Using Priorities in Concurrency Control for RTDBSs

Jan Lindström
University of Helsinki, Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland
Email: {jan.lindstrom}@cs.Helsinki.FI
Seminar on Real-Time and Embedded Systems

Abstract

In a real-time database system, it is difficult to meet all timing constraints due to the consistency requirements of the underlying database. Real-time database transaction scheduling requires the development of efficient concurrency control protocols that try to maximize the number of transactions satisfying their real-time constraints. Therefore transaction priorities should be taken account when making decisions in concurrency control. Although optimistic approach has been shown to be better than locking protocols for RTDBS, it has the problems of unnecessary restarts and heavy restart overhead. In this paper, we propose a new optimistic concurrency control protocol called OCC-PDATI which uses priority information in conflict resolution. In OCC-PDATI number of transaction restarts is minimized by dynamically adjusting the serialization order of the conflicting transactions. Need for dynamic serialization order adjustment is checked and serialization order is updated in the validation phase. This provides more freedom to adjust serialization order of conflicting transactions. OCC-PDATI has several advantages over the other optimistic concurrency control protocols. The protocol maintains all the nice properties with forward validation, high degree of concurrency, free from deadlock, early detection and resolution of conflicts, resulting in both less wasted resources and smaller number of restarts. All of these are important to the performance of RTDBS and contribute to greater chances of meeting transaction deadlines. Performance studies of our protocol has been carried out in RODAIN RTDBMS and the results confirm the believe that OCC-PDATI clearly offers better changes for high priority transactions to complete according to their deadlines.

1 Introduction

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

Many alternatives to two-phase locking for real-time systems have been proposed and studied [3, 11, 16]. Among them is a class of concurrency control schemes based on the well-know optimistic approach [18, 10].

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 to resolve 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 this paper, we propose a new optimistic concurrency control protocol which used priority information in conflict resolution, called OCC-PDATI. In this paper we concentrate on firm real-time transaction models. OCC-PDATI is a fully optimistic protocol and uses forward adjustment.

The rest of the paper is organized as follows. Section 2 introduces recent related work. Section 3 introduces the basic mechanisms of our new optimistic CC protocol. Section 4 presents the OCC-PDATI algorithm. Finally, the conclusion of the paper is presented in Section 5.

2 Using priorities in Concurrency Control

2.1 Related work

In recent years, different real-time concurrency control protocols have been proposed for single-site RTDBS by incorporating transaction priority information for resolving data conflicts into some well-known non-real-time concurrency control protocols. This work on concurrency control can be divided in three basic concurrency control protocols: locking [9], optimistic [18, 10] and others.

Most of these are based on *dynamic two phase locking* (D2PL) due to its popularity for conventional database system and simplicity. In D2PL, transactions must obtain their locks on data objects just before accessing the data objects. If the requested lock is used in conflicting mode by another transaction, the requesting transaction will be blocked until the lock is not in conflicting mode.

One of the earliest suggestions of real-time protocol based on D2PL is *Hight Priority Two Phase Locking* (H2PL) [1, 31] which uses the same priority for both CPU and data scheduling. In H2PL, if a higher priority transaction request to use a lock which is used by a lower priority transaction, the lower priority transaction must release the lock and restart. To reduce blocking time *priority inheritance* [14] and *conditional priority inheritance* [14] have been suggested. In these protocols the lower priority transaction inherits higher priority transactions priority. *Hybrid Two Phase Locking* (HB2PL) [15] adopts the concept of cautious waiting to prevent locking induced trashing and to reduce the impact of priority inversion.

Other ideas include *Two Phase Locking with Ordered Sharing* (2PL-OS) [2], *Two Phase Locking with Ordered Sharing and Before Images* (2PL-OS/BI) [2], *Two Phase Locking with Avoiding Cascaded Aborts* (ACA 2PL-OS) [2], and *Preemptive Two Phase Locking* (P2PL) [27]

An alternative for concurrency control in RTDBS is to use *real-time Static 2PL* (RT-S2PL) [23]. In S2PL, the required locks of transaction are assumed to be known before its execution. A transaction has to obtain all its required locks before the processing of its operations. *Real-Time Static Two Phase Locking* (RT-S2PL), *Real-Time Static Two Phase Locking with Resource Priority* (RT-S2PL-RP), and *Real-Time Static Two Phase Locking with Priority Inheritance* (RT-S2PL-PI) [23] are variants of S2PL.

