

# Extensions to Optimistic Concurrency Control with Time Intervals

Jan Lindström \*

University of Helsinki, Department of Computer Science  
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, FINLAND  
jplindst@cs.helsinki.fi

## Abstract

*Although an optimistic approach has been shown to be better suited than locking protocols for real-time database systems (RTDBS), it has the problems of unnecessary restarts and heavy restart overhead. In this paper we identify the unnecessary restart problem in OCC-TI (Optimistic Concurrency Control with Time Intervals), propose a solution to this problem and demonstrate that the solution will produce a correct result. Additionally, two extensions to the basic dynamic adjustment of the serialization order conflict resolution method used in OCC-TI are proposed. Experiments with a prototype implementation of a real-time database system show that the proposed method clearly outperforms the original OCC-TI.*

## 1 Introduction

A *real-time database system* (RTDBS) is a database system that must process transactions within definite time bounds, usually defined as a deadline. Failure to complete transactions before their deadlines greatly decreases the usefulness of the transactions. Deadlines may be lost due to problems in scheduling or transaction data contention. Considerable research has been devoted to designing concurrency control algorithms for RTDBSs and to evaluating their performance. Most of these algorithms use serializability as correctness criterion and are based on one of the two basic concurrency control mechanisms: 2PL [4, 6, 12, 13, 16, 19] or *optimistic concurrency control* (OCC) [9, 7, 3, 2, 13, 10, 11]. 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 [18].

Optimistic concurrency control [9, 5] protocols have the properties of non-blocking and deadlock-free which make them especially attractive for RTDBS. 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 real-time 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. Since transactions in a real-time database are time-constrained, it is essential that any concurrency control algorithm must minimize waste of resources [19].

In this paper we will identify an unnecessary restart problem in OCC-TI [12] optimistic concurrency control algorithm. We will present a solution to this problem and demonstrate that our solution will produce the correct result. Additionally we propose two extensions to the basic conflict resolution method used in OCC-TI. The remainder of this paper is organized as follows. In Section 2, we review the principles of the original OCC-TI concurrency control protocol. Section 3 proposes extensions to the conflict resolution method used in OCC-TI. A revised OCC-TI algorithm is provided in Section 4. Section 5 presents results of experiments and finally, Section 6 summarizes the main conclusions of the study.

## 2 Optimistic Concurrency Control

### 2.1 OCC-TI

The OCC-TI (Optimistic Concurrency Control with Timestamp Intervals) [12] is one of the optimistic concurrency control protocols proposed for the real-time database systems. In the OCC-TI protocol, every transaction in the read phase

---

\*This work is partially funded by Nokia Telecommunications, Solid Information Technology Ltd., and the National Technology Agency Finland.

is assigned a timestamp interval (TI). At the start of the execution, the timestamp interval of the transaction is initialized as  $[0, \infty[$ , i.e., the entire range of the timestamp space. This timestamp interval is used to record a temporary serialization order during the execution of the transaction. Whenever the serialization order of the transaction is altered by its data accessing operation or the validation of other transactions, its timestamp interval is adjusted to represent the dependencies. This conflict resolution method is called *dynamic adjustment of the serialization order*.

When a read operation is executed in the read phase, the write timestamp (*WTS*) of the accessed object ( $D_i$ ) is verified against the time interval allocated to the transaction ( $T_i$ ). When a write operation is executed in the read phase, a *pre-write* operation is used to verify the read timestamp (*RTS*) and write timestamps of the written object against the time interval allocated to the transaction. If another transaction has read or written the object outside the time interval, the transaction must be restarted.

At the beginning of the validation phase, the final timestamp ( $TS(T_v)$ ) of the validated transaction is determined from the timestamp interval allocated to the transaction. In this algorithm, the minimum value of  $TI(T_v)$  is selected as the timestamp  $TS(T_v)$  for the validating transaction [12]. The adjustment of timestamp intervals iterates through the read set (RS) and write set (WS) of the validating transaction. The protocol iterates 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 time interval of the active transaction is adjusted. 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.

