Integrated and Adaptive Optimistic Concurrency Control Method for 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
FAX: +358 9 19144441

Abstract

Real-time database systems must meet time constraints in addition to the integrity constraints. Researchers have speculated that priority cognizant optimistic concurrency control (OCC) methods, if designed well, could outperform priority insensitive ones in real-time database systems. This paper describes an efficient integrated and adaptive concurrency control method for real-time database systems. The proposed method provides both serializability and real-time properties for the transactions. The proposed method is demonstrated to produce serializable histories and the method is tested in practice. The proposed method clearly offers better chances for critical transactions to complete according to their time constraints. The results clearly indicate that the proposed method meets the goal of favoring critical transactions.

1 Introduction

Despite the fact that implementations in most commercial database management systems uses locking for concurrency control, optimistic concurrency control has recently gained attention for its efficiency in new types of data-intensive applications, for example, telephone switching systems, network management, navigation systems, stock trading, and command and control systems.

Traditional databases, hereafter referred to as databases, deal with persistent data. Transactions access this data while maintaining its consistency. The goal of transaction and query processing in databases is to get a good throughput or response time. In contrast, *real-time systems* can also deal with temporal data, i.e., data that becomes outdated after a certain time. Due to the temporal character of the data and the response-time requirements forced by the environment, tasks in real-time systems have time constraints, e.g. periods or deadlines. The important difference is that the goal of real-time systems is to meet the time constraints of the tasks [29].

*This work funded by Nokia OYj Foundation.

One of the most important points to remember here is that real-time does not just mean fast [29]. Additionally real-time does not mean timing constraints that are in nanoseconds or microseconds. Real-time means the need to manage *explicit* time constraints in a predictable fashion, that is, to use time-cognizant methods to deal with deadlines or periodicity constraints associated with tasks.

Concurrency control is one of the main issues in the studies of real-time database systems. With a strict consistency requirement defined by serializability [2], most real-time concurrency control schemes considered in the literature are based on two-phase locking (2PL) [7]. In recent years, various real-time concurrency control methods have been proposed for the single-site RTDBS by modifying 2PL (e.g. [18, 24, 28]). 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 [27].

Optimistic concurrency control methods [8, 15] are especially attractive for real-time database systems because they are non-blocking and deadlock-free. Therefore, in recent years, numerous optimistic concurrency control methods have been proposed for real-time databases (e.g. [16, 20, 21]). Although optimistic approaches have been shown to be better than locking methods for real-time database systems [9, 10], they have the problem of unnecessary restarts and heavy restart overhead. This is due to the late conflict detection that increases the restart overhead since some near-to-complete transactions have to be restarted. Because conflict resolution between the transactions is delayed until a transaction is near its completion, there will be more information available in making the conflict resolution.

Priority-cognizant concurrency control methods based on the optimistic methods have not been widely studied. Exceptions are OCC-APR [4, 6] and OCC-TI [5], where

OCC-TI was extended with priority cognizance. These studies concluded that priority cognizance is not a feasible approach for improving the performance of real-time concurrency control methods beyond the current state of the art.

However, these proposals used priority in the conflict resolution, which might not be a very good choice in dynamic systems, because transactions with very short deadlines might not be very critical and vice versa. Therefore, we propose that criticality of the transaction should be used instead of priority in the conflict resolution. Because time cognizance is important to offer better support for timing constraints as well as predictability, the major concern in designing real-time optimistic concurrency control methods is not only to incorporate information about the criticality of transactions for conflict resolution but also to design methods that minimize the number of transactions to be restarted.

This paper presents a method based on integrated concurrency control [2, 28] ,which extents the optimistic concurrency control method with transaction attributes. The basic ideas of three proposed optimistic concurrency control methods are presented and showed how they are used in an integrated concurrency controller. The rest of the paper is organized as follows. Section 2 presents recent related work in telecommunication and real-time databases. Section 3 presents basic ideas of the proposed optimistic concurrency control methods. Section 4 describes an experiment setup and results. Finally, the conclusion of the paper is presented in Section 5.

2 Real-Time Databases in Telecommunication

A *Real-Time Database System* (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. General concepts of real-time databases have been studied extensively. Ramamritham [27] offers an excellent tutorial on these concepts.

