AN OPTIMISTIC CONCURRENCY CONTROL PROTOCOL FOR REAL-TIME DATABASE SYSTEMS

Juhnyoung Lee and Sang H. Son

Department of Computer Science, University of Virginia, Charlottesville, VA 22903, U.S.A.

ABSTRACT

Transactions in real-time database systems are associated with certain timing constraints derived either from temporal consistency requirements of data or from requirements imposed on system reaction time. Fundamental requirements of real-time database systems are *timeliness*, i.e., the ability to produce expected transaction results early or at the right time, and *predictability*, i.e., the ability to function as deterministic as necessary to satisfy system specifications including timing constraints. There are a number of issues that have to be addressed in processing real-time transactions. To achieve the fundamental requirements, not only conventional transaction processing mechanisms have to be tailored to take timing constraints into consideration, but also new mechanisms that have not been required in conventional transaction processing need to be designed and added. In this paper, we focus on the problem of concurrency control for processing real-time transactions, and propose an optimistic concurrency control protocol. The proposed protocol employs *priority-based conflict resolution* schemes developed on *forward validation*. In addition, it utilizes the notion of *lazy serialization* implemented using dynamic timestamp allocation and dynamic adjustment of timestamp intervals. With these features, the proposed protocol is expected to produce transaction results in a timely manner.

1. INTRODUCTION

Real-time database systems are vital to a wide range of operations. As computers have been faster and more powerful, and their use more widespread, real-time database systems have grown larger and become more critical. For example, they are used in program stock trading, telephone switching systems, network management, automated factory management, and command and control systems. More specifically, in the program stock market application, we need to monitor the state of the stock market and update the database with new information. If the database is to contain an accurate representation of the current market, then this monitoring and updating process must meet certain timing constraints. Also, in this system, we need to satisfy certain real-time constraints in reading and analyzing information in database in order to respond to a user query or to initiate a trade in the stock market. For other examples given above, we can consider similar operations with timing constraints.

This work was supported in part by ONR, by DOE, by IBM, and by CIT.

All of these real-time database operations are characterized by (1) their time constrained access to data and (2) access to data that has temporal validity. They involve gathering data from the environment, processing of gathered information in the context of information acquired in the past, and providing timely response. They also involve processing not only archival data but also *temporal data* which loses its validity after a certain time interval. Both of the temporal nature of the data and the response time requirements imposed by the environment make transactions possess timing constraints in the form of either periods or deadlines. Therefore, the correctness of real-time database operation depends not only on the logical computations carried out but also on the time at which the results are delivered. The goal of real-time database systems is to meet timing constraints of transactions.

One key point to note here is that real-time computing does not imply fast computing. Rather than being fast, more important properties of real-time (database) systems should be *timeliness*, i.e., the ability to produce expected results early or at the right time, and *predictability*, i.e., the ability to function as deterministic as necessary to satisfy system specifications including timing constraints [Stan90]. Fast computing which is busy doing the wrong activity at the wrong time is not helpful for real-time computing. While the objective of real-time computing is to meet the individual timing constraint of each activity, the objective of fast computing is to minimize the average response time of a given set of activities. Fast computing is helpful in meeting stringent timing constraints, but fast computing alone does not guarantee timeliness and predictability. In order to guarantee timeliness and predictability, we need to handle explicit timing constraints, and to use time-cognizant techniques to meet deadlines or periodicity associated with activities.

There are a number of issues that have to be addressed in processing real-time transactions. To achieve the fundamental requirements, i.e., timeliness and predictability, not only conventional transaction processing mechanisms have to be tailored to take timing constraints into consideration, but also new mechanisms that have not been required in conventional transaction processing need to be designed and added. In this paper, we focus on the problem of concurrency control for processing real-time transactions

2. RELATED WORK

Most work on real-time concurrency control uses the notion of serializability as the correctness criteria [Abbo88, Abbo89, Buch89, Hari90, Hari90b, Huan89, Huan91, Lin90,

Sha88, Stan88], while some consider relaxation of consistency requirements based on the argument that it is desirable to trade off timeliness with data consistency [Lin89, Liu88, Son88]. In enforcing serializability, conventional concurrency control methods such as two-phase locking (2PL), timestamp ordering (TO), and optimistic concurrency control (OCC) have been used as the basis of real-time concurrency control. Most current work combines the conventional concurrency control schemes with priority-based conflict resolution methods such as *priority abort* [Abbo88, Huan91], *priority inheritance* [Sha88], *priority ceiling* [Sha88], *priority wait* [Huan91], and *serialization order adjustment* [Lin90].

