Multi-version Speculative Concurrency Control with Delayed Commit

Azer Bestavros
(best@cs.bu.edu)

Biao Wang
(bwang@cs.bu.edu)

Computer Science Department
Boston University
Boston, MA 02215

October 27, 1993

Abstract

This paper presents an algorithm which extends the relatively new notion of speculative concurrency control by delaying the commitment of transactions, thus allowing other conflicting transactions to continue execution and commit rather than restart. This algorithm propagates uncommitted data to other outstanding transactions thus allowing more speculative schedules to be considered. The algorithm is shown always to find a serializable schedule, and to avoid cascading aborts. Like speculative concurrency control, it considers strictly more schedules than traditional concurrency control algorithms. Further work is needed to determine which of these speculative methods performs better on actual transaction loads.

Keywords: Real-time systems, Time constraints, Database Management Systems, Concurrency control algorithms, Serializability.

1 Introduction

A real-time database management system is a combination of a conventional database management system and a real-time system. Like a database system, it has to process transactions and guarantee database consistency. Furthermore, it has to operate in real-time, satisfying time constraints on each transaction [Abbo92].

For example, consider a database system to monitor the stock market. It must contain data such as current stock prices, trading trends, and other economic indices. A process in the system that checks the state of this database and updates it with new information must satisfy certain timing constraints in order for the database to be accurate. For instance, buy and sell orders must meet timing constraints so that prices and quantities are constant throughout the transaction. Likewise, processes which read and analyze data to provide decision information for users, must complete before the data values in the database change again. Thus, having transactions produce results before their time constraints is as important as serial consistency.

Existing concurrency control algorithms for conventional database systems attempt to maximize concurrency (throughput), but ignore timing constraints. Deadline scheduling algorithms for conventional real-time systems do consider timing constraints, but ignore data consistency problems. Since concurrency control algorithms may introduce unpredictable delays due to transaction restarts and blocking, there is clearly a real need for a concurrency control model that combines the timeliness of deadline scheduling algorithms and the data consistency provided by conventional concurrency control algorithms.

The idea of redundant computation has been used in some real-time system applications. For example, the space shuttle control program redundantly runs some time critical computations on several computers simultaneously. The major advantage of doing this is that as long as one of the computations finishes successfully, the result is known. Thus several versions of the computation can be run, with different time bounds, guaranteeing an upper bound on the time to produce some result.

Mena [Mena82] has classified concurrency control algorithms into two classes: optimistic algorithms (*e.g.* [Hari90b, Hari90a, Huan90, Kim91, Lin90, Son92]) and pessimistic algorithms (*e.g.* [Abbo88, Stan88, Huan90, Sha91]). Bestavros has proposed (in [Best92]) a new approach, Speculative Concurrency Control (SCC), which incorporates redundant computation into concurrency control algorithms. Speculative concurrency control redundantly executes the same transaction, under different ordering assumptions. The algorithm then determines which ordering to chose, based on the real-time constraints.

Redundant transaction execution is better at meeting time and synchronization constraints because a variety of different synchronization strategies can be tried in parallel, increasing the possibility that one will meet the deadline. Furthermore, penalties incurred in one of these computations need not affect the others. By comparison, in single threaded implementations, a restart really must begin the entire transaction again with less time remaining before the deadline. Redundant

transaction execution has not been very widely used in today's commercial database applications because of the cost factor. Bestavros has argued in [Best92] that for many real-time systems this cost is justified by their critical nature. Furthermore, as processors and memory become cheaper, the availability of such resources is more realistic. Hence, redundant computation can be a very powerful technique in real-time DBMSs.

SCC combines the Pessimistic and Optimistic Concurrency Control algorithms by using redundant computation for the same transaction. One copy of the transaction runs under an optimistic concurrency control algorithm. Meanwhile, if a conflict that threatens the consistency of the database is detected, a redundant computation is started as early as possible creating an alternate schedule. The alternate computation is blocked at the point which caused the conflict. The alternate schedule is adopted *only if* the suspected inconsistency materializes; otherwise, it is abandoned. Even though it may do redundant computations, SCC will always start computation for a transaction at the earliest time a potential conflict is detected, by creating multiple copies of conflicting transactions, rather than waiting until the conflict materializes.

In this paper, we propose a concurrency control model, called Multi-version Speculative Concurrency Control with Delayed Commit (MSCC-DC), which is based on the SCC scheme, but which combines it with other ideas that have been studied for real-time DBMSs. One typical way transactions conflict with each other is if one transaction writes some data, and a second transaction then attempts to read that data. This creates potential conflicts since there are two values of the data, one that previously existed in the database, and another one that was written by the first transaction. Under SCC, both transactions will start out running using the Optimistic Concurrency Control algorithm. When the potential conflict is detected, a copy of the second transaction will be started and will continue until the conflict point (the attempt to read the data) where it will be blocked. In MSCC-DC, instead of blocking the alternate schedule at the conflict point, we allow the alternate schedule to continue and read the data value written by the first transaction. Since at this stage the first transaction has not yet committed, we say the second transaction has read uncommitted data. In general, presently used concurrency control schemes do not consider allowing transactions to read uncommitted data since it could easily cause cascading aborts. However, by limiting the length of a chain of transactions which read uncommitted (dirty) data, we can bound the number of aborts caused by a materialized conflict.

Moreover, the objective of a real-time DBMS is to minimize the percentage of transactions which miss their deadlines. If a transaction T commits immediately after it finishes its computation, it will cause all the other transactions that conflict with it to abort, and if most of the aborted transactions do not conflict with each other, a better percentage of deadlines may be met by committing the other transactions instead. Thus delaying the commit of a transaction T would give us more time to determine which is the better combination of transactions to commit. Meanwhile, since the data written by transaction T is made available to other transactions, redundant computation for active transactions can be started as early as possible.

The remainder of this paper is organized as follows. In section 2, we review some of the previous work done in concurrency control for real-time DBMSs and provide motivation for our

research direction. In section 3, we present a basic overview of our concurrency control algorithm. In section 4, we prove that MSCC-DC maintains database consistency, avoids unbounded cascading aborts, and show some analysis of the algorithm. Finally, in section 5, we conclude this paper, and describe our future research.

2 Previous Work

In a conventional DBMS, Agrawal has concluded in [Agra87] that pessimistic locking protocols, due to their conservation of resources, perform better than optimistic techniques. Pessimistic two-phase locking algorithms detect potential conflicts as they occur. However, they may suffer possible unbounded waiting due to blocking.