## 2.2 Unnecessary restarts

A major performance problem with OCC protocols are the heavy restart overheads, wasting a large amount of resources. This is because conflict-checking is done in the validation phase. If the transaction has read or updated many objects and the transaction has to be aborted, all the changes to the database must be rolled back and the transaction restarted. Thus the transaction re-executes all read and write operations. In many cases, this will result in missed deadlines and wasted resources. Forward Validation (OCC-FV) [5] is based on the assumption that the serialization order of transactions is determined by the arrival order of the transactions

at the validation phase. A validation process based on this assumption can incur restarts that are not necessary to ensure data consistency. These restarts should be avoided.

This same major problem can be found in the original OCC-TI. The problem with the existing algorithm is best described by the example given below.

**EXAMPLE 2.1** *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 timestamp interval of the transaction to be forward adjusted to  $[100, \infty[$ .  $T_2$  then executes a read operation on the same object, which causes the timestamp interval of the transaction to be forward adjusted similarly, e.g. to  $[100, \infty[$ .  $T_1$  then executes  $w_1[x]$ , which causes the timestamp interval of the transaction to be forward adjusted to  $[100, \infty[$ .  $T_1$  starts the validation, and the final (commit) timestamp is selected to be  $TS(T_1) = \min([100, \infty]) = 100$ . Because we have one write-read conflict between the validating transaction  $T_1$  and the active transaction  $T_2$ , the timestamp interval of the active transaction must be adjusted: Thus  $TI(T_2) = [100, \infty[ \cap [0, 99] = []$ . The timestamp 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  $\square$ .*

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. Hence, unnecessary restart problems found in OCC-TI using a very simple history is not desirable. Therefore we will propose a solution to this problem in section 4.

## 3 Extensions to OCC-TI

An earlier unnecessary restart problem was detected from OCC-TI protocol. Secondly, there is no real-time properties in the OCC-TI protocol. In this section we propose an extension to the OCC-TI protocol to solve these problems. This paper includes the following extensions to OCC-TI:

- 1) Reversible Dynamic Adjustment of Serialization Order

## 2) Prioritized Dynamic Adjustment of Serialization Order

In the first extension we try to undo dynamic adjustments done to an active transaction when the adjustment was unnecessary. For example, if the validating transaction aborts then all dynamic adjustment to other conflicting active transactions were unnecessary. In the second extension we take into account priorities before using dynamic adjustment.

### 3.1 Reversible Dynamic Adjustment

Let  $TI(T_i)$  denote the timestamp interval for transaction  $T_i$  and let  $RTI_n(T_i)$ ,  $n = 1, ..k$ ,  $k \in \mathbb{N}$  denote the  $n$ :th removed timestamp interval from transaction  $T_i$ . One modification to timestamp interval can be reversed if the current timestamp interval and the removed timestamp interval are continuous. Formally,

**DEFINITION 3.1** *The timestamp interval  $TI(T_i)$  of transaction  $T_i$  is **reversible** with removed timestamp interval  $RTI_n(T_i)$ ,  $n = k, .., 1$ ,  $k \in \mathbb{N}$  if and only if:*

$$\begin{aligned} & \forall x \forall y ((x \in TI(T_i) \wedge y \in RTI_n(T_i)) \wedge \\ & \nexists z (z \in ([0, \infty[ \setminus (TI(T_i) \cup RTI_n(T_i)) \wedge \\ & (x < z < y) \vee (y < z < x))). \end{aligned}$$

**EXAMPLE 3.1** *Let  $TI(T_1) = [100, 1000]$ ,  $RTI_1(T_1) = [0, 100]$ , and  $RTI_2(T_1) = [1002, 2000]$ . Using definition 3.1, the first removed timestamp interval to be checked for reversing is  $RTI_2(T_1)$ . If we set  $x = 1000$  and  $y = 1002$  then clearly  $\exists z (z \in [0, \infty[ \wedge (1000 < z < 1002))$  e.g.  $z = 1001$ . Thus the removed timestamp interval  $RTI_2(T_1)$  cannot be reversed. When checking  $RTI_1(T_1)$  for reversing we can see that definition 3.1 holds and we can rollback the removed timestamp interval  $\square$ .*

Next the definition 3.2 we shows how dynamically adjusted timestamp intervals can be reversible.