As an example application let us consider in more detail database system for telecommunication applications called *Telecommunication Database System*. Recent developments in network and switching technologies have increased the data intensity of telecommunications systems and services. This is clearly seen in many areas of telecommunications including network management, service management, and service provisioning. For example, in the area of network management the complexity of modern networks leads to a large amount of data on network topology, configuration, equipment settings, and so on. In the area of service management there are customer subscriptions, the registration of customers, and service usage (e.g. call detail records) that lead to large databases. The performance, reliability, and availability requirements of data access operations are demanding. Thousands of retrievals must be executed in a second and the allowed down time is only a few seconds per year.

A telecommunication database system must offer real-time access to data [12, 13]. This is due to the fact that most read requests are for logic programs that have exact time limits. If the database cannot give a response within a specific time limit, it is better not to waste resources and hence abort the request. As a result of this, the request management policy should favor predictable response times with the cost of less throughput. The best alternative is that the database can guarantee that all requests are replied to within a specific time interval. The average time limit for a read request is around 50ms. About 90% of all read requests must be served in that time. For updates, the time limits are not as strict. It is better to finish an update even at a later time than to abort the request.

Therefore, telecommunication databases are inherently dynamic and must adapt to workload changes and to counter uncertainties in the system and its environment. This paper concentrates on a concurrency control method for real-time databases. The proposed method dynamically adapts different criticalities of the transactions in the real-time system. The feasibility of the proposed method is evaluated in a real-time database system for telecommunications.

The method presented in this paper is related to two previously presented optimistic concurrency control methods, namely OCC-TI (Optimistic Concurrency Control with Timestamp Intervals) [20] and OCC-DA [16, 17] (Optimistic Concurrency Control with Dynamic Adjustment). OCC-TI is based on the forward validation scheme. The number of transaction restarts is reduced by using dynamic adjustment of the serialization order. OCC-DA is based on the forward validation scheme [8] and on the use of 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. Neither of these optimistic protocols take into account transaction attributes (e.g. priority or criticality).

3 Optimistic Concurrency Control

Optimistic Concurrency Control (OCC) [8, 15], is based on the assumption that conflict is rare, and that it is more efficient to allow transactions to proceed without delays. When a transaction wishes to commit, a check is performed to determine whether a conflict has occurred. There are three phases to an optimistic concurrency control method:

- *Read phase*: The transaction reads the values of all

data items it needs from the database and stores them in local variables. In some methods updates are applied to a local copy of the data and announced to the database system by an operation named *pre-write*.

- *Validation phase*: The validation phase ensures that all the committed transactions have executed in a serializable fashion. For a read-only transaction, this consists of checking that the data values read are still the current values for the corresponding data items. For a transaction that has updates, the validation consists of determining whether the current transaction leaves the database in a consistent state, with serializability maintained.

- *Write phase*: This follows the successful validation phase for update transactions. During the write phase, all changes made by the transaction are permanently stored into the database.

3.1 Real-Time Properties for OCC

Many earlier proposed methods do not include any real-time properties. Therefore, these methods are too ”fair”. A characteristic of most real-time scheduling algorithms is the use of priority-based scheduling [1]. Here transactions are assigned ’priorities’, which are implicit or explicit functions of their deadlines or *criticality* or both. The criticality of a transaction is an indication of its level of importance. However, these two requirements sometimes conflict with each other. That is, transactions with very short deadlines might not be very critical, and vice versa [3].

In real-time systems transactions are scheduled according to their priorities [27]. Therefore, high priority transactions are executed before lower priority transactions. This is true only if the high priority transaction has some database operation ready for execution. If no operation from the higher priority transaction is ready for execution, then the operation from the lower priority transaction is allowed to execute its database operation. Therefore, the operation of the higher priority transaction may conflict with the already executed operation of the lower priority transaction. In traditional methods the higher priority transaction must wait for the release of resources. This is the priority inversion problem. Therefore, data conflicts in the concurrency control should also be based on transaction priorities or criticality or both.

Therefore, the criticality of the transaction is used in place of the deadline in choosing the appropriate conflict resolution method. This avoids the dilemma of the priority-based conflict resolution, yet integrates criticality and deadline so that, not only do the more critical transactions meet their deadlines. The overall goal is to maximize the net worth of the executed transactions to the system.

