In Proceedings of ISCA'94: The 1994 International Symposium on Computers and their Applications, Long Beach, CA, March 1994.

Multi-version Speculative Concurrency Control with Delayed Commit

Azer Bestavros*
Computer Science Department
Boston University
Boston, MA 02215

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. The algorithm propagates uncommitted data to other outstanding transactions thus allowing more speculative schedules to be considered.

1 Introduction

A real-time database management system is an amalgamation of conventional database management and real-time scheduling. Like database systems, it has to process transactions and guarantee database consistency using a concurrency control algorithm. Furthermore, it has to operate in real-time, satisfying time constraints on each transaction [1].

Mena [8] classified concurrency control algorithms into optimistic and pessimistic algorithms. In [3], we proposed a new approach, Speculative Concurrency Control (SCC), which combines the Pessimistic and Optimistic Concurrency Control (PCC and OCC) algorithms. SCC adapts to developing conflicts by creating multiple shadows, each dealing with a different set of conflicts, rather than waiting for these conflicts to materialize or subside. This makes SCC-based algorithms better suited for real-time applications. In this paper, we propose the Multi-version SCC algorithm with Delayed Commitment (MSCC-DC), which combines the basic SCC algorithms with other ideas that have been studied for real-time DBMS. This is summarized below.

Typically, transaction conflicts result from an uncommitted transaction's attempt to write some data that is later read by a second uncommited transaction. This read/write conflict creates a potential hazard since there are two values of the data: the committed value (previously existing in the database), and the new value (written by the first transaction). Under SCC, both transactions run using the OCC algorithm. However, when a read/write conflict is detected, an alternate *shadow* of the second (reader) transaction is started and executes until the conflict point (the attempt to read the data) where it is blocked. In MSCC-DC, instead of blocking, the alternate shadow is allowed to continue by reading the data value written by the first transaction. Since the first transaction has not yet committed, we say that the second transaction has read uncommitted data. In general, most concurrency control schemes being studied in the literature do not allow transactions to read uncommitted data since it could easily cause commit dependencies and cascading aborts. However, by limiting the chain of transactions that read uncommitted (dirty) data, we can bound the number of aborts caused by a materialized conflict.

If a transaction T commits immediately after it finishes its computation, it will cause all other transactions that conflict with it to abort. 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 commitment of a transaction T may result in the discovery of a 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 are not delayed by the delayed-commitment of T.

2 Previous Work

In [2], Agrawal concluded that pessimistic locking protocols, due to their conservation of resources, perform better than optimistic techniques for conventional DBMS. Pessimistic two-phase locking algorithms detect potential conflicts as they occur. However, they may suffer possible unbounded waiting due to blocking and deadlocks. The resource conservation nature of pessimistic algorithms becomes a drawback in a real-time environment, where meeting time-

*This research is partially supported by GTE Labs.

constraints has a much higher priority than conserving resources.

In [6, 5], Haritsa, Carey and Linvy showed that for a real-time DBMS with firm deadlines (transactions missing their deadlines are immediately discarded), optimistic algorithms outperform 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. This makes OCC restart-based protocols more attractive in real-time DBMS than PCC blocking-based algorithms.

Classical OCC [7] consists of three stages of execution for a transaction: *read, validation, and write*. The key stage 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 are made visible to other transactions.

Conflict resolution in OCC schemes is always done by aborting the validating transaction. 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) [8, 9] 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 restarts.

SCC combines the advantage of both optimistic and pessimistic schemes while avoiding their disadvantages [3]. 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 the 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, consider two transactions T_1, and T_2, such that T_2 reads item x after T_1 has updated it. The basic OCC algorithm (figure 1) restarts transaction T_2 when it enters its validation stage. Obviously, the likelihood of the restarted transaction T_2 meeting its timing constraint decreases. The OCC-BC algorithm avoids waiting unnecessarily until T_2's validation stage in order to restart it. This is illustrated in figure 2, where T_2 is restarted as soon as T_1 broadcasts its commit.

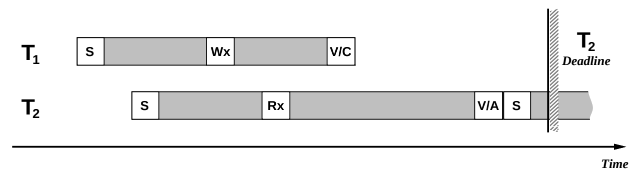

Figure 1: Example under the basic OCC algorithm.