**DEFINITION 3.2** *The timestamp interval  $TI(T_i)$  of transaction  $T_i$  is **reversed** with removed timestamp interval  $RTI_n(T_i)$ ,  $n \in \mathbb{N}$  calculating the new timestamp interval*

$$TI(T_i) = TI(T_i) \cup RTI_n(T_i).$$

**EXAMPLE 3.2** *Let  $TI(T_1) = [100, 1000]$  and  $RTI_1(T_1) = [0, 100]$ . Then rollbacking is done with*

$$TI(T_1) = [100, 1000] \cup [0, 100] = [0, 1000] \square.$$

This method, while important, needs additional data structure to store removed timestamp intervals and in case of rollbacking quite expensive iteration of data structure holding removed timestamp intervals. We develop a far better method in section 4. If the final timestamp of the validating transaction is selected carefully in the validation phase, there is no need for rollbacking. Another used method is to store dynamic adjustments of the timestamp intervals to local variables and update timestamp intervals of a conflicting transaction when validating transaction is guaranteed to commit.

### 3.2 Prioritized Dynamic Adjustment

In this section a priority-dependent extension to dynamic adjustment of the serialization order is presented. In real-time database systems, the conflict resolution should take into account the priority of the transactions. This is especially true in the case of heterogeneous transactions. Some transactions are more important or valuable than others. The dynamic adjustment should be done in favor of a higher priority transaction. Here, we present a method, where we try to make more room for the higher priority transaction to commit in its timestamp interval. This offers the high priority transaction better chances to commit before its deadline and meeting timing constraints.

The Prioritized Dynamic Adjustment of the Serialization Order (PDASO) implemented with timestamp intervals creates a partial order between transactions based on conflicts and priorities. Suppose we have a validating transaction  $T_v$  and an active transaction  $T_j$  ( $j \in \mathbb{N}$ ). Let  $TS(T_v)$  be the final timestamp of the validating transaction  $T_v$  and  $TI(T_j)$  the timestamp interval of the active transaction  $T_j$ . Let  $TI(T_v)$  be the timestamp interval of the validating transaction and  $pri(T_i)$  be the priority of transaction  $T_i$ . There are three possible types of data conflicts which are resolved using PDASO between  $T_v$  and  $T_j$ :

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

A read-write conflict between  $T_j$  and  $T_v$  can be resolved by adjusting the timestamp interval of the active transaction **forward**, e.g.  $T_v \rightarrow T_j$ . If the validating transaction has higher priority than the active conflicting transaction, forward adjustment is correct. If the validating transaction has lower priority than the active conflicting transaction, we should favor the higher priority transaction. This is supported by reducing the timestamp interval of the validating transaction and selecting a new final timestamp earlier in timestamp interval. Normally the current time or maximum

value from the timestamp interval is selected. But now the middle point is selected. This offers greater changes for the higher priority transaction to commit in its timestamp interval. If the middle point cannot be selected, the validating is restarted. This is wasted execution, but it is required to ensure the execution of the transaction of higher priority.

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

A write-read conflict between  $T_j$  and  $T_v$  can be resolved by adjusting the serialization order between  $T_v$  and  $T_j$  by adjusting the timestamp interval of the active transaction **backward**, e.g.  $T_j \rightarrow T_v$ . If the validating transaction has higher priority than the active conflicting transaction, backward adjustment is correct. If the validating transaction has lower priority than the active conflicting transaction, then backward adjustment is done if the active transaction is not aborted in backward adjustment. Otherwise, the validating transaction is restarted. This is wasted execution, but it is required to ensure the execution of the transaction of higher priority.