3.2 Integrated Optimistic Concurrency Control

Each instance of the transaction object has the attributes *priority*, *deadline*, and *criticality*. The developer assigns a value for the deadline based on the estimate or value measured through experiments of worst case execution time. The priority attribute is assigned by the real-time scheduler and is based on the deadline and arrival time. The criticality attribute is assigned by the developer and is static and the same goes for all instances of the same transaction class. We have used the following values coded as numbers:

- **Normal**: The transaction is not essential but should be completed if the execution history is serializable. For this transaction class we use basic conflict resolution.

- **Medium**: The transaction is important and should not be restarted if there is a data conflict with the transaction with normal criticality. For this transaction class we use conflict resolution where a lower criticality transaction is restarted if an active conflicting higher criticality transaction would be restarted because of a data conflict.

- **Critical**: The transaction is critical and should not be restarted even if there is data conflict with the transaction with normal or medium criticality. For this transaction class we use conflict resolution where lower criticality transaction is *always* restarted if there is an active conflicting higher criticality transaction.

Let us assume that the conflict detection algorithm (see Figure 2) has found that a validating transaction T_v and active transaction T_1 are conflicting. Assume that the validating transaction has read the data item X and the active transaction has pre-written the data item X. Assume that $criticality(T_v) = criticality(T_1) = normal$. In this case the conflict is resolved using normal forward adjustment.

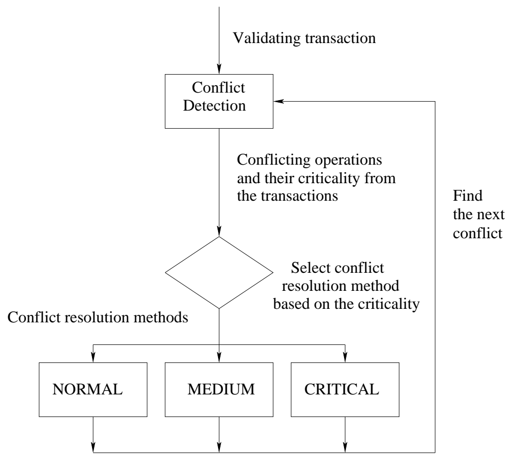


Figure 1: Integrated Optimistic Concurrency Control Method.

Assume that the conflict detection algorithm finds another conflict between the validating transaction and an active transaction T_2 and that the validating transaction has read the data item X and the active transaction has pre-written the data item X. Let $criticality(T_v) = normal < criticality(T_2) = medium$. In this case the conflict is resolved according to the medium forward adjustment. Therefore, the conflict resolution method is selected according to the criticality of the higher criticality transaction (see Figure 1).

Secondly, assume that the conflict detection algorithm finds another conflict between the validating transaction and an active transaction T_3 and that the validating transaction has read the data item X and the active transaction has pre-written the data item X. Let $criticality(T_v) = normal < criticality(T_3) = critical$. In this case the conflict is resolved according to the critical forward adjustment.

Therefore, the proposed method dynamically adapts to different load situations. This is because the conflict resolution method dynamically selects a resolution method based on the criticality of the conflicting transactions. The proposed method offers three different conflict resolution methods. However, inside all the methods the criticality of the conflicting transactions are still taken into account. Therefore, the proposed method can offer better changes for the more critical transaction even if both conflicting transactions belong to the same criticality level. This is because one criticality level can be constructed from a large interval (i.e. the criticality boundaries can be any positive integer values).

In the following sections we present the conflict detection and conflict resolution methods for an integrated and adaptive optimistic concurrency control method.

3.3 Conflict Detection

This section presents conflict detection. Conflict detection is based on forward validation [8]. 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 [20]. Unlike the OCC-TI method, 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 is delayed until a transaction is near completion, there will be more information available for making the choice in resolving the conflict. The proposed method also has a new final timestamp selection method.

Additionally, a new dynamic adjustment of the serialization method is proposed, called *deferred dynamic adjustment of serialization order*. In the deferred dynamic adjustment of serialization order all adjustments of timestamp intervals 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 the 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.