The *Affected Set Priority Ceiling* (ASPC) protocol uses the affected sets [4] of each method of each object to determine the compatibilities of the methods of an object. Using affect set semantics, the critical section is a method lock. The ASPC protocol assigns a conflict priority ceiling to each method of each object. The conflict priority ceiling of a method m is the priority of the highest priority transaction that will ever lock a method that is not compatible with method m. The ASPC protocol allows a transaction T to receive a lock on a critical section if and only if: The priority of T is strictly higher than the conflict priority ceiling of locks held by all other transactions.

The *Alternative Version Concurrency Control* (AVCC) [12, 35] maintains more than two version of a transaction at the same time. By maintaining the stopped version and initiating the restarted version, AVCC can reduce wasted restart and wasted execution.

There are also several protocols designed for a *priority ceiling protocol* [6, 21, 22, 32]. The priority ceiling protocol is designed for systems with a fixed priority scheme. In this protocol, a priority ceiling is defined for every critical section. The ceiling value is the priority of the highest priority task which may enter the critical section.

We have found only a few priority-driven concurrency control protocols which use optimistic method. These include *Optimistic concurrency control using locking* (OCCL) [13] which is based on a locking mechanism, *Optimistic Concurrency Control with Adaptive Priority* (OCC-APR) [8], *Adaptive Priority Fan Out (OCC-APFO)*, and [7] *Adaptive Priority Fan-out Sum (OCC-APFS)* [7]. OCC-APR and its variants attempt to reduce the number of restarts by exploiting and extending the capabilities of priority insensitive algorithms. However experiments and the analysis of the corresponding results showed that there appears to be little advantage to be gained by incorporating priority cognizance in the validation-phase confict resolution of optimistic concurrency control methods for firm real-time databases.

The design of some protocols is based on pre-declaration as in *Pre-declaration Protocol* [34], timestamps as in *Priority Timestamp Order* (PTO) [27], and serialization graph as in *Priority Serialization Graph Testing* (PSGT) [27].

2.2 Optimistic Concurrency Control

In *Optimistic Concurrency Control* (OCC) [18], 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, the *read phase*, the *validation phase*, and the *write phase*. The read phase is the normal execution of the transaction. Write operations are performed on private data copies in the local workspace of the transaction. This kind of operation is called *pre-write*.

Traditional two-phase locking contains wasted restart and wasted wait problems. Optimistic protocols contains wasted execution and unnecessary restart problems. Below we define these problems:

- **wasted restart:** A wasted restart happens if a higher priority transaction aborts a lower priority transaction and then the higher priority transaction is discarded as it misses its deadline.

- **wasted wait:** A wasted wait happens if a lower priority transaction waits for the commit of higher priority transaction and later the higher priority transaction is discarded as it misses its deadline.

- **wasted execution:** A wasted execution happens when a lower priority transaction in the commit time validation phase is restarted due to a conflicting higher priority transaction which has not finished yet.

- **unnecessary restart:** An unnecessary restart happens when a transaction in the validation phase is restarted even when execution was serializable.

In real-time database systems, the conflict resolution of a transactions should be based on transaction priority. Therefore we need to develop an optimistic concurrency control protocol where the decision which transaction is restarted should be based on transaction priority.

2.3 Basic Notation

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

The major performance problem with OCC protocols is the late restart. Therefore, one important mechanism to improve the performance of OCC protocols is to reduce the number of restarted transactions. In conventional OCC protocols many transaction restarts are unnecessary. For example, consider transactions T_1, T_2 and history H_1:

$$T_1 : r_1[x]w_1[x]v_1c_1$$

$$T_2 : r_2[x]w_2[y]v_2c_2$$

$$H_1 : r_2[x]w_2[y]r_1[x]w_1[x]v_1$$

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

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

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

2.4 Dynamic Adjustment

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

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

Suppose we have a validating transaction T_v and a set of active transactions $T_j (j = 1, 2, ..., n)$. There are three possible types of data conflicts which can cause a serialization order between T_v and T_j:

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

   Read-write conflict between T_v and T_j can be resolved by adjusting the serialization order between T_v and T_j as $T_v \rightarrow T_j$ so that the read of T_v cannot be affected by T_j's write. This type of serialization adjustment is called *forward ordering* or *forward adjustment*.