Most of the initial work on real-time concurrency control has been conducted on utilizing 2PL [Abbo88, Abb89, Huan89, Sha88, Stan88]. This is not surprising because 2PL has been well studied in traditional database systems and is being widely used in commercial database systems. Besides, recovery mechanisms for use with 2PL are well understood. 2PL uses pure blocking for conflict resolution. The beneficial effect of blocking is conservation of resources, which makes 2PL advantageous particularly under resource-limited situations. However, blocking results in prevention of the progress of transaction execution. Subsequently, a low degree of concurrency is resulted, and conflict ratio is increased rapidly. Moreover, 2PL has inherent problems such as possibility of deadlocks and unpredictable blocking duration, which become even more serious for real-time transaction processing.

OCC scheduler [Kung81, Haer84] uses abort and restart to serialize concurrent data operations, thereby avoids blocking. Thus, OCC is free from deadlock. In addition, it has a potential for high degree of parallelism. These features of OCC make it promising particularly for real-time transaction processing. However, the abort-based conflict resolution of OCC has the problem of wasted resources and time, which becomes more serious for real-time transaction scheduling.

In OCC protocols that perform *backward validation*, the validating transaction either commits or aborts depending on whether it has conflicts with transactions that have already committed. Thus, this validation scheme does not allow us to take transaction characteristics into account. In *forward validation* [Haer84], however, either the validating transaction or conflicting ongoing transactions can be aborted to resolve conflicts. This validation scheme is advantageous in real-time database systems, because it may be preferable not to commit the validating transaction, depending on the timing characteristics of the validating transaction and the conflicting ongoing transactions. A number of real-time concurrency control methods based on OCC using forward validation scheme have been studied [Hari90, Hari90b, Huan91].

The rationale for OCC is based on an "optimistic" assumption regarding run-time conflicts: if only few run-time conflicts are expected, we can assume that most execution is serializable [Bern87]. Therefore OCC simultaneously avoids blocking and restarts in the optimistic situations. Unfortunately, however, this optimistic assumption on transaction behavior may not always be true in real world situations. In a database system where run-time conflicts are not rare, OCC depends on transaction restarts to eliminate nonserializable executions.

The adverse effect of transaction restarts for serialization is that resource and time are wasted. Especially in OCC, because data conflicts are detected and resolved only during the validation phase, a transaction can end up aborting after having used resources and time for most of the transaction's execution. When the transaction is restarted, previously performed work has to be redone. This problem of *time and resource waste* becomes even more serious in real-time transaction scheduling, because it reduces the chances of meeting the deadlines of transactions.

Another problem of OCC is *unnecessary aborts*. This problem is often caused by imperfect validation tests used in OCC protocols. Many validation test schemes are based on intersection of the read sets and write sets of transactions rather than on the actual execution order of transactions, since in general it is difficult to record and use entire execution history efficiently. Hence sometimes a validation process using read sets and write sets erroneously concludes that a nonserializable execution has occurred when it has not in actual execution. We call such a conflict virtual. A *virtual conflict* leads to one or more unnecessary transaction aborts. This problem of unnecessary aborts also results in waste of resource and time, and is serious in real-time transaction processing.

The concurrency control protocol proposed in this paper is based on OCC scheme using forward validation [Haer84, Robi82], which was also used as the basis for OCC-based real-time concurrency control schemes proposed in [Hari90b, Huan91]. Its goal is to overcome those two problems of basic OCC scheme described above to process efficiently transactions in a real-time database system. The proposed protocol employs forward validation and priority-based conflict resolution schemes developed for forward validation. In addition, it utilizes the notion of lazy serialization implemented using dynamic timestamp allocation [Baye82] and dynamic adjustment of timestamp intervals [Boks87]. With these features, the proposed scheme can reduce resource and time waste, avoid unnecessary aborts, and hence produce transaction results in a timely manner. Before we provide a procedural description of the protocol, we briefly explain the idea of its components.

