Speculative Concurrency Control

A POSITION STATEMENT

Azer Bestavros
Computer Science Department
Boston University

1 Introduction

Traditional concurrency control algorithms can be broadly classified as either *pessimistic* or *optimistic* [Mena82]. Pessimistic Concurrency Control (PCC) algorithms avoid any concurrent execution of transactions as soon as conflicts that *might* result in future inconsistencies are detected. On the contrary, Optimistic Concurrency Control (OCC) algorithms allow such transactions to proceed at the risk of having to restart them in case these suspected inconsistencies *materialize*.

For real-time database applications where transactions execute under strict timing constraints, maximum concurrency (or throughput) ceases to be an expressive measure of performance. Rather, the number of transactions completed before their set deadlines becomes the decisive performance measure. Recently, several attempts at modifying PCC and OCC algorithms to suit real-time database applications have been proposed. These attempts have been successful in the sense that they improved the performance of the basic PCC and OCC algorithms in the context of real-time database management systems (RTDBMS).

Most real-time concurrency control schemes considered in the literature are based on Two-Phase Locking (2PL) [Abbo88, Stan88, Huan90, Sha91] – a PCC algorithm that has been well studied in traditional database management systems (DBMS). Despite its widespread use, 2PL has some properties (such as the possibility of deadlocks and/or long, unpredictable blocking times), which damage its appeal for RTDBMS, where in addition to preserving database consistency, strict timing constraints must be honored. Recently, some alternatives to 2PL for real-time systems have been proposed [Hari90b, Hari90a, Huan91, Kim91, Lin90, Son92]. A class of these concurrency control protocols is based on OCC, which due to its potential for a high degree of concurrency was expected to perform better than 2PL when integrated with priority-driven CPU scheduling in real-time database systems. In addition, the non-blocking and deadlock free properties of OCC are especially attractive to real-time transaction processing. The performance studies in [Hari90b, Hari90a, Huan91] confirm that, for systems with firm deadlines. OCC outperforms 2PL under low system loads and high resource availability.

In this position statement we propose a categorically different approach to Concurrency Control that is particularly well-suited for real-time database applications. We propose the use of redundant computations to start as early as possible on an alternative schedule, once a conflict that threatens the consistency of the database is detected. This alternative schedule is adopted *only if* the suspected inconsistency materializes; otherwise, it is abandoned. Due to its nature, we term our concurrency control algorithm *Speculative*. The description given here encompasses many algorithms that we call collectively Speculative Concurrency Control (SCC) algorithms.

SCC algorithms combine the advantages of both PCC and OCC algorithms, while avoiding their disadvantages. On the one hand, SCC resembles PCC in that potentially harmful conflicts are detected as early as possible, allowing a head-start for alternative schedules, and thus increasing the chances of meeting the set time constraints – should these alternative schedules be needed. On the other hand, SCC resembles OCC in that it allows conflicting transactions to proceed concurrently, thus avoiding unecessary delays that may jeopardize their timely commitment.

Because of its reliance on redundant computation, SCC algorithms require the availability of enough capacity in the system. Throughout this paper, we make the assumption that an abundance of computing resources is, indeed, available. This *abundant resources assumption* may not be acceptable in a conventional system; for a real-time system, it is. Real-time systems are usually embedded in critical applications, in which human lives or expensive machinery are at stake. The sustained demands of the environments in which such systems operate pose relatively rigid and urgent requirements on their performance. Consequently, these systems are usually sized to handle transient bursts of heavy loads. This requires the availability of enough computing resources that, under normal circumstances, remain idle.

2 Speculative Concurrency Control

Various concurrency control algorithms differ basically in the time when conflicts are detected, and in the way they are resolved. The PCC and OCC alternatives represent the two extremes in terms of data conflict detection and conflict resolution. PCC locking protocols detect conflicts as soon as they occur and resolve them using blocking. OCC protocols, on the other hand, detect conflicts at transaction commit time and resolve them using restarts. In this section, we present SCC protocols, which detect conflicts as soon as they occur and resolve them using speculative redundant computations.