Consider the example below:

Transaction 1	*Transaction 2*
Read(a)	
ReadLock(a)	
	Read(b)
	ReadLock(b)
Write(b)	
	Write(a)

At this stage, transaction 1 is blocked waiting for transaction 2 to release the *ReadLock* on b so that it can get the *WriteLock* on b. On the other hand, transaction 2 is blocked waiting for transaction 1 to release the *ReadLock* on a so that it can get the *WriteLock* on b. None of the transaction can proceed (deadlock). Furthermore, the resource conservation nature of pessimistic algorithms becomes a draw back in the real-time environment where meeting the time-constraint has a much higher priority than saving resources.

In [Hari90b, Hari90a], Haritsa, Carey and Linvy showed that for a real-time DBMS with firm deadlines (transactions which misses the deadlines are immediately discarded), optimistic algorithms outperforms the pessimistic schemes. The key result is that, if low resource utilization is acceptable (*i.e.* a large amount of wasted resources can be tolerated), then computing resources wasted due to restart do not adversely affect performance. Thus, optimistic restart-based concurrency control algorithms that allow a higher degree of concurrency become more attractive in real-time DBMS over pessimistic blocking-based algorithms.

Classical Optimistic Concurrency Control algorithm [Kung81] consists of three stages of execution for a transaction: *read, validation, and write*. The key stage in the Optimistic Concurrency Control scheme is the validation phase where the fate of the transaction is determined. A transaction is allowed to execute unhindered (during its read stage) until it reaches its commit point, at which time a validation test is applied. This test checks if there is any conflict between the actions

of the transaction being validated and those of any other committed transactions. A transaction is restarted if it fails its validation test, otherwise it commits by going through its write stage, in which modifications to the database (updates or writes performed by the transaction during its read stage) are made visible to other transactions.

One serious problem with optimistic schemes is that conflict resolution is always done by aborting the transaction that is being validated. However, conflicts are not detected until the validation phase, at which time it may be too late to restart. The Broadcast Commit variant (OCC-BC) [Mena82, Robi82] of the classical Optimistic Concurrency Control scheme partially remedies this problem. When a transaction commits, it notifies those concurrently running transactions which conflict with it. Those transactions are restarted immediately. Note that there is no need to check for conflicts with already committed transactions since such transactions would have, in the event of a conflict, informed the validating transaction to restart. Thus, the validating transaction is always guaranteed to commit. The broadcast commit method detects conflicts earlier than the classical OCC algorithm resulting in earlier restart.

SCC combines the advantage of both optimistic and pessimistic schemes while avoiding their disadvantages [Best92]. It goes one step further in utilizing information about conflicts. Instead of waiting for a potential consistency threat to materialize and then taking a corrective measure, SCC uses redundant resources to start *speculating* on corrective measures as soon as conflict in question develops. By starting on such corrective measures as early as possible, the likelihood of meeting any set timing constraint is greatly enhanced.

To better illustrate the point, we use the following example. Assume there are two transactions T_1, and T_2, which (among others) perform some conflicting actions. In particular, T_2 reads item x after T_1 has updated it. The basic optimistic algorithm restart transaction T_2 when it enters its validation stage, and because it conflicts with the already committed transaction T_1 on data x. The scenario is illustrated in figure 1. Obviously, the likelihood of the restarted transaction T_2 meeting its timing constraint decreases.

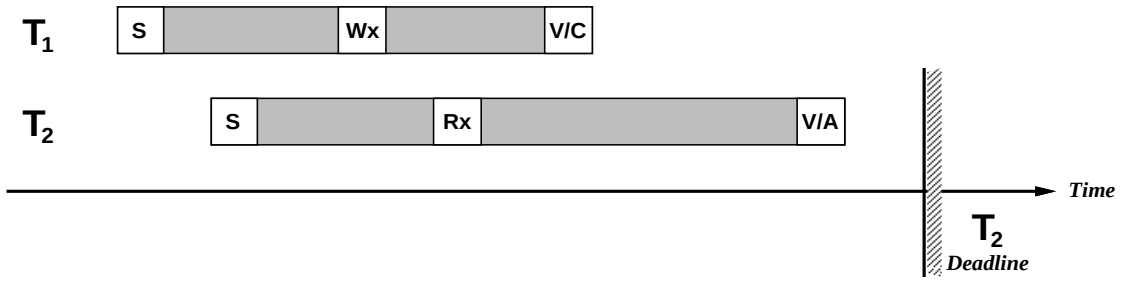

Figure 1: Transaction management under the basic OCC algorithm.

The OCC-BC algorithm avoids waiting unnecessarily until a transaction's validation stage in order to restart it. In particular, a transaction is aborted if any of its conflicts with other transactions in the system become a materialized consistency threat, *i.e.* one of the other transaction commits. This is illustrated in figure 2.

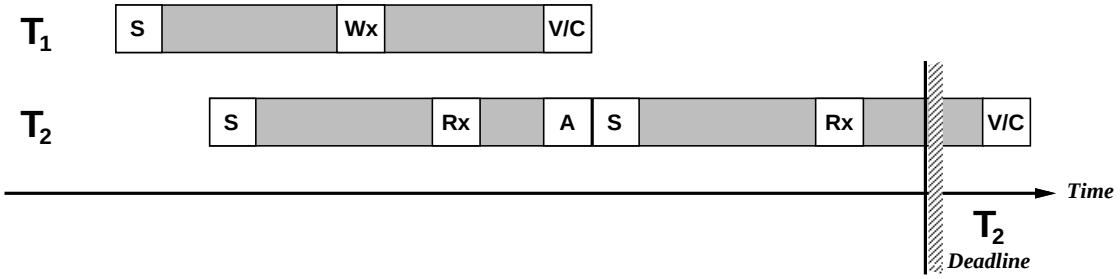

Figure 2: Transaction management under the basic OCC-BC algorithm.

With the SCC approach, at the time when transaction T_2 requests to read data item x, all the information necessary to conclude that there is a conflict (hence a potential consistency threat) between transaction T_2 and T_1 (which previously updated data item x) is available. Instead of pessimistically blocking T_2, or optimistically ignoring the conflict until the validation stage, SCC makes a copy , or *shadow*, of the reader transaction T_2. The original transaction T_2 continues to run uninterrupted, while the shadow transaction T_2' is restarted on a different processor and allowed to run concurrently using a pessimistic (locking) algorithms. Both versions of the same transaction are allowed to run in parallel, each one being at a different point of its execution. Only one of the two transactions will be allowed to commit; the other will be aborted. Figure 3 and figure 4 show two possible scenarios that may develop depending on the time needed for transaction T_2 to reach its validation stage. In figure 3, T_2 reaches its validation stage before T_1. T_2 is validated[1] and committed without any need to disturb T_1. The shadow transaction T_2' is aborted.

