OCC and Its Variants

Jan Lindström

Helsinki 7. November 1997

Seminar on real-time systems

UNIVERSITY OF HELSINKI

Department of Computer Science

Contents

1 Introduction

Real-time database systems (RTDBS) are database systems which must process transactions within definite time bounds usually defined in the form of a deadline. Failure to complete transactions before their deadlines greatly degrades the usefulness of the transactions. The applications of RTDBS can be found in programs trading in the stock market, computer integrated manufacturing, military command and control management, and network management systems.

In RTDBS, the primary performance criterion is timeliness level and not average response time or throughput. Scheduling of transactions is driven by priority considerations rather than fairness considerations. Performance goals of RTDBSs is different from those for conventional database systems. Therefore RTDBSs require new transaction management methods to satisfy the performance goals.

Considerable research has recently been devoted to designing concurrency control algorithms for RTDBSs and to evaluating their performance. Most of these studies use the *serializability* [EGLT76] for maintaining data consistency. Because at present there is no general-purpose consistency criterion available that is less stringent than serializability. New consistency criterion should be as obviously correct and easily implementable as serializability.

Most of the proposed algorithms are based on one of the two basic mechanisms: *two-phase locking* (2PL) [EGLT76] and *optimistic protocols* (OPT) [KR81]. Conventional concurrency control protocols are often unsatisfactory for real-time databases, because they do not observe transaction priorities. In this paper we concertrate on optimistic protocols.

The remainder of this paper is organized in the following fashion: Section 2 reviews principles of optimistic concurrency control. In Section 3, we present classical optimistic protocols and two new OCC protocols in detail. Finally, Section 3 summarizes the main conclusions of this study.

2 Optimistic Concurrency Control

In *Optimistic Concurrency Control* (OCC) [KR81], transactions are allowed to execute unhindered until they reach their commit point, at which time they are validated. The execution of a transaction consists of three phases, *read phase*, *validation phase*, and *write phase*.

The read phase is the normal execution of the transaction. Write operations are performed on the private data copies in the local workspace of the transaction. This kind of operation is called *pre-write*. Identification of all read data items are stored in a *readset*. Identification of all pre-written data items are stored in a *writeset*. Readset and writeset are used to check conflicts against other active transactions.

The validation phase ensures that all the committed transactions have executed in a serializable fashion. Every validation scheme is based on the following principles to ensure serializability. If a transaction T_i is serialized before transaction T_j, the following two conditions must be satisfied:

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

Generally, condition 1 is automatically ensured in most optimistic algorithms because I/O operations in the write phase are required to be done sequentially in critical section. Thus most validation schemes consider only condition 2. During the write phase, all changes made by the transaction are permanently installed into the database. To design efficient real-time OCC protocol, three issues have to be considered:

1. which validation scheme should be used to detect data conflicts amongst transactions;
2. how to minimize the number of transaction restarts; and
3. how to select a transaction to restart when there are conflicts.

2.1 Backward Validation

In *Backward Validation* [Här84], the validating transaction is checked for conflicts against (recently) committed transactions. Data conflicts are detected by comparing the read set of the validating transaction and the write set of the committed transactions. If the validating transaction has a data conflict with any committed transactions, it will be restarted. The classical optimistic algorithm in [KR81] is based on this validation process.

Let T_v be the validating transaction and $T_c(c = 1, 2, ..., n, c \neq v)$ be the transactions recently committed with respect to T_v. Let $RS(T)$ and $WS(T)$ denote the read set and the write set of the transaction T, respectively. Backward Validation protocol is presented in figure 1.

```
validate(T_v)
{
    valid = true;
    for T_c  (c = 1, 2, ..., n)
    {
        if WS(T_c) ∩ RS(T_v) ≠ ∅   then
            valid = false;
        if not valid  then
            break;
    }

    if valid  then
        commit WS(T_v) to database;
    else
        restart(T_v);
}
```

Figure 1: Backward Validation algorithm.

2.2 Forward Validation

In *Forward Validation* [Här84], the validating transaction is checked for conflicts against other active transactions. Data conflicts are detected by comparing the write set of validating transaction and the read set of the active transactions. If active transaction has read an object that has been concurrently written by the validating transaction, the values of the object used by the transactions are not consistent. Such data conflict can be resolved by restarting either the validating transaction or the conflicting transactions in the read phase. Optimistic algorithms based on this validation process are studied in [Här84].