To illustrate the basic idea of the SCC approach, let us consider an example. Figure 1 shows a simple schedule for two transactions under the Broadcast Commit variant of OCC (OCC-BC). At the time when transaction T_2 requests to read data item x, all the information necessary to conclude that there is a conflict (and hence a potential consistency threat) between transactions T_2 and T_1 (which previously updated data item x) is available. Instead of pessimistically blocking T_2 – like PCC blocking-based protocols – and instead of optimistically ignoring the potential conflict – like OCC restart-based protocols – our suggested SCC approach would make a copy, or *shadow*, of the reader transaction – T_2 in this example. The original reader transaction T_2 continues to run uninterrupted, while the shadow transaction T_2' is restarted on a different processor and allowed to run concurrently. In other words, two versions of the same transaction are allowed to run in parallel, each one being at a different point of its execution. Obviously, only one of these two transactions will be allowed to commit; the other will be aborted. Notice that these two transactions will possibly have different underlying requirements for their commitment. In particular, the conflicts that will develop between each one of these two transactions and the remaining transactions in the system may well be different.

The protocol suggested above uses redundancy to explore *potential* serializable schedules as early as possible, thus increasing the possibility of committing the one that ends up being *adopted* without missing any of the deadlines of its constituent transactions. Figure 2 and figure 3 show two possible scenarios that may develop depending on the time needed for transaction T_2 to reach its validation phase. Each one of these scenarios corresponds to a different serialization order.

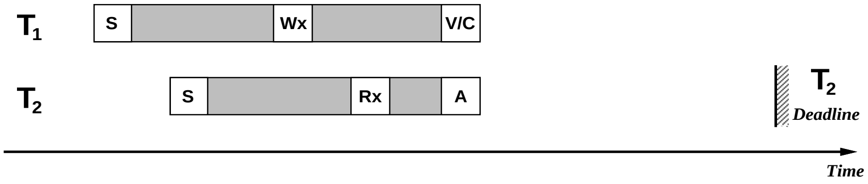

Figure 1: Transaction management under the OCC-BC algorithm.

In figure 2 T_2 reaches its validation phase before T_1. Thus, T_2 will be validated[1] and committed without any need to disturb T_1. Therefore, this schedule will be serializable with transaction T_2 preceding transaction T_1. Obviously, once T_2 commits, the shadow transaction T_2' has to be aborted.

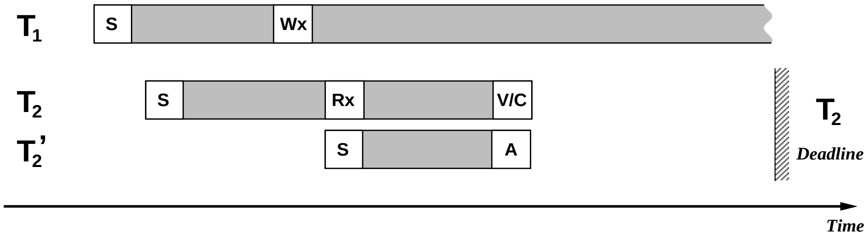

Figure 2: Schedule with an undeveloped potential conflict.

If, however, 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 must abort. With OCC-BC algorithms, T_2 would have had to restart when T_1 commits. This might be too late if T_2's deadline is near. With our SCC protocol, instead of restarting T_2, we simply abort T_2 and adopt its shadow transaction T_2'. This scenario is illustrated in figure 3.

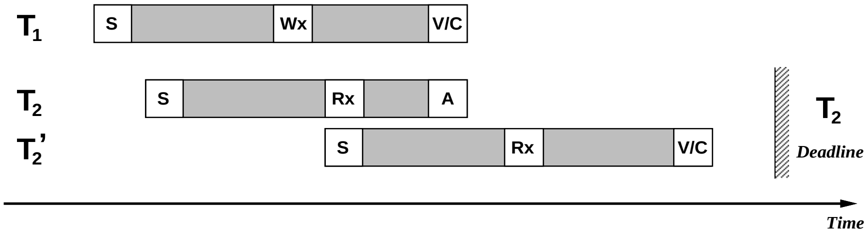

Figure 3: Schedule with a developed conflict.