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

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

Thus, in backward adjustment, we cannot move the validating transaction to the future to obtain more space for the higher priority transaction. We can only check if the timestamp interval of the higher priority transaction would become empty. In forward ordering we can move the final timestamp backward if there is space in the timestamp interval of the validating transaction. Again, we check if the timestamp interval of the higher priority transaction would shut out. We have chosen to abort the validating transaction when the timestamp interval of the higher priority transaction shuts out. Thus, this algorithm favors the higher priority transactions and might waste resources aborting near to complete transactions.

**EXAMPLE 3.3** Let  $TI(T_1) = [100, 1000]$ ,  $TS(T_1) = 1000$ , and  $TI(T_2) = [0, \infty[$ . Let  $pri(T_1) < pri(T_2)$ . Assume that we have a read-write conflict between transactions. We first make room for the active transaction  $T_2$  and then move the active transaction forward:

$$TS(T_1) = (100 + 1000)/2 = 550$$

$$TI(T_2) = [0, \infty[ \cap [550, \infty[ = [550, \infty[ \square$$

## 4 Revised OCC-TI Algorithm

In this section a validating algorithm for extended OCC-TI is presented. OCC-TI is extended with a new final timestamp

selection method and priority-depended conflict resolution. We should select the final (commit) timestamp  $TS(T_v)$  in such a way that room is left for backward adjustment. We propose a new validation algorithm where the commit timestamp is selected differently. In our revised validation algorithm for OCC-TI (Figure 1) we set  $TS(T_v)$  as the validation time if it belongs to the time interval of  $T_v$  or the maximum value from the time interval otherwise. Additionally, the original OCC-TI is extended to use prioritized dynamic adjustment of the serialization order. 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 intervals 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.

The adjustment of timestamp intervals ( $TI$ ) iterates through the read set (RS) and write set (WS) of the validating transaction ( $T_v$ ). First we check that the validating transaction has read from committed transactions. This is done by checking the object's read timestamp ( $RTS$ ) and write timestamp ( $WTS$ ). These values are fetched when the read/write operation to the current object is made. Then the algorithm iterates 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. 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 timestamps and write timestamps of accessed objects are updated and changes to the database are committed. Figure 2 presents forward and backward adjustment algorithms for dynamic adjustment of the serialization order using timestamp intervals with deferred dynamic adjustment and priorities.

Backward and Forward adjustment algorithms creates order between conflicting transaction timestamp intervals. A final (commit) timestamp is selected from the remaining timestamp interval of the validating transaction. Therefore the final timestamps of the transactions create partial order between transactions.

Having described the basic concepts and the protocol, we now prove the correctness of the protocol. To prove a history  $H$  produced by revised OCC-TI is serializable, we only

```

occti_validate( $T_v$ ) {
  /* Select final (commit) timestamp */
  if (validation_time  $\in$   $TI(T_v)$ )
     $TS(T_v) = validation\_time$ ;
  else  $TS(T_v) = max(TI(T_v))$ ;

  /* Iterate read/writeset of the validating transaction */
  for  $\forall D_i \in (RS(T_v) \cup WS(T_v))$ {

    /* Iterate conflicting active transactions */
    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  $WS(T_v)$  to database;
}

```

**Figure 1. Revised validation algorithm for the OCC-TI.**

have to prove that the serialization graph for  $H$ , denoted by  $SG(H)$ , is acyclic [1].

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

*Proof:* Since 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 implies 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-TI adjusts  $TI(T_2)$  to follow  $RTS(x)$  that is equal to or greater than

```

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

  if ( $priority(T_v) \geq priority(T_a)$ )
     $TI = TI \cap [TS(T_v), \infty]$ ;
  else {
    if ( $((min(TI(T_v)) + TS(T_v)) / 2) \in TI(T_v)$ ) {
       $TS(T_v) = (min(TI(T_v)) + TS(T_v)) / 2$ ;

      if ( $TS(T_v) > max(TI)$ ) restart( $T_v$ );

       $TI = TI \cap [TS(T_v), \infty]$ ;
    }
    else {
      if ( $TS(T_v) > max(TI)$ ) restart( $T_v$ );

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

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

  if ( $priority(T_v) \geq priority(T_a)$ )
     $TI = TI \cap [0, TS(T_v) - 1]$ ;
  else {
    if ( $TS(T_v) - 1 < min(TI)$ ) restart( $T_v$ );

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