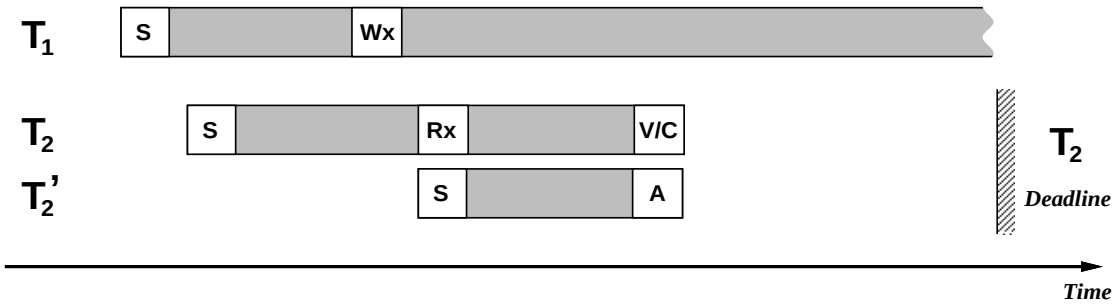

Figure 3: SCC schedule with an undeveloped potential conflict.

If transaction T_1 reaches its validation phase first, then transaction T_2 cannot continue to execute due to the (now visible) conflict over x. T_2 is aborted. The shadow transaction T_2' is adopted. Compare figure 4 to 2, we can see that SCC gains an earlier restart over OCC-BC.

One more problem with OCC-BC and other common concurrency control schemes is that by committing a transaction as soon as it finishes validating, it may cause a larger number of

[1] since T_2's write-set does not intersect T_1's read-set.

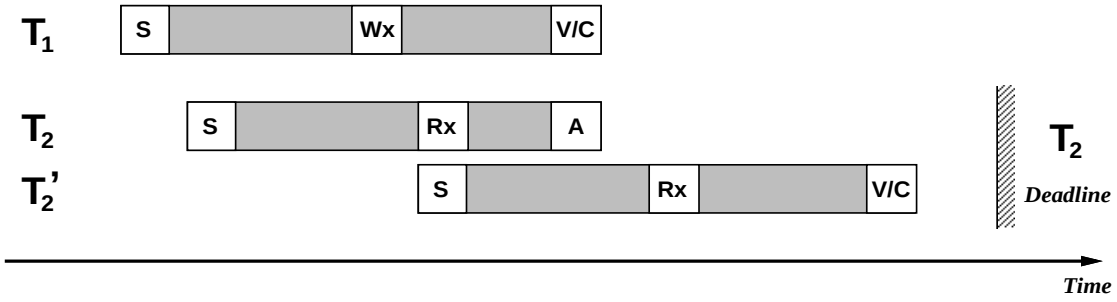

Figure 4: SCC schedule with a developed potential conflict.

transactions to abort and miss their deadlines. For example, in figure 5, committing T_1 as soon as it finishes validating causes both T_2 and T_3 to abort, and both of them cannot be restarted early enough to meet their deadlines. In [Hari90b], Harista showed that by making a lower priority transaction wait after it finishes validating, the numbers of transaction restarts are reduced, thus increase the number of the transactions meeting their deadlines.

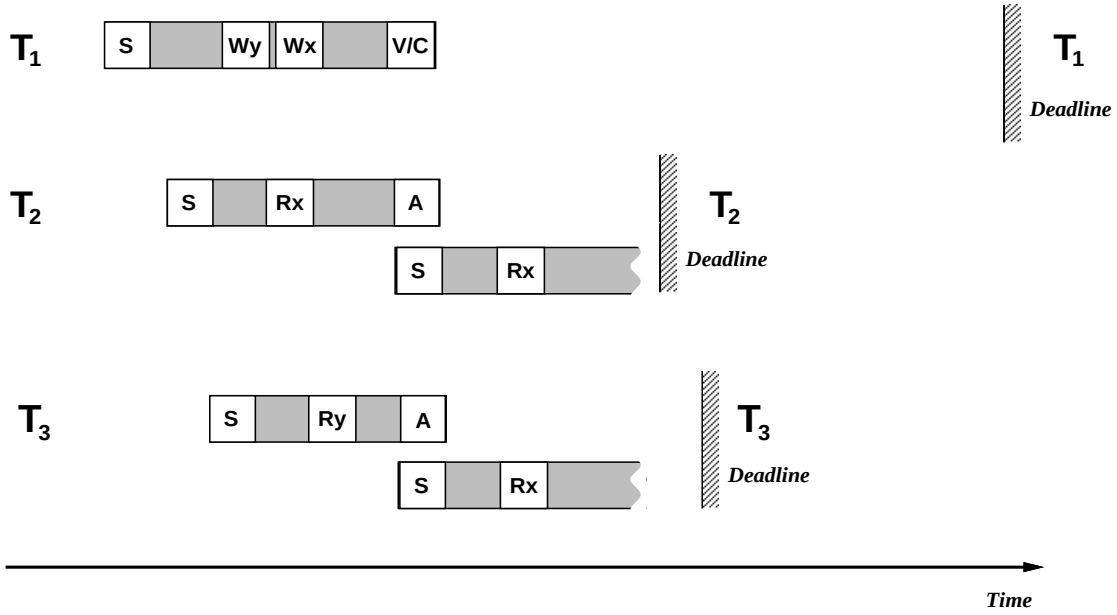

Figure 5: Missing deadlines under OCC-BC algorithm.

However, if we are not careful, delaying commit could also increase the number of transactions missing deadlines because the transactions were not restarted as early as they could have been. Let us look at Figure 6. The commitment of transaction T_2 is delayed, but since T_1 was not restarted until T_2 has committed, T_1 still misses the deadline. If T_1 could restart immediately after T_2 finishes, it would have a better chance of meeting its deadline. The problem here is that the

data written by a transaction is not made available to other transactions until the transaction has committed. In MSCC-DC, we allow T_1 to read the item x written by T_2 after the validation of T_2, without waiting for the write stage (*i.e.* Commit) to finish. This gives T_1 the opportunity to restart as if T_2 was committed immediately after the validation stage as illustrated in Figure 7.