The proposed method resolves conflicts using the timestamp intervals [19] 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. The timestamp interval is adjusted when a transaction validates. In this method 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. The final validation timestamp $TS(T_v)$ of the validating transaction T_v is set to be the current timestamp, if it belongs to the timestamp interval $TI(T_v)$, otherwise $TS(T_v)$ is set to be the maximum value of $TI(T_v)$.

```
conflict_detection(T_v)
  {
    // Select final times-
tamp for the transaction
    TS(T_v) = min(validation_time, max(TI(T_v)));
    // Iterate for all objects read/written
    for ( ∀ D_i ∈ (RS(T_v) ∪ WS(T_v) ))
    {
      if (D_i ∈ RS(T_v))
        TI(T_v) = TI(T_v) ∩ [WTS(D_i),∞[ ;
      if (D_i ∈ WS(T_v))
        TI(T_v) = TI(T_v) ∩ [WTS(D_i),∞[ ∩ [RTS(D_i),∞[ ;
      if (TI(T_v) == []) restart(T_v);

      // Conflict checking and TI calculation
      for ( ∀ T_a ∈ active_conflicting_transactions() )
      {
        if (D_i ∈ (RS(T_v) ∩ WS(T_a)))
          forward_adjustment(T_a,T_v,adjusted);

        if (D_i ∈ (WS(T_v) ∩ RS(T_a)))
          backward_adjustment(T_a,T_v,adjusted);

        if (D_i ∈ (WS(T_v) ∩ WS(T_a)))
          forward_adjustment(T_a,T_v,adjusted);
      }
    }

    // Adjust conflicting transactions
    for ( ∀ T_a ∈ adjusted)
    {
      TI(T_a) = adjusted.pop(T_a);

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

    // Update object timestamps
    for ( ∀ D_i ∈ (RS(T_v) ∪ WS(T_v) ))
    {
      if (D_i ∈ RS(T_v))
        RTS(D_i) = max(RTS(D_i),TS(T_v));
      if (D_i ∈ WS(T_v))
        WTS(D_i) = max(WTS(D_i),TS(T_v));
    }

    commit T_v to database;
  }
```

Figure 2: Conflict Detection Method.

The adjustment of timestamp intervals iterates through the read set (RS) and write set (WS) of the validating transaction. First it is checked that the validating transaction has read from the committed transactions. This is done by checking the object's read and write timestamp. These values are fetched when the first read and write to the current object is made. Then the set of active conflicting transactions is iterated. 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, the deferred dynamic adjustment of the serialization order is used.

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, the current read and write timestamps of the accessed objects are updated and changes to the database are committed.