3. ELEMENTS OF THE PROPOSED PROTOCOL

3.1. Forward Validation

In real-time database systems, forward validation scheme is preferable to backward validation, because it provides flexibility for conflict resolution in that a transaction is validated against active transactions in their read phase instead of committed ones, i.e., either the validating transactions or the conflicting active transactions can be aborted to resolve conflicts. A number of conflict resolution policies that are based on forward validation have been proposed. Some of the examples are *priority abort, priority sacrifice,* and *priority wait* [Hari90b, Huan91], which we will explain in Section 7. Aborting active transactions in their read phase instead of validating transaction that has finished its execution to eliminate nonserializable executions means that conflicts are detected and resolved earlier than in backward validation. Thus, forward validation scheme reduces the waste of resource and time.

3.2. Conflict Classification

Since the forward validation test is conducted against active transactions, when a test is performed for a transaction, say T_i, active transactions in the system are classified into sets according to their execution history (with respect to that of T_i). First, the set of the active transactions are grouped into two sets; a *conflicting set*, that contains transactions in conflict with T_i, and a *nonconflicting set*, that contains transactions not in conflict with T_i. The conflicting set is further divided into two sets; a *Reconcilably Conflicting (RC) set* and an *Irreconcilably Conflicting (IC) set*. Transactions in the RC set are in conflict with T_i, but the conflicts are reconcilable, i.e., serializable. However, transactions in the IC set are in conflict with T_i, and the conflicts are irreconcilable, i.e., nonserializable. Details about each of the transaction sets will be given later in this paper. The RC transactions do not have to be aborted for serialization, but are required only to adjust their execution histories with respect to the validating transaction, T_i, using the timestamp interval facility of this protocol. However, the IC transactions should be handled with priority-based real-time conflict resolution schemes such as priority abort, priority sacrifice, and priority wait.

3.3. Dynamic Timestamp Assignment

Another important aspect of this protocol is that it combines timestamp ordering flavor with OCC using dynamic timestamp allocation. Most timestamp-based concurrency control protocols use a static timestamp allocation method, i.e., each transaction is assigned a timestamp value at its startup time, and a total ordering among transactions in the system is built up. The static timestamp allocation is based on the notion of *eager serialization*, which is considered harmful. First, the total ordering built by the eager serialization scheme does not reflect any actual conflict. Hence, it is possible that a transaction is aborted even when it requests its first data access [Baye82]. Second, the total ordering of all transactions is too restrictive, and degrades the degree of concurrency considerably. The dynamic timestamp allocation method is based on the notion of *lazy serialization* that builds only a partial ordering among transactions on demand to reflect actual execution history. This dynamic timestamp allocation scheme is possible in this protocol due to the OCC's phase-dependent structure of transaction execution, which allows the determination of the final serialization order to be delayed until the final phase of transaction execution. During the read phase, a transaction gradually builds its temporary serialization order with respect to committed transactions on demand whenever a conflict with such transactions occurs. Only when the transaction commits (after passing its validation test), is its permanent timestamp order, i.e., the final serialization order, determined.

3.4. Dynamic Adjustment of Serialization Order

The dynamic timestamp allocation scheme implementing lazy serialization is elaborated with a timestamp interval facility [Boks87] that keeps track of temporary serialization order and allows serialization order to be dynamically adjusted. In this scheme, a timestamp interval (initially, the entire range of the timestamp space) is assigned to each transaction instead of a single value timestamp. The timestamp intervals of active transactions preserve the partial ordering constructed by serializable execution. The timestamp interval of a transaction is adjusted (shrunk) whenever the transaction reads or writes a data object to preserve the serialization order induced by committed transactions. When the timestamp interval of a transaction shuts out, it means the transaction has been involved in a nonserializable execution, and the transaction should be restarted. Thus, this facility provides another means to detect and resolve nonserializable execution early in read phase.

When a transaction, T_i, commits after its validation phase, the timestamp intervals of those active transactions categorized as reconcilably conflicting with respect to T_i are adjusted, i.e., the serialization order between the validating transaction T_i and its RC transactions are determined. The permanent serialization order, i.e., final timestamp of these active transactions is not determined, but the partial ordering between T_i and these active transactions is determined by adjusting their timestamp intervals. Therefore these transactions do not have to be aborted even though they are in conflict with the committed transaction, i.e., unnecessary aborts are avoided.

4. PROCEDURAL DESCRIPTION

In this section, we provide a more detailed procedural description of the proposed protocol. To execute the proposed protocol, the system maintains a *transaction table* and an *object table*. The transaction table maintains the following information on each ongoing transaction:

- *RS(i)*: read set of transaction T_i;
- *WS(i)*: write set of transaction T_i; and
- *TI(i)*: timestamp interval of transaction T_i.