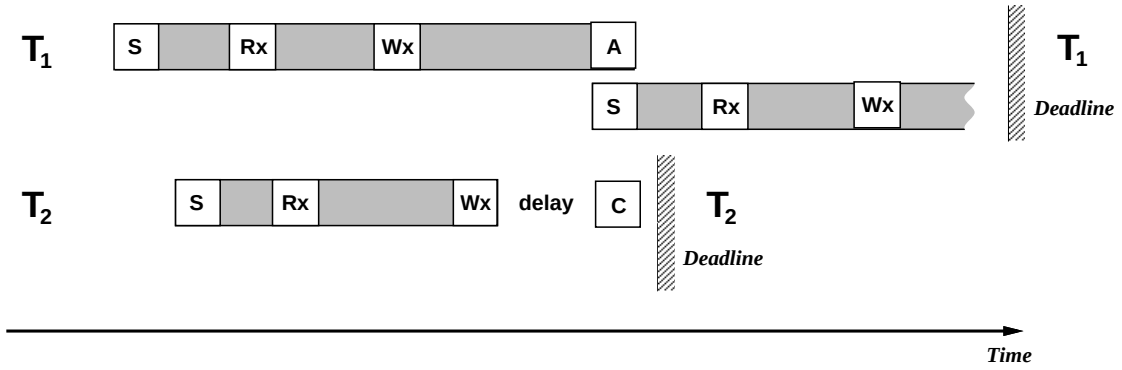

Figure 6: Example of delayed commit under OCC-BC algorithm.

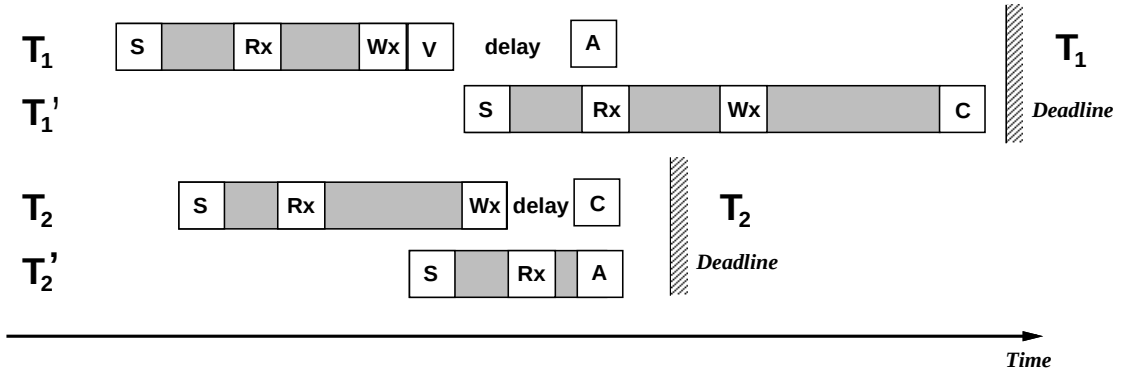

Figure 7: Example using Multi-version Speculative Concurrency Control with Delayed Commit.

3 MSCC with Delayed Commit

To simplify the problem, we will assume that transaction execution goes through 3 stages: *read, validate, and write*. However, during the validation stage, if (potential) conflicts are detected, instead of aborting the conflicting transactions, it will follow the algorithm described below. We say a transaction is *FINISHED* when it is at the end of the validation stage.

3.1 Assumptions

Without any loss of generality, we make the following assumptions:

1. Each transaction only writes once to each variable. All writes occur at the end of each transaction.
2. All transactions have equal priority.
3. Once a transaction is started, the system is aware of its deadline.

3.2 Definition and notation

Here are some terms and notation symbols which are used throughout this paper:

- T_i^0: Primary copy of transaction T_i. T_i^0 is the transaction that was first run on behalf of transaction number i.[2] It runs optimistically, and only reads the committed data.
- T_i^j: Secondary copies for transaction T_i. T_i^k is a running copy of transaction i. It is started because the primary transaction T_k^0 of another transaction k has *FINISHED*. The superscript k indicates it is started on behalf of the transaction k. T_i^k will reads dirty data from that primary transaction T_k^0.
- *WriteSet*(T_i^j): contains all the objects that transaction T_i^j wrote.
- *WriteList*(T_i): a list of all the objects and their values that transaction (T_i) wrote.
- *ReadSet*(T_i^j): contains all the objects that transaction T_i^j has read.
- *ReadList*(T_i^k)[3]: contains all the objects that transaction T_i^k read from *WriteList*.
- t: Current system time.
- D_i: Deadline of transaction T_i.

[2] All primary transactions will have the superscript 0.

[3] only a secondary transaction will have a *ReadList*

3.3 Primary transaction

We describe the algorithm for the primary transactions:

for each primary transaction T_i^0
  if T_i^0 wants to read data X
    if $X \in$ *WriteSet*(T_j^0)
      fork a copy of transaction T_i^0, call it T_i^j;
      T_i^j proceeds using the rules for secondary transactions described next with *WriteList*(T_j);
    T_i^0 reads the committed value of X;
    add X to *ReadSet*(T_i^0) and proceed;
  if T_i^0 writes data X
    add X to *WriteSet*(T_i^0);
    add X and its value, (X, v_x), to the *WriteList*(T_i);
  if T_i^0 reaches the end of transaction (at the *FINISHED* point)
    if for all transaction T_j^n, *ReadSet*$(T_j^n) \cap$ *WriteSet*$(T_i^0) = \phi$
      Commit(T_i^0);
      Secondary-Abort(T_i^k);
    else
      for each transaction T_j
        if *ReadSet*$(T_j^0) \cap$ *WriteSet*$(T_i^0) \neq \phi$
          start a new secondary transaction for T_j, call it T_j^i;
          send the *WriteList*(T_i) to T_j^i;
      Block(T_i^0);

3.4 Secondary transaction

We describe the algorithm for the secondary transactions:

for each secondary transaction $T_i^j (j \neq 0)$
  set *WaitForSet* $= T_k$ where T_k is the primary transaction which T_i^j received the *WriteList* from;
  if T_i^j wants to read data X
    if $X \in$ *WriteList*(T_j)
      read X from the *WriteList*;
      add X to *ReadList*(T_i^j);
    else
      read the committed value of X;
      add X to *ReadSet*(T_i^j);
    continue the transaction;
  if T_i^j wants to write data X
    add X to *WriteSet*(T_i^j);
  if T_i^j reaches the end of transaction (at the *FINISHED* point)
    Block(T_i^j);

3.5 Committing

The scheduler process determines when a transaction will actually attempt to commit, and follows the algorithm below:

for each transaction T_i
    When $t = D_i$ - slack time
        if no secondary transactions has finished
            Primary-Commit(T_i^0);
        else
            call Secondary-Commit(T_i^k) where
            T_i^k is the last secondary transaction for T_i that finished before D_i;

Slack time is the time needed to allow execution of one of the commit functions, and for the transaction to write data value into the database.