Let $T_a(a = 1, 2, ..., n, a \neq v)$ be the conflicting transactions in their read phase. Forward Validation protocol is presented in figure 2.

```
validate(T_v)
{
    valid = true;
    for T_a  (a = 1, 2, ...n)
    {

        if  RS(T_a) ∩ WS(T_v) ≠ ∅   then
            valid = false;
         if not valid  then
            break;
    }

    if  valid  then
        commit WS(T_v) to database;
    else
        conflict_resolution(T_v);
}
```

Figure 2: Forward Validation algorithm.

In the real-time database systems, data conflicts should be resolved in favor of the higher priority transactions. Forward Validation provides flexibility for conflict resolution. Either the validating transaction or the conflicting active transactions may be chosen to restart. Therefore it is preferable for the real-time database systems. In addition to this flexibility, Forward Validation has the advantage of early detection and resolution of data conflicts.

2.3 Unnecessary restarts

The major performance problem with OCC protocols is the heavy restart overheads, wasting large amount of resources. Forward Validation is based on the assumption that the serialization order of transactions is determined by the arriving order of transactions at validation phase. Thus the validating transaction, if not restarted, always precedes concurrently running active transactions in serialization order. Validation process based on previous assumption can incur restarts not necessary to ensure data consistency. These restarts should be avoided.

Let $r_i[x]$ and $w_i[x]$ denote a read and write operation, respectively, on the data object x by transaction T_i, and let v_i and c_i denote the validation and commit of transaction T_i, respectively. Consider transactions T_1 and T_2:

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

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

and suppose they execute as follows:

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

Based on the traditional OCC, T_2 has to be restarted. However, it is not necessary. Because if T_2 is allowed to commit such as:

$H_1 = r_2[x]w_2[y]r_1[x]w_1[x]v_1c_1\mathbf{v_2c_2}$,

the history, H_1, is equivalent to the serialization order of $T_2 \rightarrow T_1$. There is no cycle in the history H_1. Therefore execution is serializable $\Box$.

3 Optimistic Concurrency Control protocols

In recent years, the use of optimistic schemes for concurrency control in real-time optimistic protocols has received more and more attention. Different real-time optimistic protocols have been proposed. They incorporate different priority conflict resolution methods in the validation phase of a transaction. In this paper we briewly present few popular optimistic protocols and three optimistic protocols in detail.

OPT-BC [HCL90b] extends classical optimistic protocol with *Broadcast Commit* [MN82, Rob87] protocol. In Broadcast Commit protocol the transaction notifies other running transactions that conflict with it. The conflicting transactions are restarted.

OPT-SACRIFICE [HCL90a] is an optimistic protocol which uses a priority-driven abort for conflict resolution. When a transaction reaches its validation phase, algorithm checks for conflicts with currently executing transactions. If conflicts are detected and at least one of the transactions in the conflict set is a higher priority transaction, then the validating transaction is restarted. It is *sacrificed* in an effort to help the higher priority transactions make their deadlines.

OPT-WAIT [HCL90a] algorithm incorporates a *priority wait* mechanism. When transaction reaches its validation phase, if its priority is not the highest among the conflicting transactions, it waits for the conflicting transactions with higher priority to complete. This gives the higher priority transactions a change to make their deadlines first. If transaction finally commits after waiting for some time, it causes all its conflicting transactions with lower priority to be restarted. This problem becomes worse with the possibility of chained blocking, which may cause cascaded aborts.

3.1 WAIT-50

The WAIT-50 [HCL90a] algorithm is an extension of the OPT-WAIT. It incorporates a *wait control* mechanism. This mechanism monitors transaction conflict states and dynamically decides when, and for how long, a low priority transaction should be made to wait for its conflicting higher priority transactions.

In WAIT-50 scheme, a validating transaction is made to wait as long as more than half the transactions that conflict with it have higher priorities. Otherwise it commits and all the conflicting transactions are restarted. The basic idea of WAIT-50 scheme is to maximize the beneficial effects of blocking, while reducing the effects of its drawbacks.

A transaction's conflict state is assumed to be characterized by the index *HPpercent*. HPpercent is the percentage of the conflicting higher priority transactions. The validation algorithm is presented in figure 3.

```
validate(T_v)
{
    while conflicting higher priority transactions and HPpercent ≥ 50
        wait;

    for ∀ T_a ∈ conflict_set(T_v)
        restart(T_a);

    commit WS(T_v) to database;
}
```

Figure 3: Validation algorithm for WAIT-50.