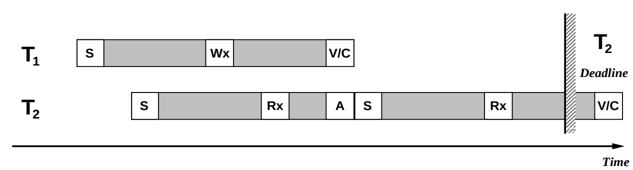

Figure 2: Example under the OCC-BC algorithm.

Using the SCC approach, instead of pessimistically blocking T_2, or optimistically ignoring the conflict until the validation stage, a copy (or *secondary shadow*) of the reader transaction T_2 is made. The original transaction T_2 (*primary shadow*) continues to run uninterrupted, while the shadow T_2' is restarted (or forked off). Only one of the two shadows is allowed to commit; the other is aborted. Figures 3 and 4 show two possible scenarios that may develop depending on the time needed for transaction T_2 to reach its validation stage. Obviously, SCC achieves 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 validat-

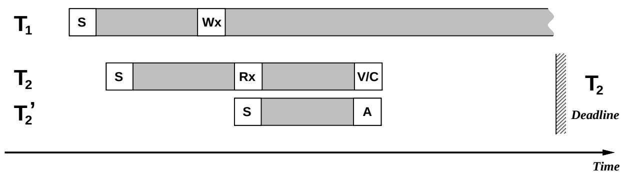

Figure 3: SCC schedule with an undeveloped conflict.

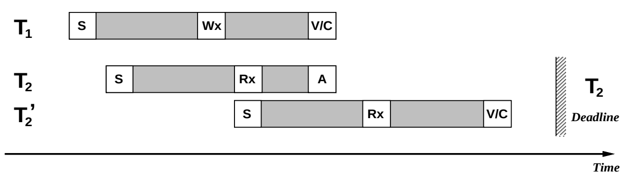

Figure 4: SCC schedule with a developed conflict.

ing, it may cause a larger number of transactions to abort and miss their deadlines. For example, in figure 5, committing T_1 as soon as it is validated causes both T_2 and T_3 to abort, and both of them cannot be restarted early enough to meet their deadlines. In [6], Harista showed that by making a lower priority transaction wait after it is validated, the number of transaction restarts is reduced, thus increasing the number of transactions meeting their deadlines.

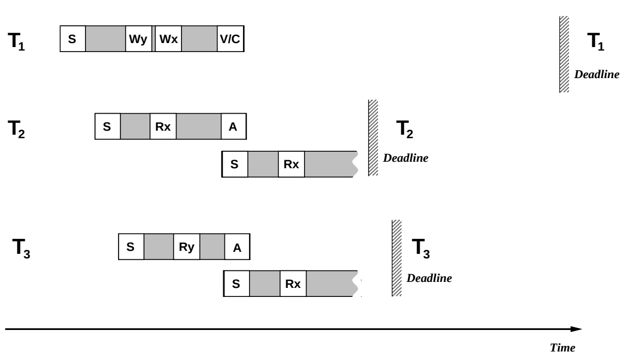

Figure 5: Missing deadlines under OCC-BC.

However, if we are not careful, delaying the commitment of a transaction could also increase the number of transactions missing their deadlines because the transactions were not restarted as early as they could have been. For example, in 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 had 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 necessarily waiting for the 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.

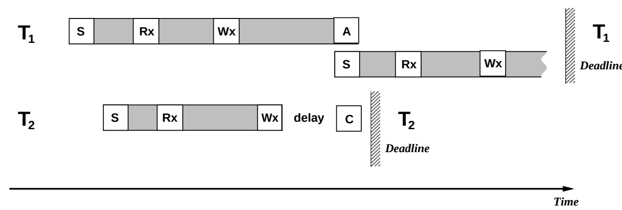

Figure 6: A delayed commit under OCC-BC.

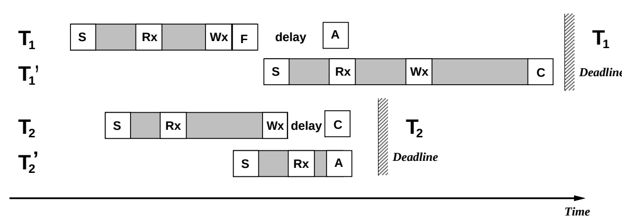

Figure 7: Example under the MSCC-DC algorithm.