3.6 Functions

Some functions that are used in the primary and secondary transactions' algorithms:

Primary-Commit(T_i^0)
BEGIN
    for each primary transaction T_j^0
        if $ReadSet(T_j^0) \cap WriteSet(T_i^0) \neq \phi$
            Primary-Abort(T_j^0);
            promote the secondary transaction T_j^i to be the new primary transaction T_j^0;
            /* T_j^i was started when T_i^0 *FINISHED* */
    for each secondary transaction T_j^k
        if $T_i \in WaitForSet(T_j^k)$ of secondary transaction T_j^k
            set $WaitForSet(T_j^k) = \phi$;
        else if $ReadSet(T_j^k) \cap WriteSet(T_i^0) \neq \phi$
            Secondary-Abort(T_j^k);
    for all k **Secondary-Abort**(T_i^k);
    Commit(T_i^0);
END

Secondary-Commit(T_i^j)
BEGIN
    Primary-Abort(T_i^0);
    Secondary-Abort(T_i^k), $k \neq j$;
    if $WaitForSet(T_i^j) \neq \phi$
        Primary-Commit(T_j^0);
    for each primary transaction T_n^0

```
        if ReadSet(T_n^0) ∩ WriteSet(T_i^j) ≠ φ
            Primary-Abort(T_n^0);
            restart a new primary transaction for T_n;
    for each secondary transaction T_n^k
        if ReadSet(T_n^k) ∩ WriteSet(T_i^j) ≠ φ
            Secondary-Abort(T_n^k);
    Commit(T_i^k);
END

Primary-Abort(T_i^0)
BEGIN
    for all j
        if WaitForSet(T_j^i) = T_i
            Abort(T_j^i);
    Abort(T_i^0);
END

Secondary-Abort(T_i^k)
BEGIN
    Abort(T_i^k);
END
```

4 Analysis of the MSCC-DC algorithm

We will now go through an example to illustrate exactly how the algorithm works. Suppose we have the transactions in figure 8. When transaction T_1 $FINISHES$, it is blocked. Meanwhile, the *WriteList*(T_1), containing the variable X and its value wrote by T_1^0, is made available to both T_2 and T_3, T_2 restarts a secondary transaction T_2^1 since the *ReadSet*(T_2^0) contains X. T_3 starts a secondary copy T_3^1 later on, when it attempts to read X. Both T_2^1 and T_3^1 will read the X value from the *WriteList*(T_1), and all other variable value from the committed data. This situation is shown in figure 9.

By the time T_2^0 *FINISHES*, secondary transactions are started for both T_1(namely T_1^2) and T_3(namely T_3^2). Similarly, when T_3^0 *FINISHES*, T_1^3 and T_2^3 are started for T_1 and T_2 respectively. Figure 10 demonstrates the case. The deadline of T_2 is eventually reached. Since neither secondary copy of T_2 has finished yet, T_2^0 is committed and all the secondary transaction for T_2 are aborted. This leads to the abortion of primary transactions T_1^0 and T_3^0 because they both conflict with T_2^0. Secondary transactions T_1^2 and T_3^2 are promoted to become primary transactions now that all the data they read are committed. T_1^3 and T_3^1 are aborted since primary transactions that caused them to be started have been aborted. The result is shown in figure 11.