The object table keeps a read timestamp and a write timestamp of each data object in the database, that are determined as follows;

- *RTS(p)*: the largest timestamp of committed transactions that have read data object D_p; and
- *WTS(p)*: the largest timestamp of committed transactions that have written data object D_p.

The timestamp interval assigned to each active transaction is used to record temporary serialization order induced during the read phase of the transaction. In addition to the timestamp interval, a final timestamp, denoted as *TS(i)*, is assigned to each transaction, say T_i, when it has successfully passed its validation test, and is guaranteed to commit. The final timestamps of committed transactions are not kept in the transaction table. However, the final timestamp of a committed transaction is used to update the read and write timestamps of the data objects it has accessed, that are recorded in the object table.

Figure 1 shows a procedural description of the read, validation and write phase of a transaction, say T_i, executed with the proposed protocol. At the start of the execution of a transaction, say T_i, its timestamp interval *TI(i)* is initialized as

[0, ∞), i.e., the entire range of timestamp space. For each read or write operation made by T_i, $TI(i)$ is adjusted to represent the dependencies, i.e., serialization order induced by the operation. The adjustment of $TI(i)$ preserves the order induced by the timestamps of all committed transactions which have accessed that data object.

This adjustment is accomplished using different set intersection operations for read and write operations. When T_i reads a data object, the order of the read operation is adjusted to place after all the write operations made by committed transactions. Thus, after the read operation, $TI(i)$ is adjusted to include only the intersection portion of the current $TI(i)$ and the timestamp space following the write timestamp of the data object, which is by definition the largest timestamp among the committed transaction that have written the data object. When T_i writes a data object, the order of the write operation is adjusted to place after all the read as well as write operations made by committed transactions. Thus $TI(i)$ is updated to include only the intersection of the current $TI(i)$ and the timestamp space determined by the read as well as write timestamp of the data object. In the given procedure, we assume timestamp intervals contain only integers. In the read phase, any operation of an active transaction, T_i, which

Read Phase

```
for every D_p in RS(i) do
    read(D_p);
    TI(i) := TI(i) ∩ [WTS(p) + 1, ∞);
    if TI(i) = ∅
    then restart(i);
    endif;
enddo;

for every D_q in WS(i) do
    write(D_q);
    TI(i) := TI(i) ∩ [WTS(q) + 1, ∞)
                   ∩ [RTS(q) + 1, ∞);
    if TI(i) = ∅
    then restart(i);
    endif;
enddo;
```

Validation and Write Phase

```
determine RC_set(i) and IC_set(i);
if IC_set(i) ≠ ∅
then conflict_resolution(IC_set(i));
endif;
if not ABORTED(i)
then select TS(i) from TI(i);
     update RTS(p) for every D_p in RS(i);
     update WTS(q) for every D_q in WS(i);
     adjustment(RC_set(i));
     execute write_phase(i);
endif;
```

Figure 1 The Proposed Protocol

introduces a nonserializable execution results in $TI(i)$ to be ∅. A transaction T_i must be restarted if $TI(i)$ becomes ∅.

When a transaction T_i finishes the read phase and enters the validation phase, first, its RC set, i.e., the set of active transactions reconcilably conflicting with T_i, and its IC set, i.e., set of active transactions irreconcilably conflicting with T_i are determined. To determine these sets, the protocol uses the read and write sets of active validating transaction and active transactions, recorded in the transaction table. The categorization procedure will be discussed in detail in the next section. If there is one or more IC transactions in the system, the protocol invokes a priority-based conflict resolution scheme such as priority abort, priority sacrifice, and priority wait, to resolve the conflict between T_i and the irreconcilably conflicting transactions. Depending on the priorities of the conflicting transactions and the chosen conflict resolution scheme, either the validating transaction or the conflicting active transactions are aborted, as we will explain in Section 7.

If T_i has not been aborted during the priority-based conflict resolution (if any), then it can be committed now. The committing process consists of the following actions. First, the final timestamp, $TS(i)$, of T_i is determined so that the order induced by the final timestamp should not destroy the serialization order constructed by the already committed transactions. In fact, any timestamp in the range of $TI(i)$ satisfies this condition, because $TI(i)$ preserves the order induced by all committed transactions. Thus, any timestamp from $TI(i)$ can be chosen for the final timestamp. Second, the read and write timestamps of the data objects T_i has accessed should be updated, if necessary, to reflect the final timestamp of T_i. Finally, the timestamp intervals of all the RC transactions should be adjusted to reflect the final serialization order between T_i and them. This adjustment of the timestamp intervals of RC transactions is similar to the timestamp interval adjustment of an active transaction in the read phase in that it is done with set intersection operations and is based on the same reasons to determine serialization order. This adjustment procedure will be given in detail in Section 6.