Performance studies in [HCL90a] has shown WAIT-50 to provide significant performance gains over OPT-BC, OPT-SACRIFICE and OPT-WAIT.

3.2 OCC-TI

OCC-TI [LS93, Lee94] protocol resolves conflicts using time intervals of the transactions. Every transaction must be executed in a specific time slot. When an access conflict occurs, it is resolved using the read and write sets of the transaction together with the allocated time slot. Time slots are adjusted when a transaction commits.

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

In the read phase when a read operation is executed, write timestamp of the object accessed is verified against the time interval allocated to the transaction. If another transaction has written the object outside the time interval, the transaction must be restarted. In the read algorithm (figure 4) D_i is the object to be read, T_i is the transaction reading the object, $TI(T_i)$ is the time interval allocated to the transaction, $WTS(D_i)$ is the write timestamp of the object, and $RTS(D_i)$ is the read timestamp of the object.

```
read(T_i, D_i)
{
    TI(T_i) = TI(T_i) ∩ [WTS(D_i), ∞[ ;

    if TI(T_i) = [] then
        restart(T_i);

    read(D_i);
}
```

Figure 4: Read algorithm for OCC-TI.

In the read phase when a write operation is executed, the modification is made to local copy of the object. A *pre-write* operation is used to verify read and write timestamps of the written object against the time interval allocated to the transaction (figure 5). If another transaction has read or written the object outside the time interval, the transaction must be restarted.

```
pre-write(T_i, D_i)
{
    TI(T_i) = TI(T_i) ∩ [WTS(D_i), ∞[ ∩ [RTS(D_i), ∞[ ;

    if TI(T_i) = [] then
        restart(T_i);
}
```

$$
\begin{aligned}
&\texttt{pre-write}(T_i, D_i)\\
&\{\\
&\quad TI(T_i) = TI(T_i) \cap [WTS(D_i), \infty[\;\cap\; [RTS(D_i), \infty[\;;\\
&\quad \textbf{if}\ TI(T_i) = [] \ \textbf{then}\\
&\qquad \texttt{restart}(T_i);\\
&\}
\end{aligned}
$$

Figure 5: Pre-write algorithm for OCC-TI.

At the beginning of the validation (figure 6) the final timestamp of the validated transaction is determined from the timestamp interval allocated to the transaction. In this algorithm, we always select the minimum value of $TI(T_v)$ for $TS(T_v)$. The timestamp intervals of all concurrently running and conflicting transactions must be adjusted to reflect the serialization order. Any transaction which timestamp interval becomes empty must be restarted.

$$
\begin{aligned}
&\texttt{validate}(T_v)\\
&\{\\
&\quad \textbf{select}\ TS(T_v)\ \textbf{from}\ TI(T_v) \;;\\
&\quad \textbf{for}\ \forall\, T_a \in \mathit{active\_transactions}()\\
&\qquad \texttt{adjust}(T_v, T_a);\\
&\quad \textbf{for}\ \forall\, D_i \in RS(T_v)\\
&\qquad \texttt{update}(RTS(D_i));\\
&\quad \textbf{for}\ \forall\, D_i \in WS(T_v)\\
&\qquad \texttt{update}(WTS(D_i));\\
&\quad \texttt{commit}\ WS(T_v)\ \texttt{to database;}\\
&\}
\end{aligned}
$$

Figure 6: Validation algorithm for OCC-TI.