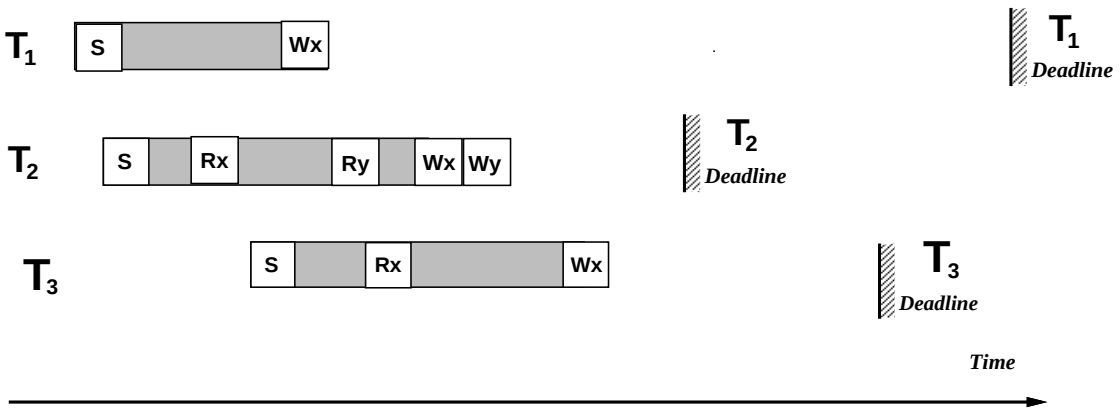

Figure 8: MSCC-DC: start

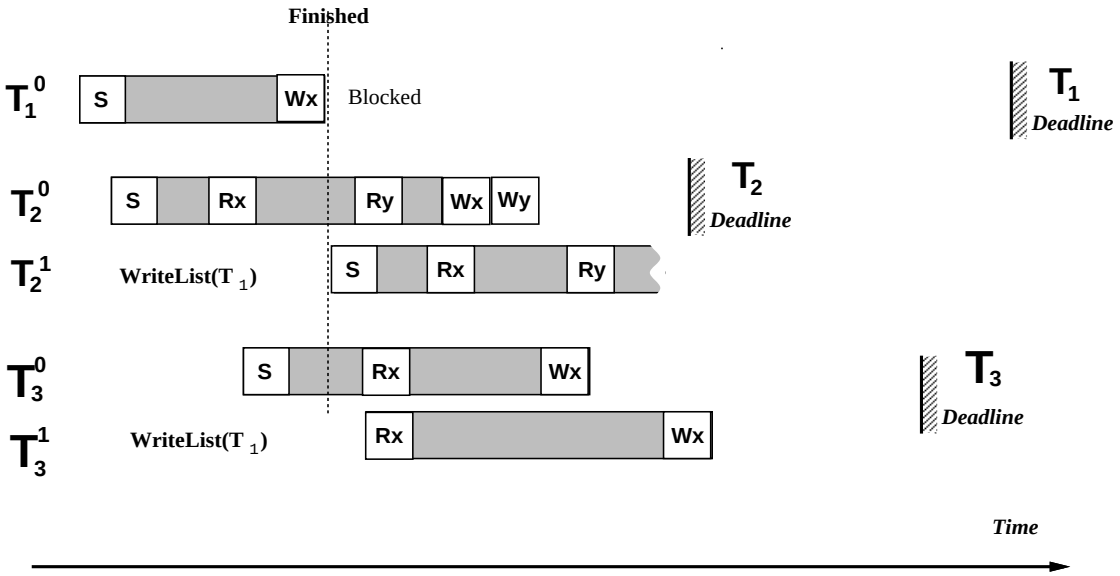

Figure 9: MSCC-DC: T_1 *finishes*

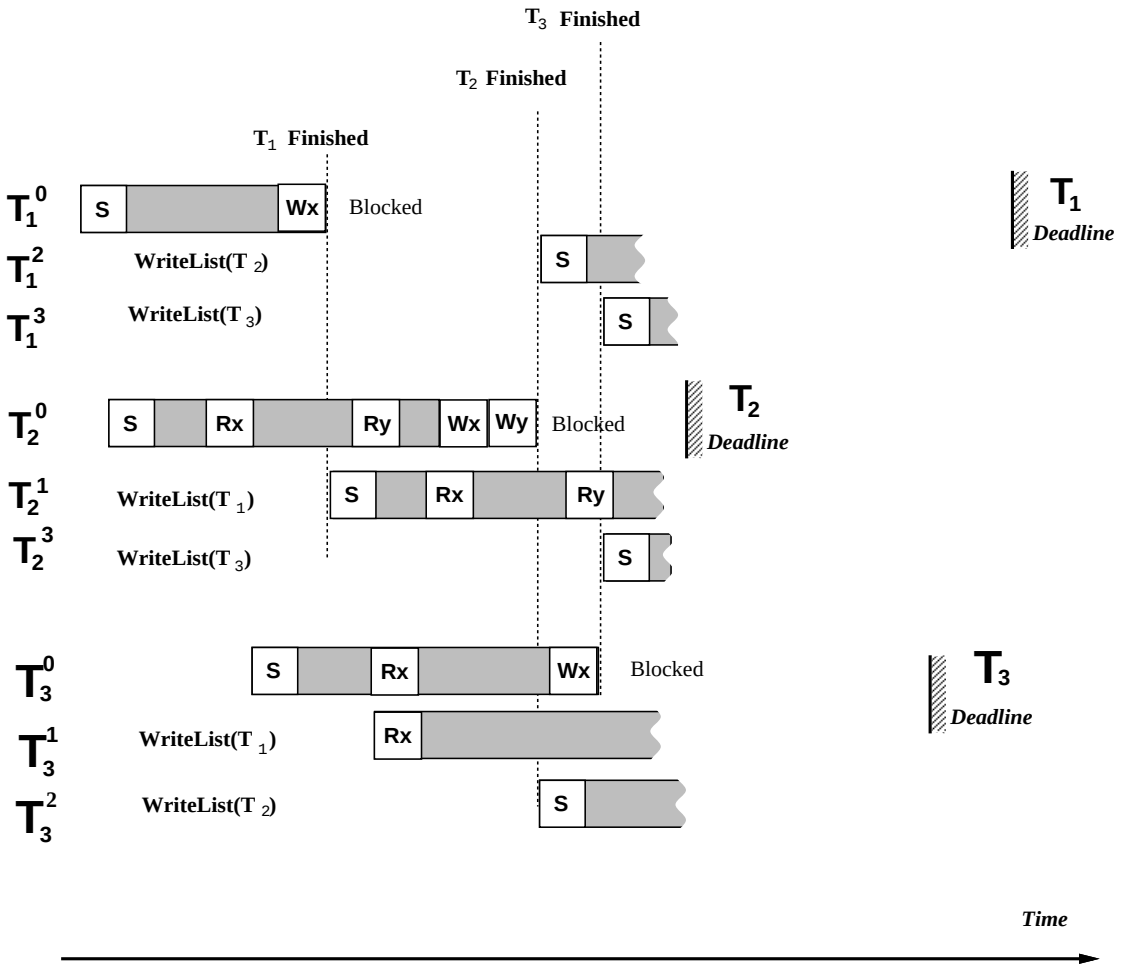

Figure 10: MSCC-DC: T_2 and T_3 *finish*

The same steps will be repeated when the new primary transactions for T_1 and for T_3 *finish*. New secondary transactions will be started, as seen in figure 12. Eventually, T_3^2 will commits because of the deadline, aborting the primary transaction for T_1, and the secondary transaction for T_1 becomes the primary transaction.

We shall sketch a proof that the algorithm always produces a schedule that is serializable by using induction on the number of transactions in the system. First, we know from the algorithm that only one copy of each transaction will commit, and all the other copies of that transaction will be aborted. Consider the base step of two transactions T_1 and T_2 in the system.