3 The MSCC-DC Algorithm

To simplify the problem, we will assume that transaction execution goes through 3 stages: *read, validate, and write*. During the validation stage of a transaction $\mathcal{T}$, and if conflicts with other transactions are detected, then instead of aborting the conflicting transactions, we *delay* the commitment of $\mathcal{T}$. A transaction is said to be *finished* when it is at the end of the validation stage, but not yet committed. Data written by a finished transaction may be propagated to secondary shadows of other transactions in the system, but not to primary shadows. Without loss of generality, we assume that a transaction writes objects it modifies only once near the end of its execution, that all transactions have equal priority, and that transactions' deadlines are known to the system.

Let T_i^0 denote the first process (primary shadow) created on behalf of transaction T_i. T_i^0 runs optimistically, and only reads data committed to the database. Let $T_i^j, j > 0$, denote the secondary shadows of transaction T_i. Such a shadow, T_i^k, is started to account for a read/write conflict between T_i and the primary shadow T_k^0 of some other transaction T_k. If T_i^k needs to read data written by a not-yet-finished T_k^0, then T_i^k blocks waiting for T_k^0 to fininsh. When T_k^0 is finished, its uncommitted data may be propagated to T_i^k, whose execution is resumed.

The MSCC-DC algorithm requires the maintenance of a number of data structures. $WriteSet(T_i^j)$ is the set of objects written by shadow T_i^j. $WriteList(T_i)$ contains the values of objects written by the *finished* transaction T_i. $ReadSet(T_i^j)$ is the set of objects read by shadow T_i^j. $ReadList(T_i^k)$ contains all the objects read by the shadow T_i^k from *WriteList*. We denote the current time by t and the deadline of transaction T_i by D_i.

The details of MSCC-DC can be found in [4]. Here we introduce the algorithm using an example. Consider the set of shadows 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 shadow 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 value of X from the $WriteList(T_1)$, and all other variables from the committed data in the database. This is shown in figure 9.

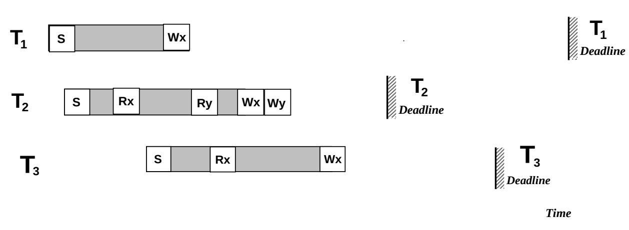

Figure 8: MSCC-DC: start

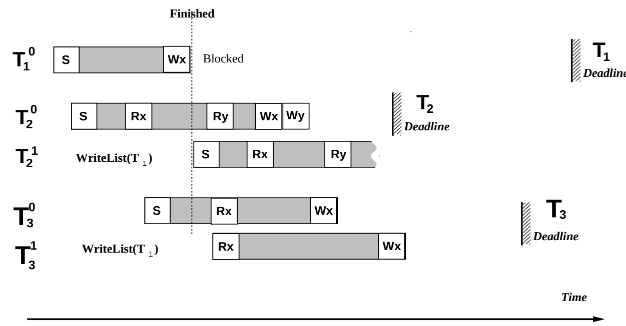

Figure 9: MSCC-DC: T_1 *finishes*

By the time T_2^0 *finishes*, secondary shadows 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 (see Figure 10). The deadline of T_2 is eventually reached. Since the secondary shadows of T_2 are not finished yet, T_2^0 is committed and all the secondary shadows for T_2 are aborted. This leads to the abortion of the primary shadow T_1^0 and T_3^0 because they both conflict with T_2^0. Secondary shadows T_1^2 and T_3^2 are *promoted* to become primary shadows, now that all the data they read is committed. T_1^3 and T_3^1 are aborted since the primary shadows that caused them to be started were aborted. The state of the set of shadows is shown in figure 11.

The same steps will be repeated when the new primary shadow for T_1 and for T_3 *finish*. New secondary shadows will be started, as seen in figure 12. Eventually, when it gets closer to its deadline, T_3^2 will commit, thus resulting in the abortion of the primary shadow for T_1 and the secondary shadow for T_1 is promoted to become the primary shadow.

In [4], we sketch a proof (by induction) that the algorithm always produces a schedule that is serializable. Also, we prove that MSCC-DC does not suffer from the problem of cascading aborts since the propagation of uncommitted data occurs from primary to secondary shadows and not vice-versa.