The adjustment of timestamp intervals of a running transaction iterates through the readset and writeset (figure 7). When access has been made to the same objects both in validating transaction and in the compared transaction, the time interval of the compared transaction is adjusted. If a compared transaction has made a conflicting operation in an adjusted time interval, the transaction must be restarted.

```
adjust(T_v, T_a)
{
    for ∀ D_i ∈ RS(T_v)
    {
        if D_i ∈ WS(T_a) then
            TI(T_a) = TI(T_a) ∩ [TS(T_v), ∞[ ;

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

    for ∀ D_i ∈ WS(T_v)
    {
        if D_i ∈ RS(T_a) then
            TI(T_a) = TI(T_a) ∩ [0, TS(T_v) − 1];

        if D_i ∈ WS(T_a) then
            TI(T_a) = TI(T_a) ∩ [TS(T_v), ∞[ ;

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

Figure 7: Adjust algorithm for OCC-TI.

Performance studies in [LS93] has shown that under the policy that discards tardy transactions from the system, the optimistic algorithms outperform 2PL-HP [AGM88]. OCC-TI does better than OPT-BC among the optimistic algorithms. The performance difference between OPT-BC and OCC-TI becomes large especially when the probability of a data object read being updated is low, which is true in most actual database systems.

3.3 OCC-DA

OCC-DA [LLH96] is based on the Forward Validation scheme. The number of transaction restarts is reduced by using dynamic adjustment of 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. No adjustment of the timestamps is necessary in case of data conflicts in the read phase. In OCC-DA serialization order of committed transactions may be different from their commit order.

Suppose we have a validating transaction T_v and a set of active transactions $T_j(j = 1, 2, ..., n, j \neq$

v). There are three possible types of data conflicts which can induce serialization order between T_v and T_j.

1. $RS(T_v) \cap WS(T_j) \neq \emptyset$ (write-read conflict)
   Write-read conflict between T_j and T_v 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 adjustment*.

2. $WS(T_v) \cap RS(T_j) \neq \emptyset$ (read-write conflict)
   Read-write conflict between T_j and T_v 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 adjustment*.

3. $WS(T_v) \cap WS(T_j) \neq \emptyset$ (write-write conflict)
   Write-write conflict between T_j and T_v can be resolved by adjusting the serialization order between T_v and T_j as $T_v \rightarrow T_j$ such that write of T_v cannot overwrite T_j's write. This type of serialization adjustment is called forward adjustment.

To support dynamic adjustment of serialization order in OCC-DA, a dynamic timestamp assignment method is used. For each transaction, T_i, there is a timestamp called *serialization order timestamp* $SOT(T_i)$ to indicate its serialization order relative to other transactions. Initially, the value of $SOT(T_i)$ is set to be ∞. If the value of $SOT(T_i)$ is other than ∞, it means that T_i has been backward adjusted before a committed transaction.

If T_i has been backward adjusted, $SOT(T_i)$ will also be used to detect whether T_i has accessed any invalid data item. This is done by comparing its timestamp with the timestamps of the committed transactions which have read or written the same data item. A data item in its read set and write set is invalidated if the data item has been updated by other committed transactions which have been defined after the transaction in the serialization order.

When T_v comes to validation, the sets of active transactions, T_i, whose serialization order timestamp, $SOT(T_v) \geq SOT(T_i)$ are collected to set $ATS(T_v)$. The set of active transactions, T_j, whose serialization order timestamp, $SOT(T_j) < SOT(T_v)$ are collected to set $BTS(T_v)$. In read phase $TR(T_v, D_p)$ is set to be $WTS(D_p)$ of read data item D_p.

The first part of the validation test is used only for those validating transactions which have been backward adjusted (figure 8). It is to check whether:

1. all the read operations of T_v have been read from the committed transactions T_c whose $SOT(T_c) < SOT(T_v)$, and
2. whether T_v's write is invalidated. This is done by comparing $SOT(T_v)$ with $WTS(D_p)$ and $RTS(D_p)$ of the data item D_p in T_v's write set or read set.

```
part_one(Tv)
{
    if   SOT(Tv) ≠ ∞   then
    {
        for   ∀ Dp ∈ RS(Tv)
        {
            if   TR(Tv, Dp) > SOT(Tv)   then
                 restart( Tv );
        }

        for   ∀ Dp ∈ WS(Tv)
        {
            if   SOT(Tv) < RTS(Dp)   or   SOT(Tv) < WTS(Dp)   then
                 restart( Tv );
        }
    }
}
```

Figure 8: First part of the validation algorithm for OCC-DA.

The purpose of part two of the test is to detect read-write conflicts between the active transactions and the validating transactions (figure 9). The write set of T_v is compared with the read sets of the active transactions T_j. The identity of the conflicting active transactions T_j are added to $BTlist(T_v)$ to indicate that T_j needs to be backward adjusted before T_v.