```

**Figure 2. Backward and Forward adjustment.**

$TS(T_1)$ . That is,  $TS(T_1) \leq RTS(x) < TS(T_2)$ . Therefore,  $TS(T_1) < TS(T_2)$ .

- 2)  $w_1[x] \rightarrow r_2[x]$ : This case implies that the write phase of  $T_1$  finishes before  $r_2[x]$  is executed in  $T_2$ 's read phase. For  $r_2[x]$ , OCC-TI adjusts  $TI(T_2)$  to follow  $WTS(x)$ , which is equal to or greater than  $TS(T_1)$ . That is,  $TS(T_1) \leq WTS(x) < TS(T_2)$ . Therefore,  $TS(T_1) < TS(T_2)$ .
- 3)  $w_1[x] \rightarrow w_2[x]$ : This case implies that the write phase of  $T_1$  finishes before  $w_2[x]$  is executed in  $T_2$ 's write phase. For  $w_2[x]$ , OCC-TI adjusts  $TI(T_2)$  to follow  $WTS(x)$ , which is equal to or greater than  $TS(T_1)$ . That is,  $TS(T_1) \leq WTS(x) < TS(T_2)$ . Therefore,  $TS(T_1) < TS(T_2)$ .  $\square$

**Theorem 1:** Every history generated by the revised OCC-TI

algorithm is serializable.

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

We present here the same example history as in example 2.1, which caused an unnecessary restart in the original OCC-TI. Using the same example, we show here how the revised OCC-TI produces a serializable history and avoids unnecessary restarting.

**EXAMPLE 4.1** *Let  $RTS(x)$  and  $WTS(x)$  be initialized as 100. Consider transactions  $T_1$ ,  $T_2$ , and history  $H_1$ , where  $pri(T_1) = pri(T_2)$ :*

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

*In a similar way as in Example 2.1, transactions  $T_1$  and  $T_2$  are forward adjusted to  $[100, \infty[$ . Transaction  $T_1$  starts the validation at time 1000, and the final (commit) timestamp is selected to be  $TS(T_1) = validation\_time = 1000$ . Because we have one write-read conflict between the validating transaction  $T_1$  and the active transaction  $T_2$ , the timestamp interval of the active transaction must be adjusted:  $TI(T_2) = [100, \infty[ \cap [0, 999] = [100, 999]$ . Thus the timestamp interval is not empty, and we have avoided unnecessary restart. Both transactions commit successfully. History  $H_1$  is acyclic, that is, serializable. Therefore the proposed revised OCC-TI produces serializable histories and avoids the unnecessary restart problem found in the original OCC-TI algorithm  $\square$ .*

## 5 Results of Experiments

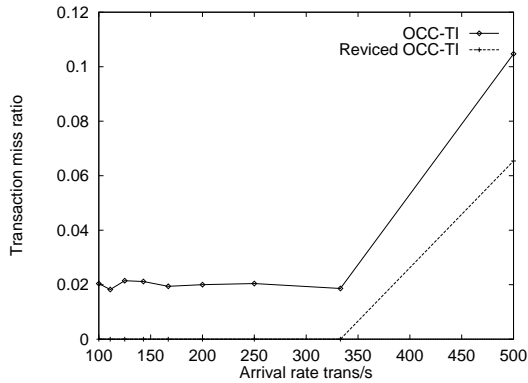
We have carried out a set of experiments in order to examine the feasibility of our algorithm in practice. *Real-Time Object-Oriented Database Architecture for Intelligent Networks* (RODAIN) [8, 14, 17] is an architecture for a real-time, object-oriented, fault-tolerant, and distributed database management system. 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 Pentium Pro 200MHz with 64 MB of main memory with the Chorus/ClassiX3.1

real-time operating system. The test environment is a reduced subset of the RODAIN architecture.