One important point to note related with the validation process is that the final timestamp $TS(i)$ from timestamp interval $TI(i)$ can be chosen in favor of high priority transactions, though any timestamp from $TI(i)$ is eligible. When choosing the final timestamp for a committing transaction, the protocol can check the priority of its reconcilably conflicting transactions, and decide the timestamp in such a way that higher priority transactions are left with larger timestamp intervals. Because a larger timestamp interval means less possibility of restarting the transaction in some sense, a transaction with higher priority needs to have a larger timestamp interval than a transaction with lower priority. Another point to note here is that the final timestamp itself does not have to be a specific number. The entire timestamp interval can be the final timestamp of a committed transaction. Accordingly, read and write timestamps of data objects can also be intervals of timestamps, instead of being a specific value. This extension of the final timestamp may provide more flexibility for adjustment of serialization order, and is to be examined in our experiment.

5. CONFLICT CLASSIFICATION

Suppose T_i is a committing transaction and T_j ($j = 1, 2, \ldots,$

$n, j \neq i$) are transactions in their read phase. Then, the occurrence of conflicts between T_i and T_j, and the types of the conflicts are detected by looking at intersections of read sets and write sets of T_i and T_j as follows:

(c_1) $RS(j) \cap WS(i) \neq \emptyset$ $\Rightarrow$ read-write conflict;
(c_2) $WS(j) \cap WS(i) \neq \emptyset$ $\Rightarrow$ write-write conflict;
(c_3) $WS(j) \cap RS(i) \neq \emptyset$ $\Rightarrow$ write-read conflict.

Before discussing the categorization of conflicting transactions, we first explain how to identify the state of an active transaction using these three conditions. For notational convenience, we introduce a simple notation as follows. We use a triple (c_1, c_2, c_3) to express the state of an active transaction. Each element of the triple indicates whether the corresponding conflict condition is satisfied or not. When a conflict condition, c_i, is satisfied, it is denoted simply as c_i. However, if a condition, c_i, is not satisfied by the transaction, it is denoted as $\sim c_i$. For example, if a transaction satisfies c_1 and c_3, but does not satisfy c_2, i.e., it has both read-write and write-read conflicts with the validating transaction, but no write-write conflict, then it is denoted as $(c_1, \sim c_2, c_3)$.

Using this notation and Venn diagrams, Figure 2 shows the six possible states of active transactions under the assumption that the write set of a transaction is a subset of its read set. There are two Venn diagrams of the triple $(\sim c_1, \sim c_2, \sim c_3)$ in (a) and (b). They indicate (a) no access to common data objects at all and (b) only read operations on common data objects, respectively. Both cases do not produce any conflict between T_j and T_i. The triple $(c_1, \sim c_2, \sim c_3)$ in (c) means that there is only read-write conflict, but neither write-write nor write-read conflict between T_j and T_i. The triple $(\sim c_1, \sim c_2, c_3)$ in (d) indicates the case when there is only write-read conflict between T_j and T_i. The triple $(c_1, \sim c_2, c_3)$ in (e) indicates the case when there are read-write conflict and write-read conflict, but no write-write conflict between T_j and T_i. Finally, the triple (c_1, c_2, c_3) in (f) means that between T_j and T_i, all the three possible conflicts, i.e., read-write, write-write, and write-read conflicts exist. Besides these combinations of conflict conditions, there are three more combinations, $(\sim c_1, c_2, \sim c_3)$, $(c_1, c_2, \sim c_3)$, and $(\sim c_1, c_2, c_3)$ which are unable to occur under the given assumption that the write set of a transaction is a subset of its read set, and hence they are not included in the figure.

We can categorize the active transactions according to their states. First, the transactions are classified into two groups;

- nonconflicting transactions, and
- conflicting transactions.

Obviously, transactions whose state satisfies the conflict conditions, $(\sim c_1, \sim c_2, \sim c_3)$ belong to the first set. The latter contains the rest of the conditions, $(c_1, \sim c_2, \sim c_3)$, $(\sim c_1, \sim c_2, c_3)$, $(c_1, \sim c_2, c_3)$ and (c_1, c_2, c_3).

