

# Using Optimistic Concurrency Control in Firm Real-Time Databases

Jan Lindström

University of Helsinki, Department of Computer Science  
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland  
{jan.lindstrom}@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 optimistic concurrency control protocols called Revised OCC-TI, OCC-DATI, and OCC-PDATI. 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. The protocols 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.*

## 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. 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. Given these challenges, considerable research has recently been devoted to designing concurrency control algorithms for RTDBSs and to evaluating their performance [2, 6–8, 12, 10] Most of these algorithms are based on one of the two basic concurrency control mechanisms: *locking* [3] or *optimistic concurrency control* (OCC) [9].

In *Optimistic Concurrency Control* (OCC) [9], 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*. 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.

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 ongoing research on optimistic methods. We present revised OCC-TI, OCC-DATI and OCC-PDATI protocols. In this paper we concentrate on firm real-time transaction models. OCC-DATI and OCC-PDATI are a fully optimistic protocols and they use forward adjustment. With the new protocol, the number of transaction restarts is smaller than with OCC-BC, OPT-WAIT, or WAIT-50 [5–7], 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 [10, 11].

The rest of the paper is organized as follows. Section 2 introduces the revised OCC-TI. Section 3 presents the OCC-DATI algorithm. In Section 4 we introduce a way to use importance of the transaction in conflict resolution and OCC-PDATI protocol. Finally, the conclusion of the paper is presented in Section 5.

## 2 Revised OCC-TI Algorithm

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  $T_i$ .

The OCC-TI [14, 13] protocol resolves conflicts using time intervals of the transactions. Every transaction must be executed within a specific time slot. When an access conflict occurs, it is resolved using the read and write sets of the transaction together with the allocated time slot. Time slots are adjusted when a transaction commits.

In this protocol, every transaction in the read phase is assigned a timestamp interval (TI). This interval is used to record a temporary serialization order induced during the execution of the transaction. At the start of the execution, the timestamp interval of the transaction is initialized as  $[0, \infty[$ , i.e., the entire range of timestamp space. Whenever the serialization order of the transaction is induced by its data operation or the validation of other transactions, its timestamp interval is adjusted to represent the dependencies.

In the read phase when a read operation is executed, the write timestamp (WTS) of the object accessed is verified against the time interval allocated to the transaction. If another transaction has written the object outside the time interval, the transaction must be restarted. In the read algorithm  $D_i$  is the object to be read,  $T_i$  is the transaction reading the object,  $TI(T_i)$  is the time interval allocated to the transaction,  $WTS(D_i)$  is the write timestamp of the object, and  $RTS(D_i)$  is the read timestamp of the object.

In the read phase when a write operation is executed, the modification is made to a local copy of the object. A *pre-write* operation is used to verify read 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.

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  $H_1$ :

$$\begin{aligned} T_1 &: r_1[x] \ w_1[x] \ v_1 \ c_1 \\ T_2 &: r_2[x] \ v_2 \ c_2 \\ H_1 &: r_1[x] \ r_2[x] \ w_1[x] \ v_1 \ . \end{aligned}$$

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.

We should select the commit timestamp  $TS(T_v)$  in such a way that room is left for backward adjustment. We propose a new validation algorithm where 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 belong to the time interval of  $T_v$  or maximum value from the time interval otherwise.

```

occti_validate( $T_v$ )
{
  if ( $validation\_time \in TI(T_v)$ )  $TS(T_v) = validation\_time$ ;
  else  $TS(T_v) = max(TI(T_v))$ ;

  for  $\forall D_i \in (RS(T_v) \cup WS(T_v))$ 
  {
    for  $\forall T_a \in active\_conflicting\_transactions()$ 
    {
      if ( $D_i \in (WS(T_a) \cap RS(T_v))$ )
         $TI(T_a) = TI(T_a) \cap [TS(T_v), \infty[$  ;
      if ( $D_i \in (RS(T_a) \cap WS(T_v))$ )
         $TI(T_a) = TI(T_a) \cap [0, TS(T_v) - 1]$ ;
      if ( $D_i \in (WS(T_a) \cap WS(T_v))$ )
         $TI(T_a) = TI(T_a) \cap [TS(T_v), \infty[$  ;
      if  $TI(T_a) = []$  restart( $T_a$ );
    }
    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;
}

```

Fig. 1. Revised validation algorithm for the OCC-TI.

### 3 OCC-DATI

In this section we present an optimistic concurrency control protocol named OCC-DATI [16]. OCC-DATI is based on forward validation [4]. 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 [15]. 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.

OCC-DA [10, 11] 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 [14] 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 2), 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
   $TS(T_v) = \min(\text{validation\_time}, \max(TI(T_v)))$ ;
  // 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;
}

```

**Fig. 2.** 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 3 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
     $TI = TI(T_a)$ ;

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

```

**Fig. 3.** Backward and Forward adjustment.

## 4 OCC-PDATI

In real-time database systems, the conflict resolution of the transactions should be based on the importance of the transactions. Therefore we have developed an optimistic concurrency control protocol where the decision about which transaction is to be restarted is based on the importance of the transaction. In this section we outline an importance based optimistic concurrency control protocol called OCC-PDATI. OCC-PDATI is based on forward validation [4] and the earlier optimistic OCC-DATI protocol developed in the RODAIN project [16]. The validation protocol is the same in OCC-DATI and OCC-PDATI. The difference is in the conflict resolution. The conflict resolution of OCC-PDATI takes into account the importance of the transaction.