3.4 Conflict Resolution

Finally, we present an integrated and adaptive method for conflict resolution. We will call this method OCC-IDATI. Below we present forward and backward adjustment algorithms, i.e. the conflict resolution method in the OCC-IDATI. In implementation, three different forward and backward adjustment algorithms are integrated to one forward and one backward adjustment algorithm. Similarly, conflict resolution method selection is integrated inside both algorithms (see Figure 3).

```
forward_adjustment(T_a, T_v, adjusted)
{
  criticality = max(T_a.criticality, T_v.criticality);

  if (T_a ∈ adjusted)
    TI = adjusted.pop(T_a);
  else
    TI = TI(T_a);

    if ( criticality >= NORMAL_CRIT_MIN && criti-
cality <= NORMAL_CRIT_MAX )
        continue;
    if ( criticality >= MEDIUM_CRIT_MIN && criti-
cality <= MEDIUM_CRIT_MAX ) {
          if ( T_v.criticality < T_a.criticality )
            if ( TI == ∅ )
              restart(T_v); /* Validation ends here /*
    }
    if ( criticality >= CRITICAL_CRIT_MIN && crit-
icality <= CRITICAL_CRIT_MAX ) {
          if ( T_v.criticality < T_a.criticality )
              restart(T_v); /* Validation ends here /*
    }

    TI = TI ∩ [TS(T_v) + 1, ∞[ ;
    adjusted.push({(T_a, TI)});
}

backward_adjustment(T_a, T_v, adjusted)
{
  criticality = max(T_a.criticality, T_v.criticality);

  if (T_a ∈ adjusted)
    TI = adjusted.pop(T_a);
  else
    TI = TI(T_a);

    if ( criticality >= NORMAL_CRIT_MIN && criti-
cality <= NORMAL_CRIT_MAX )
        continue;
    if ( criticality >= MEDIUM_CRIT_MIN && criti-
cality <= MEDIUM_CRIT_MAX ) {
          if ( T_v.criticality < T_a.criticality )
              restart(T_v); /* Validation ends here /*
    }
    if ( criticality >= CRITICAL_CRIT_MIN && crit-
icality <= CRITICAL_CRIT_MAX ) {
          if ( T_v.criticality < T_a.criticality )
              restart(T_v); /* Validation ends here /*
          if ( T_v.criticality > T_a.criticality )
              restart(T_a); return;
    }

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

Figure 3: Backward and Forward adjustment for the OCC-IDATI method.

3.5 Correctness Proof

Having described the basic concepts and the algorithm, now the correctness of the algorithm is proven. To prove that a history H produced by OCC-IDATI is serializable, it must be proven that the serialization graph for H, denoted by $SG(H)$, is acyclic [2]. Therefore, following Lemma demonstrate that if there is conflict between the validating transaction and the active transaction then there is a total order between these transactions. This total order is set to the final timestamp of the transactions, i.e. $TS(T_i)$.

LEMMA 3.1 *Let T_1 and T_2 be transactions in a history H produced by the OCC-IDATI 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-IDATI 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-IDATI 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-IDATI 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, $TS(T_1) < TS(T_2)$.* □

To show that every history generated by the OCC-IDATI algorithm is serializable, it is assumed that the algorithm will produce a cycle in the serialization graph. This case is shown to cause contradiction in the following theorem.

THEOREM 3.1 *Every history generated by the OCC-IDATI algorithm is serializable.*

Proof: *Let H denote any history generated by the OCC-IDATI 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 3.1 $TS(T_1) < TS(T_2) < ... < TS(T_n) < TS(T_1)$. By induction $TS(T_1) < TS(T_1)$. This is a contradiction. Therefore no cycle can exist in $SG(H)$ and thus the OCC-IDATI algorithm produces only serializable histories.* □

4 Results from Experiments

We have carried out a set of experiments in order to examine the feasibility of the OCC-IDATI method in practice. The prototype system used is based on the *Real-Time Object-Oriented Database Architecture for Intelligent Networks* (RODAIN) specification [14, 22, 25], which is an architecture for a real-time, object-oriented, and fault-tolerant database management system. The RODAIN prototype system is a main-memory database, which uses priority and criticality based scheduling and optimistic concurrency control. All experiments were executed in the RODAIN prototype database running on Pentium Pro 200MHz and 64 MB of main memory with the Chorus/ClassiX operating system [26].

In the test environment, transactions arrive to a specific user interface 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. The reported values are the means of the repetitions. In the experiments, we examined how well our OCC-IDATI method performs when compared to the OCC-DATI method [23] and to the OCC-TI method [19].