Second, the set of conflicting transactions are further divided into two group;

- Reconcilably Conflicting transactions (*RC*), and
- Irreconcilably Conflicting transactions (*IC*).

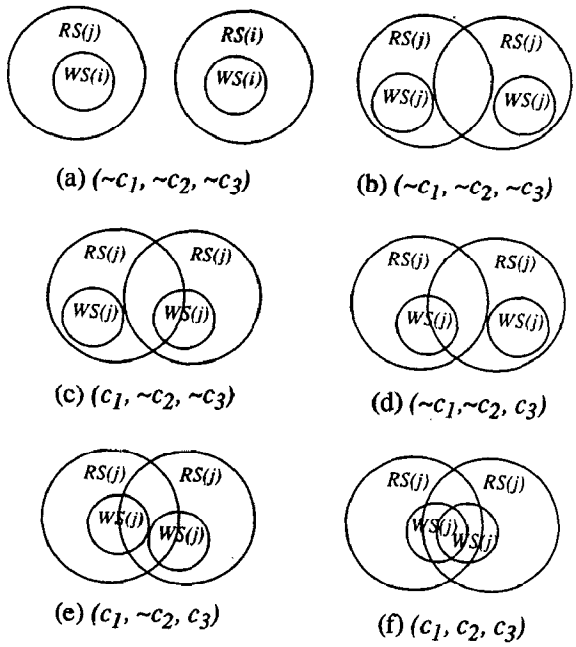

Figure 2 States of Active Transactions

Reconcilably conflicting transactions are ones which have only conflicts that can be serialized without aborting using the dynamic serialization order adjustment mechanism in this protocol, while irreconcilably conflicting transactions are ones that are involved in nonserializable execution which leads to aborting of either the validating transaction or active IC transactions.

6. SERIALIZATION ORDER ADJUSTMENT

To understand how to distinguish RC transactions and IC transactions, let us first consider the ways to resolve each conflict type by adjustment of serialization order.

(1) $RS(j) \cap WS(i) \neq \emptyset$ (read-write conflict)
The read-write conflict between the committing transaction, T_i, and the active transaction, T_j, can be serialized by the following timestamp interval adjustment of T_j;

```
TI(j) := TI(j) ∩ [0, TS(i) - 1].
```

This adjustment of *TI(j)* makes a partial ordering between T_i and T_j as $T_j \rightarrow T_i$, and is called *backward adjustment*. The meaning of this adjustment operation is that the read of T_j precedes the write of T_i on the same data object, i.e., the data read by T_j have not been written by T_i.

(2) $WS(j) \cap WS(i) \neq \emptyset$ (write-write conflict)
The write-write conflict between T_i and T_j can be serialized by the following operation:

```
TI(j) := TI(j) ∩ [TS(i) + 1, ∞).
```

This adjustment operation makes a partial ordering between T_i and T_j as $T_i \rightarrow T_j$, and is called *forward adjustment*. This operation implies that the write of T_i precedes the write of T_j on the same data object, i.e., the write of T_j is not overwritten by T_i.

(3) $WS(j) \cap WS(i) \neq \emptyset$ (write-read conflict)

The write-read conflict between T_i and T_j can be serialized by

$$TI(j) := TI(j) \cap [TS(i) + 1, \infty).$$

Similarly, this adjustment of $TI(j)$ is a forward adjustment, and makes a partial ordering $T_i \rightarrow T_j$. The implication of this operation is that the data object read by T_i has not been written T_j.

From this discussion of dynamic adjustment of serialization order method, we can define a reconcilably conflicting transactions as ones all of whose conflicts with the validating transaction can be serialized by either forward adjustments only or backward adjustments only, but not both. That is, transactions whose state is either $(c_1, \sim c_2, \sim c_3)$ or $(\sim c_1, \sim c_2, c_3)$ among other possible states, belong to the set of RC transactions.