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.

$$\begin{aligned}
 & TI = TI(T_j) \cap [TS(T_v) + 1, \infty[; \\
 & \quad \mathbf{if} ( \text{imp}(T_v) < \text{imp}(T_j) ) \\
 & \quad \quad \mathbf{if} ( TI == \emptyset ) \\
 & \quad \quad \quad \text{restart}(T_v) \\
 & TI(T_j) = TI
 \end{aligned}$$

The order of the transactions should be based on their importance. A transaction of high importance should be executed before a transaction of low importance. However, this is not possible in history if we want to maintain serializability. Therefore we must maintain a serializable order of the transactions. In the usual case of omitting the importance, we can make forward ordering of the active transaction, because all operations become permanent only after the validation phase. On the other hand, a transaction of low importance should not abort a transaction of high importance. If we have transactions of higher importance, we should guarantee their execution before their deadline. Since the transaction scheduling algorithms do not usually take into account possible restarts, the principle above should imply the requirements that a transaction of high importance should not be restarted because of a data conflict with a transaction of low importance. This is a wasted execution, but it is required to ensure execution of transaction of high importance.

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.

$$\begin{aligned}
 TI &= TI(T_j) \cap [0, TS(T_v) - 1]; \\
 &\text{if } ( \text{imp}(T_v) < \text{imp}(T_j) ) \\
 &\quad \text{restart}(T_v) \\
 TI(T_j) &= TI
 \end{aligned}$$

Again, we must ensure serializable (or another correct order of) execution. We do not know what an active transaction is going to do in the future. These future reads or writes may lead to an empty timestamp interval if we make backward adjustment. Therefore transactions of high importance should not be backward adjusted, but 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. We could make backward adjustment of a transaction of high importance if the transaction is not to be restarted due to an empty timestamp. This, however, implies that a transaction of high importance should not be allowed to read or write new data objects that belong to a new database state after a backward adjustment. In the other words, future read or write operations should reduce the timestamp interval of the transaction.

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.

$$\begin{aligned}
 TI &= TI(T_j) \cap [TS(T_v) + 1, \infty]; \\
 &\text{if } ( \text{imp}(T_v) < \text{imp}(T_j) ) \\
 &\quad \text{if } ( TI == \emptyset ) \\
 &\quad \quad \text{restart}(T_v) \\
 TI(T_j) &= TI
 \end{aligned}$$

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

Figure 4 depicts an implementation outline of a deferred 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 ( $T_a \in adjusted$ )
     $TI = adjusted.pop(T_a)$ ;
  else
     $TI = TI(T_a)$ ;

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

  if (  $imp(T_v) < imp(T_a)$  )
    if (  $TI == \emptyset$  )
      restart( $T_v$ ); /* Validation ends here */

   $adjusted.push(\{T_a, TI\})$ ;
}

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

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

  if (  $imp(T_v) < imp(T_a)$  )
    restart( $T_v$ ); /* Validation ends here */

   $adjusted.push(\{T_a, TI\})$ ;
}

```

**Fig. 4.** Backward and Forward adjustment for OCC-PDATI.

## 5 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 have presented optimistic concurrency control protocols Revised OCC-TI, OCC-DATI and OCC-PDATI. The protocols 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. The method also incorporates importance of the transaction in conflict resolution. OCC-TI [14] uses a novel approach to minimize the number of transaction restarts and wasted resources, thus providing better performance relative to pure OCC and OCC-FV. It eliminates transaction restarts for some category of conflicts. This makes OCC-TI a promising candidate for RTDBs by incorporating time sensitive conflict resolution. While the motivation behind OCC-TI is extremely important, the algorithms provided in [14] suffer from a serious problem in that unnecessary restarts are caused. We provided a simple solution to this problem by changing the way how validation timestamp are chosen. Our experimental results presented here and other results showed that revised OCC-TI is clearly better than original OCC-TI.

We have also presented two proposed optimistic protocols OCC-DATI and OCC-PDATI. When compared with the OCC-DATI protocol that uses dynamic serialization order adjustment, the OCC-PDATI protocol offers the same efficiency and the overhead is only slightly larger. Experiments show that OCC-PDATI gives the same performance as OCC-DATI which was found superior to other well known OCC protocols in [16]. The most important feature of the OCC-PDATI is that it clearly offers better chances for the transactions of high importance to complete before their deadlines when compared to the OCC-DATI. The results clearly indicate that OCC-PDATI meets the goal of favoring transactions of high importance.

## Acknowledgments

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

## References

1. R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. *ACM SIGMOD Record*, 17(1):71–81, March 1988.
2. R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In F. Bancilhon and D. J. DeWitt, editors, *Proceedings of the 14th VLDB Conference*, pages 1–12, San Mateo, Calif., 1988. Morgan Kaufmann.
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, November 1976.
4. T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.
5. J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proc. of the 11th Real-Time Symposium*, pages 94–103, Los Alamitos, Calif., 1990. IEEE, IEEE Computer Society Press.
6. J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 331–343. ACM, ACM Press, 1990.

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, San Mateo, Calif., September 1991. Morgan Kaufmann.
8. J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of the 10th Real-Time Systems Symposium*, pages 144–153, Los Alamitos, Calif., December 1989. IEEE, IEEE Computer Society Press.
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, Skövde, Sweden, June 1995. Springer.
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, Chigago, Illinois, May 1995.
12. 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, April 1997.
13. J. Lee. *Concurrency Control Algorithms for Real-Time Database Systems*. PhD thesis, Faculty of the School of Engineering and Applied Science, University of Virginia, January 1994.
14. 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, Los Alamitos, Calif., 1993. IEEE, IEEE Computer Society Press.
15. 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. Prentice-Hall, 1996.
16. 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.