With the proposed SCC algorithm, T_2' is scheduled as soon as the potentially harmful conflict between T_1 and T_2 is detected, maximizing its chances of meeting T_2's deadline. T_2' is an exact replica of T_2, in the sense that they both perform the same operations. However, it can very well be the case that they will not see the same database when they will perform their read operations. As a matter of fact, this is exactly our goal.

---

[1] since T_2's write-set does not intersect T_1's read-set (assuming that there are no conflicting actions other than the reading and writing of x).

Notice, that this flexibility is not gained without a cost. In particular, transaction T_2 had to be aborted resulting in wasted computations (see figure 3). This, however, is the same price that OCC and OCC-BC protocols would have had to incur anyway (see figure 1).[2] On the other hand, as we depicted in figure 2, T_2 could have successfully completed its execution if it reached its validation phase before T_1. In this case, T_2' becomes obsolete, and must be aborted.

3 Two-Shadow SCC Algorithm

In this section, we overview a simple yet powerful SCC-based algoritm, which can be thought of as a special case of the SCC-based algorithms described in [Best92a, Best92b, Best93a]. The algorithm – called Two-Shadow SCC (SCC-2S) – allows a maximum of two shadows per uncommitted transaction to exist in the system at any point in time: a *primary* shadow and a *standby* shadow.

Let T_i be any uncommitted transaction in the system. The primary shadow for T_i runs under the optimistic assumption that it will be the first (among all the other transactions with which T_i conflicts) to commit. Therefore, it executes without incuring any blocking delays. The standby shadow for T_i, on the contrary, is subject to blocking and restart. It is kept ready to replace the primary shadow, should such a replacement be necessary. The standby shadow runs under the pessimistic assumption that it will be the last (among all the other transactions with which T_i conflicts) to commit.

The SCC-2S algorithm resembles the OCC-BC algorithm in that primary shadows of transactions continue to execute either until they validate and commit or until they are aborted (by a validating transaction). The difference, however, is that SCC-2S keeps a *standby* shadow for each executing transaction to be used if that transaction must abort. The standby shadow is basically a replica of the primary shadow, except that it is blocked at the *earliest* point where a Read-Write conflict is detected between the transaction it represents and any other uncommitted transaction in the system. Should this conflict materialize into a consistency threat, the standby shadow is promoted to become the primary shadow, and execution is *resumed* (instead of being *restarted* as would be the case with OCC-BC) from the point where the potential conflict was discovered.

To illustrate how SCC-2S works, consider the schedule shown in figure 4. Both transactions T_1 and T_2 start with one primary shadow, namely T_1^0 and T_2^0. When T_2^0 attempts to read object x, a potential conflict is detected. At this point, a backup shadow, T_2^1, is created.[3] The primary shadows T_1^0 and T_2^0 execute without interruption, whereas T_2^1 blocks. Later, if T_1^0 successfully validates and commits on behalf of transaction T_1, the primary shadow T_2^0 is aborted and replaced by T_2^1, which resumes its execution, hopefully committing before its set deadline.

It is possible that multiple conflicts develop between executing transactions. Figure 5 illustrates the behavior of SCC-2S when a second conflict develops between T_2 and another transaction T_3. In particular, the primary shadow T_3^0 of T_3 attempts to write an object y that both shadows T_2^0 and T_2^1 had previously read. In this case, T_2^0 proceeds without any interruption, whereas T_2^1 is restarted and blocked as it attempts to read y. Should T_2^0 be aborted as a result of its conflict with T_3,[4] T_2^1 is promoted to become the primary shadow and is, thus, allowed to resume.

---

[2] Notice that this is not needed in PCC algorithms that rely on blocking.

[3] This can be easily done by forking off a process from T_2^0.

[4] Or as a result of its conflict with T_1 (as was the case in figure 4).

The SCC-2S algorithm allows at most two shadows for the same transaction to co-exist at any given time. It is possible, however, that more that two shadows will be needed over a stretch of time. Figure 6 illustrates such a situation. In particular, after T_2^1 is promoted to become the primary shadow for T_2, a standby shadow T_2^2 is forked off to account for the read-write conflict between T_2^1 and T_1.