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

$$
\begin{array}{l}
TI \;=\; TI(T\_j) \;\cap\; [TS(T\_v)+1, \infty[; \\
\;\;\textbf{if} \;(\; pri(T\_v) \;<\; pri(T\_j) \;) \\
\;\;\;\;\;\;\textbf{if} \;(\; TI \;==\; \emptyset \;) \\
\;\;\;\;\;\;\;\;restart(T\_v) \\
TI(T\_j) \;=\; TI
\end{array}
$$

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

   Write-read conflict between T_v and T_j can be resolved by adjusting the serialization order between T_v and T_j as $T_j \rightarrow T_v$. It means that the read phase of T_j is placed before the write of T_v. This type of serialization adjustment is called *backward ordering* or *backward adjustment*.

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

$$
\begin{array}{l}
TI \;=\; TI(T\_j) \;\cap\; [0, TS(T\_v)-1]; \\
\;\;\textbf{if} \;(\; pri(T\_v) \;<\; pri(T\_j) \;) \\
\;\;\;\;\;\;\;\;restart(T\_v) \\
TI(T\_j) \;=\; TI
\end{array}
$$

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

   Write-write conflict between T_v and T_j can be resolved by adjusting the serialization order between T_v and T_j as $T_v \rightarrow T_j$ such that the write of T_v cannot overwrite T_j's write (forward ordering).

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

$$
\begin{array}{l}
TI \;=\; TI(T\_j) \;\cap\; [TS(T\_v)+1, \infty[; \\
\;\;\textbf{if} \;(\; pri(T\_v) \;<\; pri(T\_j) \;) \\
\;\;\;\;\;\;\textbf{if} \;(\; TI \;==\; \emptyset \;) \\
\;\;\;\;\;\;\;\;restart(T\_v) \\
TI(T\_j) \;=\; TI
\end{array}
$$

3 OCC-PDATI

In this section we present a new optimistic concurrency control protocol named OCC-PDATI. OCC-PDATI is based on forward validation [10] and our earlier optimistic protocol OCC-DATI [26]. 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 [25]. 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-PDATI is delayed until a transaction is near completion, there will be more information available for making the choice in resolving the conflict. OCC-PDATI also has a new final timestamp selection method compared to OCC-TI. We have also identified an unnecessary restart problem in OCC-TI [26].

OCC-DA [19, 20] 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-PDATI 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 instead of one in OCC-PDATI (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-PDATI.

We have also used a *deferred dynamic adjustment of serialization order*. In the deferred dynamic adjustment of serialization order all adjustments of the 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.

The OCC-PDATI protocol resolves conflicts using the time intervals [24] 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-PDATI every transaction is assigned a timestamp interval (TI). At the start of the transaction, the timestamp interval of the transaction is initialized as $[0, \infty[$, i.e., the entire range of timestamp space. This timestamp interval is used to record a temporary serialization order during the validation of the transaction.

At the beginning of the validation (Figure 1), the final timestamp of the validating transaction $TS(T_v)$ is determined from the timestamp interval allocated to the transaction T_v. The timestamp intervals of all other concurrently running and conflicting transactions must be adjusted to reflect the serialization order. We set the final validation timestamp $TS(T_v)$ of the validating transaction T_v to be 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 $TI(T_v)$.

```
occpdati_validate(T_v)
{
  // Select final timestamp for the transaction
  TS(T_v) = min(validation_time, max(TI(T_v)));
  // Iterate for all objects read/written
  for ( ∀ 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 1: Validation algorithm.

The adjustment of timestamp intervals iterates through the read set (RS) and write set (WS) of the validating transaction. First we check that the validating transaction has read from committed transactions. This is done by checking the object's read and write timestamp. These values are fetched when the first access to the current object is made. Then we iterate the set of active conflicting transactions. When access has been made to the same objects both in the validating transaction and in the active transaction, the temporal time interval of the active transaction is adjusted. Thus we use deferred dynamic adjustment of the serialization order.

Time intervals of all conflicting active transactions are adjusted after the validating transaction is guaranteed to commit. If the validating transaction is aborted no adjustments are done. Non-serializable execution is detected when the timestamp interval of an active transaction becomes empty. If the timestamp interval is empty the transaction is restarted.

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

Figure 2 shows a sketch of implementing a dynamic adjustment of the serialization order using timestamp intervals and transaction priority information.

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

   TI = TI(T_a) ∩ [TS(T_v) + 1, ∞[;

    if ( pri(T_v) < pri(T_a) )
       if ( TI == ∅ )
         restart(T_v);

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

backward_adjustment(T_a, T_v, adjusted)
{
   if (T_a ∈ adjusted)
     TI = adjusted.pop(T_a);
   else
     TI = TI(T_a);

   TI = TI(T_a) ∩ [0, TS(T_v) − 1]

    if ( pri(T_v) < pri(T_a) )
         restart(T_v);

   adjusted.push({(T_a, TI)});
}
```

Figure 2: Backward and Forward adjustment.