Now we give a pseudo code for the adjustment procedure. Suppose T_i is the validating transaction, and T_j ($j = 1, 2, ..., n, j \neq i$) are reconcilably conflicting transactions.

```
adjustment(RC_set(i));
{
  for every Dp in RS(i) do
    for every Tj writing Dp do
      TI(j) := TI(j) ∩ [TS(i) + 1, ∞);
      if TI(j) = ∅
      then restart(j);
      endif;
    enddo;
  enddo;

  for every Dq in WS(i) do
    for every Tj reading Dq do
      TI(j) := TI(j) ∩ [0, TS(i) - 1];
      if TI(j) = ∅
      then restart(j);
      endif;
    enddo;

    for every Tj writing Dq do
      TI(j) := TI(j) ∩ [TS(i) + 1, ∞ );
      if TI(j) = ∅
      then restart(j);
      endif;
    enddo;
  enddo;
}
```

Figure 3 Serialization Order Adjustment

7. PRIORITY-BASED CONFLICT RESOLUTION

Finally, we are left only with the set of IC transactions. Transactions involved in conflicts that cannot be serialized by either forward adjustment only or backward adjustment only belong to IC transaction set. Obviously, a need of both forward and backward adjustment of timestamp interval for serialization results in the timestamp interval shut out. The transactions belonging to IC set are involved in nonserializable execution with the validating transaction. Transactions whose state is either $(c_1, \sim c_2, c_3)$ or (c_1, c_2, c_3) among other possible states, belong to the set of IC transactions.

As we mentioned in the previous section, we use priority-based conflict resolution schemes to resolve these irreconcilable conflicts. For the priority-based conflict resolution, it is useful to divide the IC set into two groups;

- Conflicting transactions with Higher Priority (*CHP*), and
- Conflicting transactions with Lower Priority (*CLP*).

CHP group contains transactions that belong to the IC set and have higher priorities than the validating transaction. In contrast, CLP group contains transactions that are in the IC set and have lower priorities than the validating transaction.

Now we summarize priority-based conflict resolution schemes proposed in [Hari90b, Huan91]. Suppose T_i is the validating transaction, and T_j ($j = 1, 2, ..., n, j \neq i$) are irreconcilably conflicting transactions. Also, suppose l is the number of CHP transactions, and m is the number of CLP transactions ($n = l + m$).

(1) Priority abort

In this scheme [Huan91], when a transaction reaches its validation phase, it is aborted if its priority is less than that of all the conflicting transactions, i.e., all the IC transactions are with higher priority; otherwise, it commits and all the conflicting transactions are restarted. It can be described in a pseudo code as follows:

```
if (m = 0)
then restart(i);
else restart(j), j = 1, 2, ..., n;
endif;
```

(2) Priority sacrifice

In this scheme [Hari90b], when a transaction reaches its validation phase, it is aborted if there exist one or more CHP transactions; otherwise it commits and all the irreconcilably conflicting transactions are restarted. Its pseudo code is given as follows:

```
if (l ≠ 0)
then restart(i);
else restart(j), j = 1, 2, ..., n;
endif;
```

(3) Priority wait

In this scheme [Hari90b], when a transaction reaches its validation phase, if there exist one or more CHP transactions, it waits for CHP transactions to complete. Its pseudo code is given as follows:

```
while (l ≠ 0) do
  wait;
endwhile;
restart(j), j = 1, 2, ..., n;
```

(4) Wait-50

In this scheme [Hari90b], a validating transaction is made to wait as long as more than half the IC transactions have higher priorities; otherwise it commits and all the IC transactions are restarted. The basic idea of this scheme is to maximize the beneficial effects of blocking, while reducing the effects of its drawbacks. It can be described in a pseudo code as follows:

```
while ((1 ≠ 0) and (1 > m)) do
  wait;
endwhile;
restart(j), j = 1, 2,..., n;
```

8. CONCLUDING REMARKS

In this paper, we have presented a new concurrency control protocol based on OCC. The goal of the protocol is to remedy the problem of wasted resource of conventional OCC protocols, and the problem of unnecessary aborts caused by incomplete validation tests. To achieve this goal, our scheme depends on early detection and resolution of conflicts using forward validation and timestamps, classification on conflicting transactions according to the types of conflicts and their priorities, and dynamic adjustment of serialization order using timestamp intervals.

Although we have argued that the proposed scheme has properties to produce results in a timely manner as well as to maintain data consistency, we need more concrete work on the performance of the scheme to support the argument. Through a simulation study using a fairly complete model of real-time database system [Lee92], we are currently evaluating the proposed scheme both in a deadline-driven scheduling environment and in a value-based scheduling environment. One of our goals in this performance evaluation study is to investigate if the complexity of the proposed scheme resulted from the dynamic management of timestamp interval is cost effective.