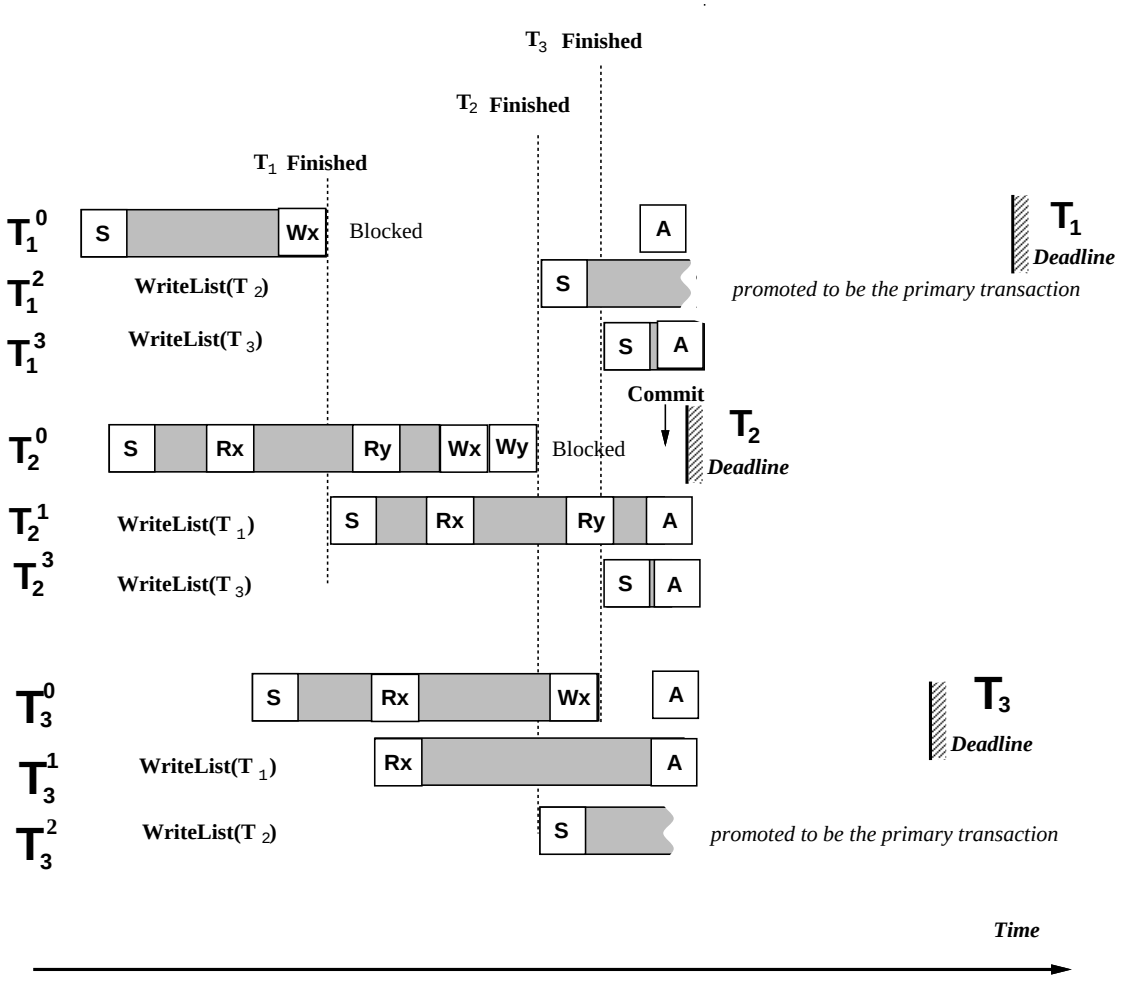

Figure 11: MSCC-DC: T_2^0 commits

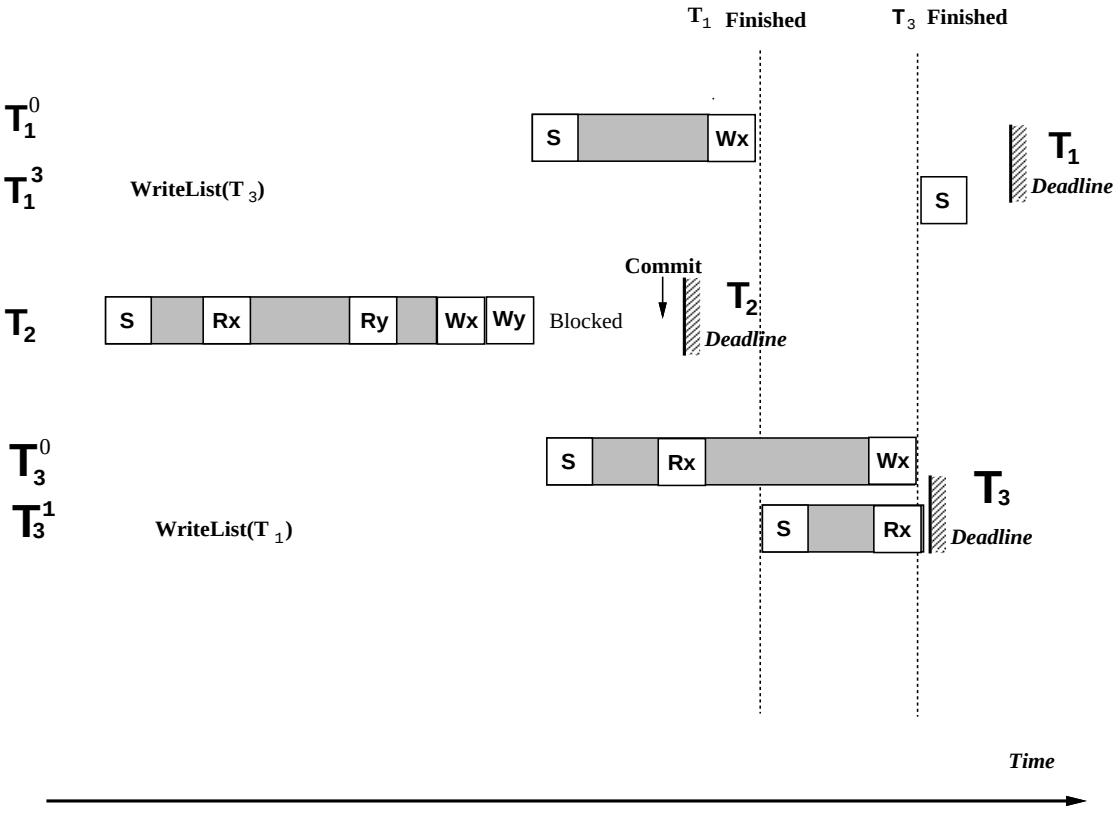

Figure 12: MSCC-DC: after aborting

There are two possible cases:

- two transactions do not conflict with each other:
  The proof of this is trivial. Both transactions will just run under an optimistic algorithm. No secondary transaction will be created. Any end result produced is equivalent to the result of a serial execution order of either $T_1 \rightarrow T_2$ or $T_2 \rightarrow T_1$.

- two transactions conflict with each other: By the MSCC-DC algorithm, there will be up to four possible transactions running in the system. T_1^0, T_1^2, T_2^0, and T_2^1.

  1. Since T_1 and T_2 conflict with each other, and T_1^0, T_2^0 both read committed data values from the database, they cannot both commit and produce a result in this system.
  2. T_1^2 and T_2^1 cannot both commit in this system either. In order for T_1^2 to commit, it has to first let T_2^0 commit. This means T_2^1 will be aborted. The same argument holds if T_2^1 commits.
  3. If the result of the system is produced by T_1^0 and T_2^1, since T_2^1 reads data values from T_1^0, this is the same schedule as the serial execution schedule of $T_1 \rightarrow T_2$.
  4. By the same token, the result of the system produced by T_1^2 and T_2^0 is the same schedule as the serial execution schedule of $T_2 \rightarrow T_1$.

  The induction step is as follows:

- Assumption: assume for any number n of transactions: $T_1, ..., T_n$, any final result produced by MSCC-DC is equivalent of a result produced by a serialized schedule.

- Induction: assume there are *n+1* number of transactions: $T_1, ..., T_n, T_{n+1}$, we need to prove any result produced by running the MSCC-DC algorithm is equivalent of a result produced by a serialized schedule.
  Consider the state when the first transaction tries to commit during the running of MSCC-DC. There are two cases:

  1. The transaction trying to commit is a primary transaction (T_i^0).
  2. The transaction trying to commit is a secondary transaction (T_i^k).

  In the first case, the call to the function **Primary-Commit**(T_i^0) is made, and T_i^0 will be committed. Any of the remaining primary transactions are aborted if they conflict with T_i^0. The secondary transactions (which read the data written by T_i^0) for those transactions are promoted to be primary transactions. This means T_i^0 would be scheduled before any of the n other transactions in any of the schedules produced by MSCC-DC. We assumed the schedule for any n transactions produced by the MSCC-DC algorithm is serializable. By adding T_i in front of that schedule, it proves the total schedule for the *n+1* number of transactions is serializable.