4 Results from Experiments

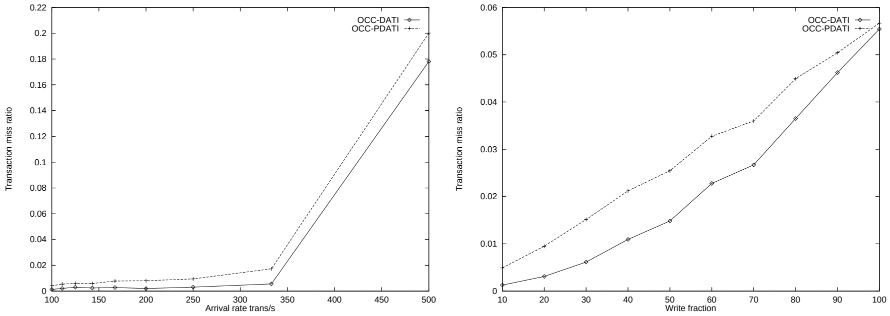
We have carried out a set of experiments in order to examine the feasibility of our algorithm in practice. *Real-Time Object-Oriented Database Architecture for Intelligent Networks* (RODAIN) [17, 28, 29, 33] is an architecture for a real-time, object-oriented, and fault-tolerant database management systems. RODAIN consists of a main-memory database, priority based scheduling and optimistic concurrency control. All experiments were executed in the RODAIN prototype database running on a machine with one processor Pentium Pro 200MHz and 64 MB of main memory with the Chorus/ClassiX operating system. The test environment is a reduced subset of the RODAIN architecture.

All transactions arrive at the RODAIN prototype through a specific user subsystem. In these tests we have used a special user subsystem that receives the arriving transactions from an off-line generated test file. Every test session contains 10 000 transactions and is repeated at least 20 times. Reported values are means of the repetitions. In particular, we examined how well our OCC-PDATI algorithm performs when compared to the OCC-DATI algorithm [26].

The test database represents a typical Intelligent Network (IN) service. The size of the database is 30 000 objects. We used four different transactions named R1, R2, W1 and W2. Transactions R1 and R2 are a read-only service provision transaction. Transaction R1 reads user one user profile. Transaction R2 represents abbreviated dialing or call forwarding services in IN [30].Transactions W1 and W2 are an update service provision transactions. Transaction W1 implements customer profile management service in IN CS-1. Transaction W2 implements updates to abbreviated dialing or call forwarding services in IN CS-1. All transactions are firm real-time transactions.

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

In the first series of experiments we have used a fixed fraction of write transactions (W1, W2) (the rest are R1 and R2) and a varied arrival rate of transactions. In the second series of experiments we have used a fixed arrival rate of transactions and varied the fraction of write transactions (W1, W2) from 10% to 100%. The rest are read transactions (R1, R2).

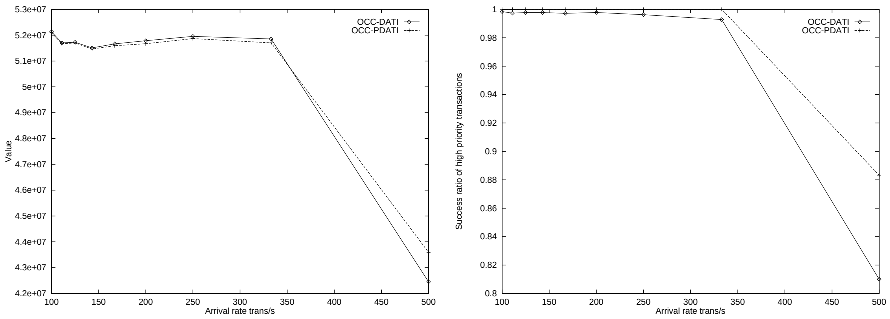


(a) 250 trans/s 20% writes

(b) 250 trans/s

Figure 3: OCC algorithms compared.

As expected, there is some performance penalty when priority information is used in dynamic adjustment of serialization order. But this penalty does not seem to be too heavy. The average performance of OCC-DATI and OCC-PDATI is almost the same. In RODAIN we have also used value attribute in priority based scheduling. Figure 4(a) shows the total value achieved by successfully completed transactions. As expected, the values received by both OCC-DATI and OCC-PDATI are very near each other. Finally Figure 4(b) shows the success ratio of high priority transactions. OCC-PDATI clearly offers better chances for high priority transactions to complete according to their deadlines.



