Mateo, Calif., 1988. Morgan Kaufmann.

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

[3] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, Nov. 1976.

[4] M. H. Graham. Issues in real-time data management. *The Journal of Real-Time Systems*, 4:185–202, 1992.

[5] M. H. Graham. How to get serializability for real-time transactions without having to pay for it. In *Real-time System Symp.*, pp 56–65, 1993.

[6] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[7] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proc. of the 11th Real-Time Symp.*, pp 94–103, Los Alamitos, Calif., 1990. IEEE Computer Society Press.

[8] J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proc. of the 9th ACM Symp. on Principles of Database Systems*, pp 331–343. ACM Press, 1990.

[9] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proc. of the 17th VLDB Conf.*, pp 35–46, San Mateo, Calif., Sept. 1991. Morgan Kaufmann.

[10] J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham. Experimental evaluation of real-time transaction processing. In *Proc. of the 10th Real-Time Systems Symp.*, pp 144–153, Los Alamitos, Calif., Dec. 1989. IEEE Computer Society Press.

[11] J. Kiviniemi, T. Niklander, P. Porkka, and K. Raatikainen. Transaction processing in the RODAIN real-time database system. In A. Bestavros and V. Fay-Wolfe, eds,*Real-Time Database and Information Systems*, pp 355–375, London, 1997. Kluwer Academic Publishers.

[12] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Tr on Database Systems*, 6(2):213–226, Jun 1981.

[13] T. Kuo and A. K. Mok. Application semantics and concurrency control of real-time data-insensive applications. In *Proc. of Real-Time System Symp.*, pp 76–86, 1993.

[14] K.-W. Lam, S. H. Son, and S. Hung. A priority ceiling protocol with dynamic adjustment of serialization order. In *13th IEEE Conf. on Data Engineering (ICDE'97)*, Birmingham, UK, Apr. 1997.

[15] K.-W. Lam, K.-Y. Lam, and S. Hung. An efficient real-time optimistic concurrency control protocol. In *Proc. of the First Int. Workshop on Active and Real-Time Database Systems*, pp 209–225, Skövde, Sweden, Jun 1995. Springer.

[16] K.-W. Lam, K.-Y. Lam, and S. Hung. Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *Proc. of IEEE Real-Time Technology and Application Symp.*, pp 174–179, Chigago, Illinois, May 1995.

[17] J. Lee and S. H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *Proc. of the 14th IEEE Real-Time Systems Symp.*, pp 66–75, Los Alamitos, Calif., 1993. IEEE, IEEE Computer Society Press.

[18] Y. Lee and S. H. Son. P.formance of concurrency control algorithms for real-time database systems. In V. Kumar, ed., *P.formance of Concurrency Control Mechanisms in Centralized Database Systems*, pp 429–460. Prentice-Hall, 1996.

[19] K.-J. Lin. Consistency issues in real-time database systems. In *Proc. of the 22nd Hawaii Int. Conf. on System Sciences*, pp 654–661, 1989.

[20] K.-J. Lin and C. Peng. Enhancing external consistency in real-time transactions. *ACM SIGMOD Record*, 25(1):26–28, Mar. 1996.

[21] J. Lindström, T. Niklander, P. Porkka, and K. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Proc. of the Workshop on Databases in Telecommunications*, Edinburgh, UK., Co-located with VLDB-99, 1999.

[22] D. Menasce and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[23] T. Niklander, J. Kiviniemi, and K. Raatikainen. A real-time database for future telecommunication services. In D. Gaïti, ed., *Intelligent Networks and Intelligence in Networks*, pp 413–430, Paris, France, 1997. Chapman & Hall.

[24] K. Raatikainen. Real-time databases in telecommunications. In A. Bestavros, K.-J. Lin, and S. H. Son, eds, *Real-Time Database Systems: Issues and Applications*, pp 93–98. Kluwer, 1997.

[25] K. Raatikainen, T. Karttunen, O. Martikainen, and J. Taina. Evaluation of database architectures for intelligent networks. In *Proc. of the 7th World Telecommunication Forum (Telecom 95), Technology Summit, Volume 2*, pp 549–553, Geneva, Switzerland, Sept. 1995. ITU.

[26] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. *IEEE Tr on Knowledge and Data Engineering*, 7(6), Dec. 1996.

[27] L. Sha, J. P. Lehoczky, and E. D. Jensen. Modular concurrency control and failure recovery. *IEEE Tr on Computers*, 37(2):146–159, Feb. 1988.

[28] M. Singhal. Issues and approaches to design of real-time database systems. *ACM SIGMOD Record*, 17(1):19–33, Mar. 1988.

[29] J. A. Stankovic and W. Zhao. On real-time transactions. *ACM SIGMOD Record*, 17(1):4–18, Mar. 1988.

[30] J. Taina and K. Raatikainen. Experimental real-time object-oriented database architecture for intelligent networks. *Engineering Intelligent Systems*, 4(3):57–63, Sept. 1996.