```
part_two(T_v)
{
    BTlist(T_v) = ∅ ;

    for ∀ T_j ∈ ATS(T_v)
    {
        for ∀ D_p ∈ WS(T_v)
        {
            if D_p ∈ RS(T_j) then
                BTlist(T_v) = BTlist(T_v) ∪ T_j ;
        }
    }
}
```

Figure 9: Second part of the validation algorithm for OCC-DA.

The third part of the test is to detect whether a backward-adjusted transaction T_j also needs forward adjustment with respect to T_v (figure 10). It compares the write set of T_j which is in $BTS(T_v)$ or in $BTlist(T_v)$ with the read set of T_v, and the write set of T_v with the write sets of T_j. If either one of them is not empty, T_j has serious conflict with T_v. In conflict resolution we select transactions for restart based on priorities.

```
part_tree(T_v)
{
    for ∀ T_j ∈ BTS(T_v) ∪ BTlist(T_v)
    {
        for ∀ D_p ∈ RS(T_v)
        {
            if D_p ∈ WS(T_j) then
                conflict_resolution(T_v,T_j);
        }

        for ∀ D_p ∈ WS(T_v)
        {
            if D_p ∈ WS(T_j) then
                conflict_resolution(T_v,T_j);
        }
    }
}
```

Figure 10: Third part of the validation algorithm for OCC-DA.

When the validating transaction reaches part four of the test, it is guaranteed to commit. The purpose is to assign a final commitment timestamp to the validating transaction and to update the necessary timestamps of the data items (figure 11).

```
part_four(T_v)
{
    if  SOT(T_v) = ∞  then
        SOT(T_v) = validation_time ;

    for ∀ T_j ∈ BTlist(T_v)
        SOT(T_j) = SOT(T_v) − ε ; //infinitesimal quantity

    for ∀ D_p ∈ RS(T_v)
        RTS(D_p) = SOT(T_v) ;

    for ∀ D_p ∈ WS(T_v)
        WTS(D_p) = SOT(T_v) ;

    commit WS(T_v) to database;
}
```

Figure 11: Fourth part of the validation algorithm for OCC-DA.

4 Conclusions

Concurrency control is one of the main issues in the studies of real-time database systems. With a strict consistency requirements defined by serializability, many real-time concurrency control schemes considered in literature are based on two-phase locking (2PL). This is not surprising since 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 in a real-time environment, transactions need to meet their time constraints as well as consistency requirements.

Therefore some alternatives to two-phase locking for real-time systems have been proposed and studied. Among them is a class of concurrency control schemes based on the well-know optimistic approach. Optimistic concurrency controll has the properties of non-blocking and deadlock freedom. These properties make the optimistic scheme especially attractive to real-time database system.

Optimistic protocols can be based on Backward Validation or Forward Validation. Most of propoced algorithms are based on Forward Validation. OPT-BC extend optimistic concurrency control protocol with Broadcast Commit. OPT-SACRIFICE uses priority driven abort conflict resolution. OPT-WAIT incorporates priority wait mechanism. WAIT-50 is extension of OPT-WAIT incorcorporating wait control mechanism. OCC-TI resolves conflicts using time intervals of transaction. OCC-DA reduces transaction restarts and resolves conflicts using dynamic timestamp assignment. From presented protocols OCC-DA is shown to be most powerful algorithm.

References

[AGM88] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In F. Bancilhon and D. J. DeWitt, editors, *Proceedings of the 14th VLDB Conference*, pages 1–12, San Mateo, Calif., 1988. Morgan Kaufmann.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, November 1976.

[Här84] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[HCL90a] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the 11th Real-Time Symposium*, pages 94–103, Los Alamitos, Calif., 1990. IEEE, IEEE Computer Society Press.

[HCL90b] J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 331–343. ACM, ACM Press, 1990.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transaction on Database Systems*, 6(2):213–226, June 1981.

[Lee94] J. Lee. *Concurrency Control Algorithms for Real-Time Database Systems*. PhD thesis, Faculty of the School of Engineering and Applied Science, University of Virginia, January 1994.

[LLH96] K. Lam, K. 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 1996. Springer.

[LS93] J. Lee and S. H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *Proceedings Real-Time Systems Symposium*, pages 66–75, Los Alamitps, Calif., 1993. IEEE, IEEE Computer Society Press.

[MN82] D. Menasce and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[Rob87] J. Robinson. *Design of Concurrency Controls for Transaction Processing Systems*. PhD thesis, Carnegie Mellon University, December 1987.