(a) Value to system, 20% writes

(b) High priority transactions, 20% writes

Figure 4: OCC algorithms compared.

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 propose a new optimistic concurrency control protocol called OCC-PDATI. It has several advantages over the other concurrency control protocols. The protocol maintains all the nice properties with forward validation, a high degree of concurrency, freedom from deadlock, and early detection and resolution of conflicts, resulting in both less waste of 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. Method also incorporates transaction priorities to conflict resolution. Compared with the OCC-DATI protocol that uses dynamic serialization order adjustment, our new method OCC-PDATI offers the same efficiency and the overhead is only slightly larger. There is no need to check for conflicts while a transaction is still in its read phase. All the checking is performed in the validation phase. As conflict resolution between the transactions in OCC-PDATI is delayed until a transaction is near completion, there will be more information available for making the choice in resolving the conflict. We have also used the deferred dynamic adjustment of the serialization order, which further minimizes unnecessary restarts. Experiments show that OCC-PDATI gives the same performance as OCC-DATI which was found superior to other well known OCC protocols in [26]. The most important feature of the OCC-PDATI is that it clearly offers better chances for the high priority transactions to complete before their deadlines when compared to the OCC-DATI.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.

[2] D. Agrawal, A. E. Abbadi, and R. Jeffers. Using delayed commitment in locking protocols for real-time databases. In *ACM SIGMOD'92*, 1992.

[3] J. H. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-free transactions for real-time systems. In *Proc. of the First International Workshop on Real-Time Databases: Issues and Applications*, Newport Beach, California, March 7-8, 1996.

[4] B. Badrinath and K. Ramamritham. Synchronizing transactions on objects. *IEEE Transactions on Computers*, 37:541–547, May 1988.

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

[6] M.-I. Chen and K.-J. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, pages 325–346, 1990.

[7] 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.

[8] 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*, Newport Beach, California, March 7-8, 1997.

[9] 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.

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

[11] 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.

[12] D. Hong, S. Chakravarthy, and T. Johnson. Locking based concurrency control for integrated real-time database systems. In *Proc. of the First International Workshop on Real-Time Databases: Issues and Applications*, Newport Beach, California, March 7-8, 1996.

[13] 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.

[14] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley. On using priority inheritance in real-time databases. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 210–221, Los Alamitos, Calif., 1991. IEEE, IEEE Computer Society Press.

[15] S.-L. Hung and K.-Y. Lam. Locking protocols for concurrency control in real-time database systems. *ACM SIGMOD Record*, 21(4):22–27, December 1992.

[16] R. Kataoka, T. Satoh, and U. Inoue. A multiversion concurrency control algorithm for concurrent execution of partial update and bulk retrieval transactions. In *Proceedings of the 10th Phoenix Conference on Computers and Communications*, pages 130–136, 1991.

[17] 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.

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

[19] 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.

[20] 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.

[21] K.-W. Lam, V. Lee, S.-L. Hung, and K.-Y. Lam. An augmented priority ceiling protocol for hard real-time systems. *Journal of Computing and Information, Special Issue: Proceedings of Eighth International Conference of Computing and Information (ICCI'96)*, 2(1):849–866, June 1996.

[22] 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.

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

[24] 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.

[25] 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.

[26] J. Lindström and K. Raatikainen. Dynamic adjustment of serialization order using timestamp intervals in real-time dbms. In *To appear in Proc. of 6th International Conference on Real-Time Computing Systems and Applications*, 1999.

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

[28] 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.

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

[30] K. E. E. Raatikainen. Information aspects of services and service features in intelligent network capability set 1. Report C-1994-45, University of Helsinki, Dept. of Computer Science, Helsinki, Finland, September 1994.

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

[32] M. Squadrito, L. C. DiPippo, and V. F. Wolfe. Towards priority ceilings in object-based semantic real-time concurrency control. In *Proc. of the First International Workshop on Real-Time Databases: Issues and Applications*, Newport Beach, California, March 7-8, 1996.

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

[34] Ö. Ulusoy and A. Buchmann. A real-time concurrency control protocol for main-memory database systems. *Information Systems*, 23(2):109–125, 1998.

[35] I. Yoon and S. Park. Enhancement of alternative version concurrency control using dynamic adjustment of serialization order. In *Proc. of the Second International Workshop on Real-Time Databases: Issues and Applications*, Burlington, Vermont, USA, September 18-19, 1997.