For the second case, we note when a secondary transaction (T_i^k) tries to commit, it calls to the function **Secondary-Commit**(T_i^k). The function first calls **Primary-Commit**(T_k^0) which is where T_i^k read data from. Since we actually have to commit a primary transaction before we commit any secondary transaction, the proof of this case is almost the same as the first one.

One major problem in allowing transactions to read uncommitted data is that if the transaction which wrote the data aborts, it may cause cascading aborts. MSCC-DC avoids this problem because it only allows reading of dirty data from primary transactions. *i.e.* no transaction will read dirty data from a transaction which also read dirty data. Below we describe the four kinds of action a transaction can take:

1. Primary transaction commit: When a primary transaction T_i^0 commits, it aborts all of its own secondary transactions, and aborts all other conflicting primary transactions T_j^0 and secondary transactions T_j^k. (The $\rightarrow$ here means "leads to the abortion of")
   $C(T_i^0) \rightarrow A(T_i^k)$
   $C(T_i^0) \rightarrow A(T_j^0)$
   $C(T_i^0) \rightarrow A(T_j^k)$

2. Secondary transaction commit: When a secondary transaction T_i^k commits, it aborts its primary copy, and all other conflicting primary transaction T_j^0 and secondary transaction T_j^k.
   $C(T_i^k) \rightarrow A(T_i^0)$
   $C(T_i^k) \rightarrow A(T_j^0)$
   $C(T_i^k) \rightarrow A(T_j^k)$

3. Primary transaction abort: When a primary transaction T_i^0 aborts, it will aborts all secondary transactions T_j^k which read the dirty data wrote by T_i^0.
   $A(T_i^0) \rightarrow A(T_j^k)$

4. Secondary transaction abort: When a secondary transaction T_i^k aborts, it does not cause any more abort.

As we can see from above, the longest aborting sequence happens when a secondary transaction commits, which will cause a primary transaction to commit. This commitment then may abort other primary transactions, and lead to the abortion of other secondary transactions:

$$C(T_i^k) \rightarrow C(T_j^0) \rightarrow A(T_m^0) \rightarrow A(T_n^l)$$

When a primary transaction commits, it will abort another primary transaction which then in turn may abort another secondary transaction:

$$C(T_i^0) \rightarrow A(T_j^0) \rightarrow A(T_l^k)$$

4.1 Analysis

Previous concurrency control algorithms such as OCC-BC do not link the commitment of transactions with their deadlines, which is essential for real-time DBMS. These schemes heavily favor transactions which finish early instead of those which have earlier deadlines. Some schemes try to solve the problem by assigning priority to transactions according to deadlines. MSCC-DC provides the link between commitment of transactions and their deadlines without actually assigning priorities to transactions. This occurs at the expense of using polynomially more processing power. For each transaction, the number of secondary transactions is bounded by the number of conflicts it has with other primary transactions. Given that we have n primary transactions, each transaction can have up to $n-1$ secondary transactions. Hence the number of total transactions in the system is bounded by $O(n^2)$.

We only allow each secondary transaction to read uncommitted data from a single primary transaction. A better result might be obtained if we permit some secondary transactions to read uncommitted data from several primary transactions provided those primary transactions do not conflict with each other. Furthermore, in our new algorithm above, we delayed the commitment of transactions until they actually reach their deadlines. It is possible that making the decision to commit earlier may find a better solution.

5 Conclusion and Future Research

The algorithm used in MSCC-DC combines the advantages of other concurrency control algorithms. However, many interesting research problems still remain to be investigated.

- Can an optimal commit time be found? When should a primary or secondary transaction commit? How can we dynamically chose which is the better group of transactions to commit?
- What changes need to be made to add priorities? How would this change the performances of the algorithm?
- Simulation remains to be done to show how much improvement is gained. This will also provide an opportunity to test which is the more deciding factor in the performance of the algorithm.
- How would performance differ if we have some secondary transactions running pessimistically?

References

[Abbo88] Robert Abbott and Hector Garcia-Molina. "Scheduling real-time transactions: A performance evaluation." In *Prooceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, Ca, 1988.

[Abbo92] R. Abbott and H. Garcia-Molina. "Scheduling real-time transaction: A performance evaluation." *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.

[Agra87] R. Agrawal, M. Carey, and M. Linvy. "Concurency control performance modeling: Alternatives and implications." *ACM Transaction on Database Systems*, 12(4), December 1987.

[Best92] Azer Bestavros. "Speculative Concurrency Control: A position statement." Technical Report TR-92-016, Computer Science Department, Boston University, Boston, MA, July 1992.

[Hari90a] Jayant R. Haritsa, Michael J. Carey, and Miron Linvy. "Dynamic real-time optimistic concurrency control." In *Prooceedings of the 11th Real-Time Systems Symposium*, December 1990.

[Hari90b] Jayant R. Haritsa, Michael J. Carey, and Miron Linvy. "On being optimistic about real-time constraints." In *Prooceedings of the 1990 ACM PODS Symposium*, April 1990.

[Huan90] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham. "Real-time transaction processing: Design, implementation and performance evaluation." Technical Report COINS TR-90-43, University of Massachusetts, Amherst, MA 01003, May 1990.

[Kim91] Woosaeng Kim and Jaideep Srivastava. "Enhancing real-time dbms performance with multiversion data and priority based disk scheduling." In *Prooceedings of the 12th Real-Time Systems Symposium*, December 1991.

[Kung81] H. Kung and John Robinson. "On optimistic methods for concurrency control." *ACM Transactions on Database Systems*, 6(2), June 1981.

[Lin90] Yi Lin and Sang Son. "Concurrency control in real-time databases by dynamic adjustment of serialization order." In *Proceedings of the 11th Real-Time Systems Symposium*, December 1990.

[Mena82] D. Menasce and T. Nakanishi. "Optimistic versus pessimistic concurrency control mechanisms in database management systems." *Information Systems*, 7(1), 1982.

[Robi82] John Robinson. *Design of Concurrency Controls for Transaction Processing Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1982.

[Sha91] Lui Sha, R. Rajkumar, Sang Son, and Chun-Hyon Chang. "A real-time locking protocol." *IEEE Transactions on Computers*, 40(7):793–800, 1991.

[Son92] S. Son, S. Park, and Y. Lin. "An integrated real-time locking protocol." In *Prooceedings of the IEEE International Conference on Data Engineering*, Tempe, AZ, February 1992.

[Stan88] John Stankovic and Wei Zhao. "On real-time transactions." *ACM, SIGMOD Record*, 17(1):4–18, 1988.