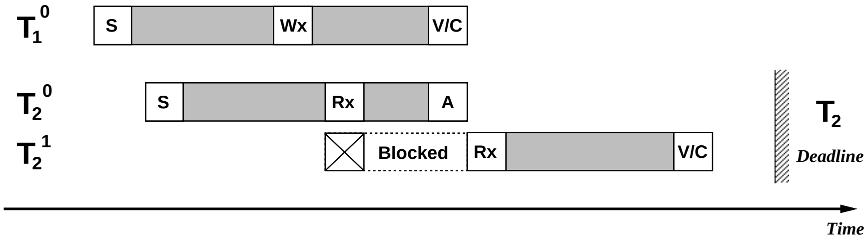

Figure 4: Schedule with a standby shadow promotion.

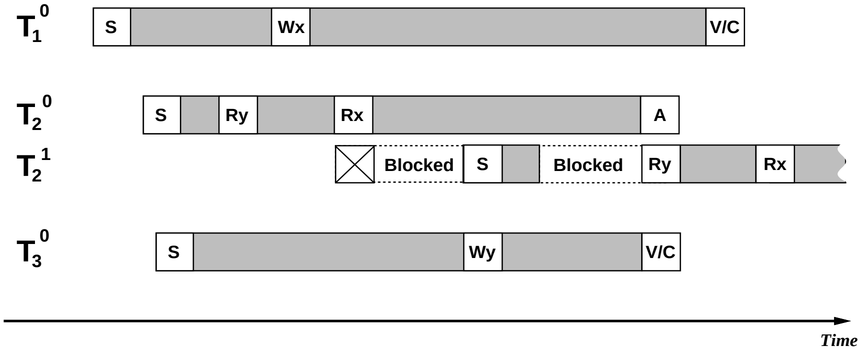

Figure 5: Schedule with a standby shadow restart and promotion.

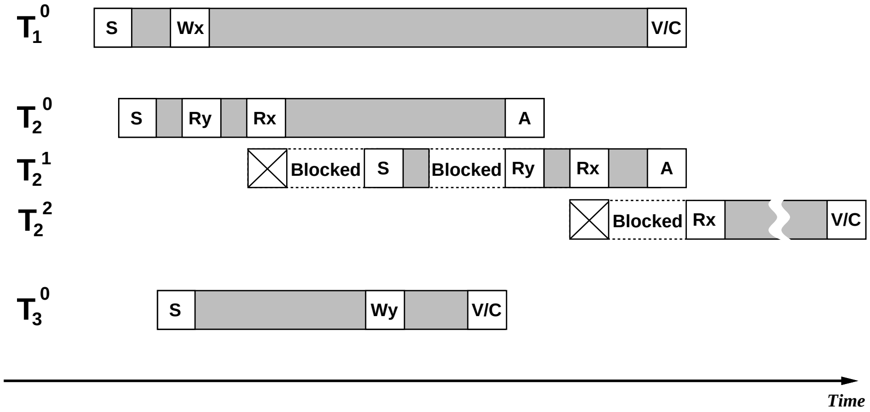

Figure 6: Schedule with two standby shadows.

We have conducted a number of experiments to compare the performance of SCC-based and OCC-based algorithms. Our simulations assume a client-server model in a distributed database subjected to *soft* deadlines. Figure 7-a depicts the total number of missed deadlines as a function of the total number of transactions submitted to the system. The simulation shows that SCC-2S is consistently better than OCC-BC by about a factor of 4 in terms of the number of transactions commited before their set deadlines. Figure 7-b depicts the tardiness[5] of the system as a function of the total number of transactions submitted to the system. Again, SCC-2S proves to be superior to OCC-BC as it reduces by almost 6-folds the tardiness of the system. In particular, with 25 transactions in the system, OCC-BC manages to commit only 3 transactions before their set deadlines, thus missing 22 deadlines with a tardiness of over 100 units of time. For the same schedule, SCC-2S manages to commit 13 transactions, missing the deadlines of only 12 transactions with a tardiness of 18 units of time. The above simulations assumed tight deadlines, which explains the high percentage of transactions missing their deadlines. Similar results confirming SCC-2S superiority were obtained for looser timing constraints, for *firm* deadlines, and for various levels of data conflicts. They are discussed in [Best93b].