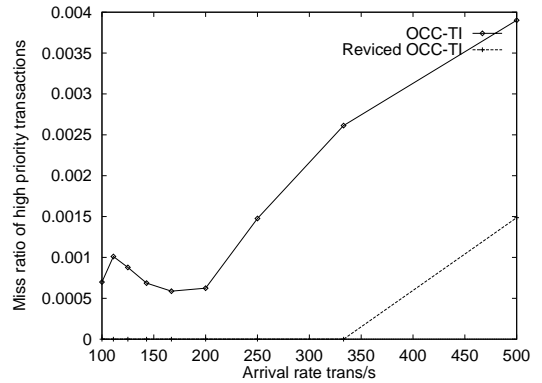
Every test session contains 10 000 transactions and is repeated at least 20 times. The reported values are means of the replications. The test database represents a typical Intelligent Network (IN) service. The size of the database is 30 000 objects. We used two different transactions named R1 and W1. Transaction R1 is a read-only service provision transaction that reads a few objects and commits. Transaction W1 is an update service provision transaction that reads a few objects, updates them and commits. New transactions are accepted up to a respecified limit, which is the number of installed processes. In the experiments the limit was 100. If there are no processes available when a new transaction arrives, the transaction 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 assumed to be exponentially distributed.

The experiment was run to compare the miss rates of the original OCC-TI and our revised OCC-TI. In the experiments, the arrival rate of the transactions is varied from 100 to 500 transactions per second. In Figure 3(a)–(b) the fraction of write transactions is varied from 20% to 30%. In Figure 3 shows that the revised OCC-TI performs better than OCC-TI, especially when the arrival rate is high. This is because the revised OCC-TI does not suffer from the unnecessary restart problem. Figures 4(a)–(b) show the miss ratio of transactions of high priority. This demonstrates how the Revised OCC-TI favors transactions of high priority. Revised OCC-TI clearly offers better chances for high priority transactions to complete according to their deadlines. The results clearly indicate that Revised OCC-TI meets the goal of favoring transactions of high priority.

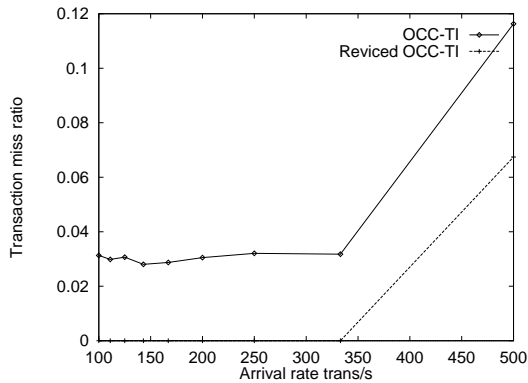
In the final experiments, we have included results from OCC-DA and OCC-DATI protocols. The arrival rate of the transactions is varied from 100 to 500 transactions per second. In Figure 5(a) the fraction of write transactions is 20%. Performance of the revised OCC-TI is similar to OCC-DA (Optimistic Concurrency Control with Dynamic Adjustment) [10, 11] and OCC-DATI (Optimistic Concurrency Control with Dynamic Adjustment of the Serialization Order using Timestamp Intervals) [15].