The test database represents a typical Intelligent Network (IN) service [11]. The database is modeled according a Virtual Private Network information service. The size of the database is 30 000 objects. Object classes and their relationships are presented in Figure 4.

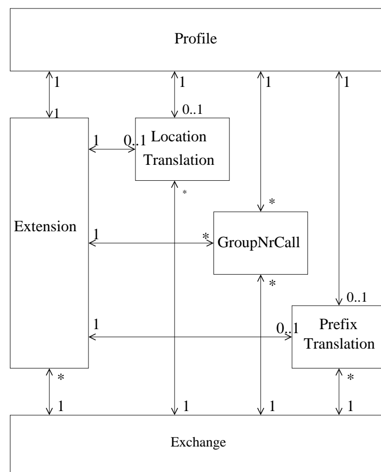


Figure 4: Object model schema of the test database.

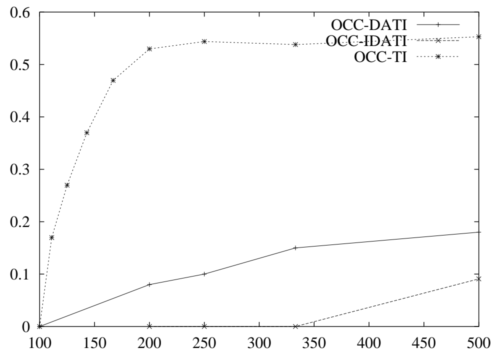
We will briefly introduce the definitions and purposes of each class. There are six classes defined:

1. Class *Profile* includes all necessary information needed for customer management.
2. Class *Exchange* includes all necessary information of switches in a system.
3. Class *PrefixTranslation* is used to give shorter prefix numbers for exchanges. The user may replace the real exchange number with a corresponding prefix number. This allows users to make ”local calls” to other exchanges.
4. Class *Extension* defines characteristics of extensions. Extension maps the Number to the real telephone number, which is stored into attribute SubscriberNumber.
5. Class *LocationTranslation* is used to determine to which extension a forwarded number should be directed.
6. Class *GroupNrCall* is used to map several terminating lines in any number of locations or zones to a unique directory number, e.g. 118.

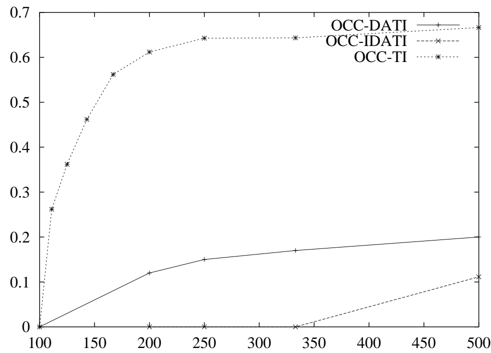
The following transactions are implemented and used in these tests:

- The **Find Subscriber** transaction is used to search a specific subscriber from the database. The transaction returns a link to the location of the subscriber in the database. The criticality of this transactions is critical.
- The **Get New Destination Number** transaction can be used in three different scenarios: abbreviated number, forwarded number and group number. Transaction returns the number to connected. The criticality of this transaction is medium.
- The **Get Subscribers Basic Data** transaction is used to fetch basic data of the subscriber. The criticality of this transactions is medium.
- The **Update Subscriber Data** transaction is used to modify some of the subscriber data. The criticality of this transactions is normal.
- The **Location Update** transaction is used to search a subscriber's data and update the subscriber's location information. The criticality of this transactions is normal.

In the first set of experiments, a fixed fraction of write transactions was used. The arrival rate of transactions was the varying parameter. OCC-IDATI clearly offers the best performance in all tests (see Figure 5). This confirms that the overhead for supporting dynamic adjustment in OCC-IDATI is smaller than the one in OCC-TI and in OCC-DATI.



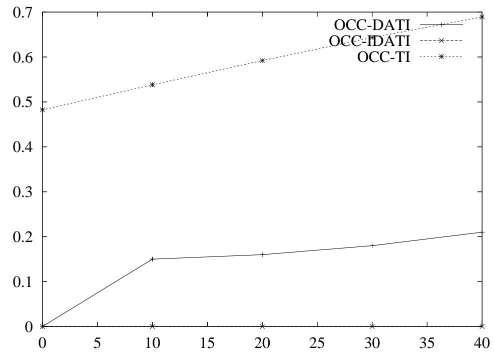
(a) 10% writes



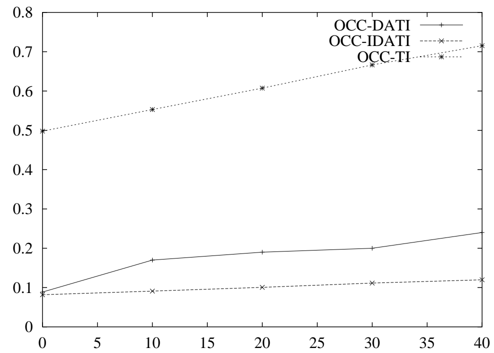
(b) 30% writes

Figure 5: Comparison with varying transaction arrival rate.

Figure 6 shows the miss-ratio transactions when the transaction's write fraction is varied. Figure 6 demonstrates how the OCC-IDATI favors critical transactions. OCC-IDATI clearly offers better chances for critical transactions to complete according to their deadlines. The results clearly indicate that OCC-IDATI meets the goal of favoring critical transactions.



(a) 333trans/s



(b) 500 trans/s

Figure 6: Comparison with varying transaction write fraction.

5 Summary

Although the optimistic approach has been shown to have a better performance than locking protocols in firm real-time database systems, it has problems of unnecessary restarts and a high restart overhead. In this paper, we have presented an integrated concurrency control method for real-time database systems. The proposed method provides both serializability and notices criticality of the transactions. We have proposed an optimistic concurrency control protocol called OCC-IDATI that takes into account the criticality of transactions. It has several advantages over the other concurrency control protocols. The protocol maintains all the nice properties of forward validation, a high degree of concurrency, freedom from deadlock, and early detection and resolution of conflicts, resulting in less waste of resources as well as a smaller number of restarts. All of these are important to the performance of RTDBSs and contribute to greater chances of meeting transaction

deadlines.