Also, we are considering several ways to improve the performance of the proposed protocol. In Section 4, without providing details, we have argued that the proposed scheme can be further optimized by using some strategies for choosing final timestamps of transactions, and using timestamp intervals for final timestamps. This argument will be verified by performing sensitivity tests in our simulation system. If the result shows that performance of the scheme is sensitive to those strategies, then we will develope appropriate mechanisms.

Finally, we will extend the proposed protocol so that it can be used in distributed real-time database systems. One advantage of OCC over 2PL is that the performance of OCC protocols does not suffer a significant degradation in distributed systems unlike 2PL protocols. Moreover, distribution of data is not only essential feature, but indeed is the key to the success of most practical real-time systems [Rodd89]. The proposed concurrency control scheme has a potential for good performance in a distributed real-time database systems.

Reference

[Abbo88] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation", *Proceedings of the 14th VLDB Conference*, August 1988.

[Abbo89] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data", *Proceedings of the 15th VLDB Conference*, August 1989.

[Baye82] Bayer, R., K. Elhardt, J. Heigert and A. Reiser, "Dynamic Timestamp Allocation for Transactions in Database Systems", *Proceedings of the 2nd Int. Symposium on Distributed Data Bases*, September 1982, pp 9-20.

[Bern87] Bernstein, P. A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987.

[Boks87] Boksenbaum, C., M. Cart, J. Ferrie, and J. Pons, "Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 4, April 1987.

[Buch89] Buchmann, A., D. R. McCarthy, M. Hsu, and U. Dayal, "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control", *Proceedings of the 5th Data Engineering Conference*, February 1989.

[Care87] Carey, M. J., "Improving the Performance of an Optimistic Concurrency Control Algorithm Through Timestamps and Versions", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 6, June 1987, pp. 746 - 751.

[Haer84] Haerder, T., "Observations on Optimistic Concurrency Control Schemes", *Information Systems*, 9(2), 1984.

[Hari90] Haritsa, J. R., M. J. Carey, and M. Livny, "On Being Optimistic about Real-Time Constraints", *Proceedings of the 1990 ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems (PODS)*, 1990.

[Hari90b] Haritsa, J. R., M. J. Carey, and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control", *Proceedings of the IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[Huan89] Huang, J., J. A. Stankovic, D. Towsley and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing", *Proceedings of the IEEE Real-Time Systems Symposium*, December 1989.

[Huan91] Huang, J., J. A. Stankovic, K. Ramamritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes", *Proceedings of the Conference on Very Large Data Bases*, September 1991.

[Kung81] Kung H. T., and J. T. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, 6(2): 213-226, June 1981.

[Lee92] Lee, J., "Scheduling Transactions in Real-Time Database Systems", *Ph.D. Dissertation Proposal*, Department of Computer Science, University of Virginia, September, 1992.

[Lin89] Lin, K., "Consistency Issues in Real-Time Database Systems", *Proceedings of the 22nd Hawaii International Conference on System Science*, January 1989.

[Lin90] Lin Y., and S. H. Son, "Concurrency Control in Real-Time Database Systems by Dynamic Adjustment of Serialization Order," *Proceedings of the 11th IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[Liu88] Liu, J., K. Lin, and X. Song, "Scheduling Hard Real-Time Transactions", *The 5th IEEE Workshop on Real-Time Operating Systems and Software*, May 1988.

[Robi82] Robinson, J. T., "Design of Concurrency Controls for Transaction Processing Systems", *Ph.D. Thesis*, Technical Report CMU-CS-82-114, Carnegie-Mellon University, 1982.

[Rodd89] Rodd, M. G., "Real-Time Issues in Distributed Data Bases for Real-Time Control", *IFAC Distributed Databases in Real-Time Control*, Budapest, Hungary, 1989, pp. 1-7.

[Sha88] Sha, L., R. Rajkumar and J. P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases", *ACM SIGMOD Record*, March 1988.

[Son88] Son, S. H., "Semantic Information and Consistency in Distributed Real-Time Systems", *Information and Software Technology*, Vol. 30, September 1988.

[Stan88] Stankovic, J. A., and W. Zhao, "On Real-Time Transactions", *ACM SIGMOD Record*, March 1988.

[Stan90] Stankovic, J. A., and K. Ramamritham, "What is Predictability for Real-Time System?", *Real-Time Systems*, 2: 247-254, Kluwer Academic Publishers, 1990.