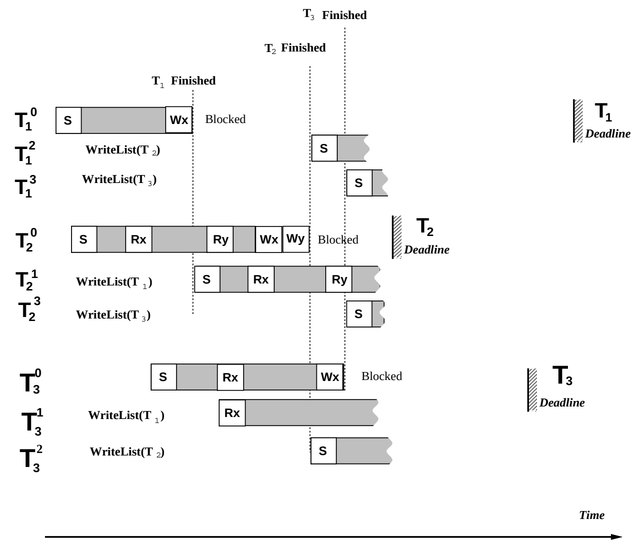

Figure 10: MSCC-DC: T_2 and T_3 *finish*

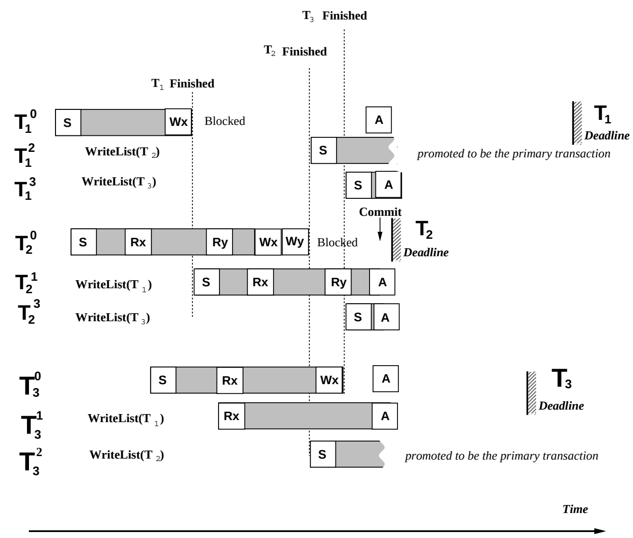

Figure 11: MSCC-DC: T_2^0 commits

4 Conclusion

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 that finish early instead of those with tighter 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 more processing resources.

In MSCC-DC, we allow a secondary shadow to read uncommitted data from a single primary transaction. A better result may 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 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 result in a better performance.

Many interesting research problems remain to be investigated: Can an optimal commit time be found? When should a primary shadow commit? How can we dynamically chose the *better* group of finished transactions to commit? What changes need to be made to add priorities? How would this change the performances of the algorithm? How would MSCC-DC perform in simulations compared to other PCC, OCC, and SCC algorithms?

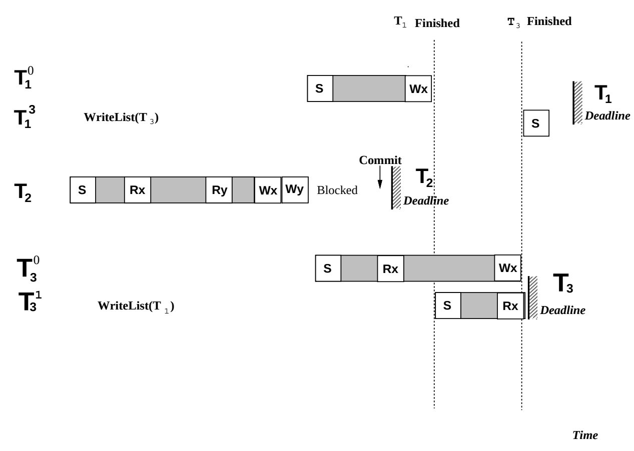

Figure 12: MSCC-DC: after aborting

Acknowledgments:
I would like to thank Biao Wang and Spyros Braoudakis for their help in developing some of the ideas and examples presented in this paper. This work was partially supported by GTE fund number 3658-3.

References

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

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

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

[4] Azer Bestavros and Biao Wang. Multi-version speculative concurrency control with delayed commit. Technical Report TR-93-014, Computer Science Department, Boston University, Boston, MA, October 1993.

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

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

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

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

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