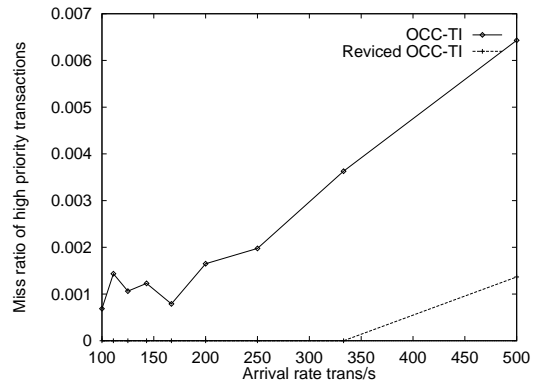
(a) write fraction 20%



(a) write fraction 20%



(b) write fraction 30%



(b) write fraction 30%

**Figure 3. OCC-TI and Revised OCC-TI compared.**

**Figure 4. Comparison with critical transactions.**

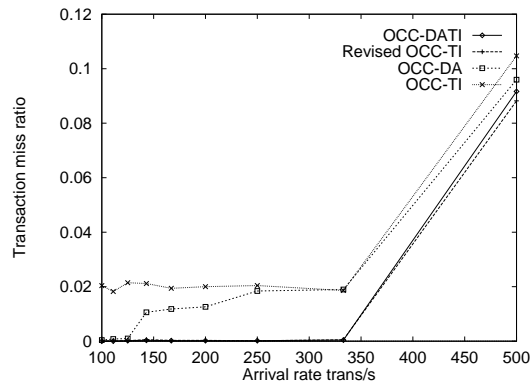
## 6 Conclusion

We have provided a simple solution to this problem by changing the way final timestamps are chosen and demonstrated that our solution will produce a correct result. Additionally, we proposed two extensions to the basic conflict resolution method used in OCC-TI. Extended OCC-TI includes a new final timestamp selection method and priority-dependent conflict resolution. We have demonstrated that the revised OCC-TI produces a correct result. Our results from experiments showed that the revised OCC-TI outperforms original OCC-TI. Additionally, the revised OCC-TI clearly offers better chances for high priority transactions to complete according to their deadlines. The results clearly

indicate that the revised OCC-TI meets the goal of favoring transactions of high priority. Performance of the revised OCC-TI is comparable even with optimistic concurrency control protocols OCC-DA and OCC-DATI. The most important feature of the revised OCC-TI is that it clearly offers better chances for the high priority transactions to complete before their deadlines when compared to the original OCC-TI. Therefore the revised OCC-TI is a promising candidate for firm real-time database systems where transactions are heterogeneous.

## Acknowledgments

I would like to thank Tiina Niklander, Lea Kutvonen, Jaakko Kurhila, Matti Luukkainen and Kimmo Raatikainen from the



(a) write fraction 20%

**Figure 5. OCC protocols compared.**

University of Helsinki for fruitful discussion during this research.

## References

- [1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [2] A. Datta and S. H. Son. A study of concurrency control in real-time active database systems. Tech. report, Department of MIS, University of Arizona, Tucson, 1996.
- [3] A. Datta, I. R. Viguier, S. H. Son, and V. Kumar. 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*, 1997.
- [4] 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, November 1976.
- [5] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.
- [6] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proc. of the 11th Real-Time Systems Symposium*, pages 94–103.
- [7] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of the 17th VLDB Conference*, pages 35–46, 1991.
- [8] 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, editors, *Real-Time Database and Information Systems*, pages 355–375, 1997.
- [9] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.
- [10] K.-W. Lam, K.-Y. Lam, and S. Hung. 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, 1995.
- [11] K.-W. Lam, K.-Y. Lam, and S. Hung. 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, 1995.
- [12] J. Lee and S. H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 66–75, 1993.
- [13] J. Lee and S. H. Son. Performance of concurrency control algorithms for real-time database systems. In V. Kumar, editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, pages 429–460, 1996.
- [14] J. Lindström, T. Niklander, P. Porkka, and K. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Databases in Telecommunications*, LNCS 1819, pages 158–173, 1999.
- [15] J. Lindström and K. Raatikainen. 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*, 1999.
- [16] D. Menasce and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.
- [17] T. Niklander, J. Kiviniemi, and K. Raatikainen. A real-time database for future telecommunication services. In D. Ga'ni, editor, *Intelligent Networks and Intelligence in Networks*, pages 413–430, 1997.
- [18] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1:199–226, 1993.
- [19] P. S. Yu, K.-L. Wu, K.-J. Lin, and S. H. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.