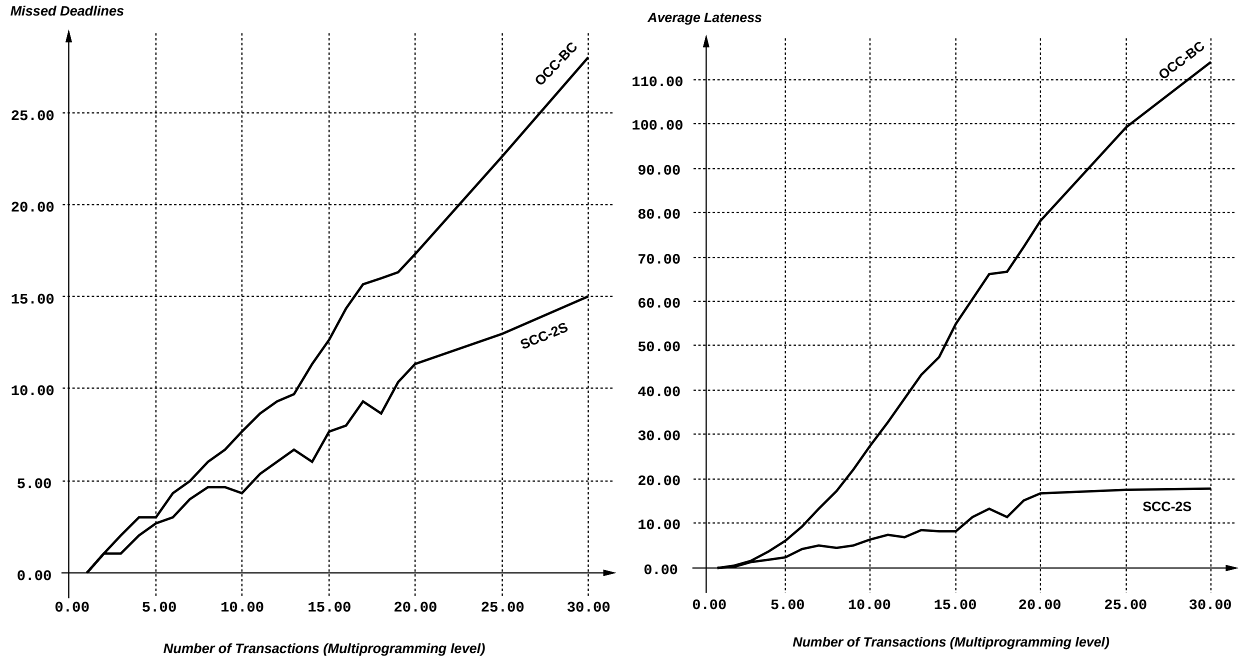

Figure 7: Simulation results for OCC-BC versus SCC-2S [(a) Left (b) Right]

---

[5] The tardiness of the system is the average time by which transactions miss their deadlines. A system that meets all imposed deadlines has an ideal tardiness of 0.

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.

[Best92a] Azer Bestavros. "Speculative concurrency control: A position statement." Technical Report TR-92-016, Computer Science Department, Boston University, Boston, MA, July 1992. Submitted for publication.

[Best92b] Azer Bestavros and Spyridon Braoudakis. "A family of speculative concurrency control algorithms." Technical Report TR-92-017, Computer Science Department, Boston University, Boston, MA, July 1992. Also submitted for publication to SIGMOD'93.

[Best93a] Azer Bestavros. "Speculative concurrency control for real-time databases." Technical Report TR-93-002, Computer Science Department, Boston University, Boston, MA, July 1993. Submitted for publication.

[Best93b] Azer Bestavros, Spyridon Braoudakis, and Euthimios Panagos. "Performance evaluation of two-shadow speculative concurrency control in a client-server distributed system." Technical Report TR-93-001, Computer Science Department, Boston University, Boston, MA, January 1993. Also submitted for publication to VLDB'93, Ireland.

[Elma89] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company Inc., 1989.

[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.

[Huan91] Jiandong Huang, John A. Stankovic, and Don Towslwy Krithi Ramamritham. "Experimental evaluation of real-time optimistic concurrency control schemes." In *Prooceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Spain, September 1991.

[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.

[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.

[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.