Compared to other OCC protocols that use dynamic serialization order adjustment, the proposed method OCC-IDATI is much more efficient and its overhead is smaller. Performance studies presented here confirm that OCC-IDATI outperforms both OCC-TI and OCC-DATI. The most important feature of the OCC-IDATI is that it clearly offers better chances for the critical transactions to complete before their deadlines when compared to the OCC-DATI and OCC-TI. The results clearly indicate that OCC-IDATI meets the goal of favoring critical transactions. Experiment results using real-time database system for telecommunications clearly indicate that the proposed method is able to dynamically adapt changing workload situations.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. *ACM SIGMOD Record*, 17(1):71–81, March 1988.

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

[3] S. R. Biyabani, J. A. Stankovic, and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pages 487–507, Huntsville, Alabama, USA, 1988. IEEE Computer Society Press.

[4] 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.

[5] A. Datta, S. H. Son, and V. Kumar. Is a bird in the hand worth more than two in the bush? limitations of priority cognizance in conflict resolution for firm real-time database systems. *IEEE Transactions on Computers*, 49(5):482–502, May 2000.

[6] 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 *Proceedings of the Second International Workshop on Real-Time Databases: Issues and Applications*, pages 167–180, Burlington, Vermont, USA, 1997. Kluwer Academic Publishers.

[7] 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.

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

[9] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 94–103, Lake Buena Vista, Florida, USA, 1990. IEEE Computer Society Press.

[10] 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, Nashville, Tennessee, 1990. ACM Press.

[11] ITU. *Introduction to Intelligent Network Capability Set 1. Recommendation Q.1211*. ITU, International Telecommunications Union, Geneva, Switzerland, 1993.

[12] ITU. *Distributed Functional Plane for Intelligent Network CS-1. Recommendation Q.1214*. ITU, International Telecommunications Union, Geneva, Switzerland, 1994.

[13] ITU. *Draft Q.1224 Recommendation IN CS-2 DFP Architecture*. ITU, International Telecommunications Union, Geneva, Switzerland, 1996.

[14] 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, London, 1997. Kluwer Academic Publishers.

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

[16] 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, 1995. Springer.

[17] K.-W. Lam, K.-Y. Lam, and S. Hung. Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *Proceedings of the IEEE Real-Time Technology and Application Symposium*, pages 174–179, Chigago, Illinois, 1995. IEEE Computer Society Press.

[18] K.-Y. Lam, S.-L. Hung, and S. H. Son. On using real-time static locking protocols for distributed real-time databases. *The Journal of Real-Time Systems*, 13(2):141–166, September 1997.

[19] 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, Raleigh-Durham, NC, USA, 1993. IEEE Computer Society Press.

[20] 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.

[21] J. Lindström. Extensions to optimistic concurrency control with time intervals. In *Proceedings of 7th International Conference on Real-Time Computing Systems and Applications*, pages 108–115, Cheju Island, South Korea, 2000. IEEE Computer Society Press.

[22] J. Lindström, T. Niklander, P. Porkka, and K. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Databases in Telecommunications*, Lecture Notes in Computer Science, vol 1819, pages 158–173, Edinburgh, UK, Co-located with VLDB-99, 1999.

[23] J. Lindström and K. Raatikainen. Dynamic adjustment of serialization order using timestamp intervals in real-time databases. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pages 13–20, Hong Kong, China, 1999. IEEE Computer Society Press.

[24] K. Marzullo. Concurrency control for transactions with priorities. Tech. Report TR 89-996, Department of Computer Science, Cornell University, Ithaca, NY, May 1989.

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

[26] Dick Pountain. The Chorus microkernel. *Byte*, pages 131–138, January 1994.

[27] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1:199–226, April 1993.

[28] S. H. Son, S. Park, and Y. Lin. An integrated real-time locking protocol. In *Proceedings of the 8th International Conference on Data Engineering*, pages 527–534, Tempe, Arizona, USA, 1992. IEEE Computer Society Press.

[29] J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32(6):29–36, June 1999.