Speculative Concurrency Control

Azer Bestavros
(best@cs.bu.edu)

Computer Science Department
Boston University
Boston, MA 02215

January 27, 1993

Abstract

In this paper, we propose a new class of Concurrency Control Algorithms that is especially suited for real-time database applications. Our approach relies on the use of (potentially) redundant computations to ensure that serializable schedules are found and executed as early as possible, thus, increasing the chances of a timely commitment of transactions with strict timing constraints. Due to its nature, we term our concurrency control algorithms *Speculative*. The aforementioned description encompasses many algorithms that we call collectively Speculative Concurrency Control (SCC) algorithms.

SCC algorithms combine the advantages of both Pessimistic and Optimistic Concurrency Control (PCC and OCC) algorithms, while avoiding their disadvantages. On the one hand, SCC resembles PCC in that conflicts are detected as early as possible, thus making alternative schedules available in a timely fashion in case they are 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.

1 Introduction

In order for multiple transactions to operate concurrently on a shared database, a protocol must be adopted to coordinate their activities. Such a protocol – called a *concurrency control algorithm* – aims at insuring a *consistent* state of the database system, while allowing the maximum possible *concurrency* among transactions [Elma89].

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 [Best92a]. 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,[1] OCC outperforms 2PL under low system loads and high resource availability.

In this paper 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.

[1] A transaction with a firm deadline is discarded if it misses its deadline.

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. The SCC algorithms we are proposing in this paper represent a host of choices in terms of the required amount of redundant computations. We show that these algorithms are superior to any existing real-time concurrency control algorithms, even in the absence of any *spare* computing resources.

The remainder of this paper is organized as follows. In section 2, we review some of the previous work done in concurrency control for RTDBMS and provide the motivation for our research direction. In section 3, we overview the basic idea of SCC-based algorithms and present particularly interesting classes of SCC algorithms that differ mainly in the amount of redundant computations they tolerate. Finally, in section 4, we conclude this paper and describe our current and future research directions.

2 Previous Work

For a conventional DBMS with limited resources, performance studies of concurrency control methods (*e.g.* [Agra87]) have concluded that PCC locking protocols, due to their conservation of resources, perform better than OCC techniques. The main reason for this good performance is that PCC's blocking-based conflict resolution policy results in resource conservation, whereas OCC with its restart-based conflict resolution policy wastes more resources.

In an environment with an abundance of resources, the advantage that PCC blocking-based algorithms have over OCC restart-based algorithms vanishes. In particular, under such conditions, OCC algorithms become attractive since computing resources wasted due to restarts do not adversely affect performance.

Haritsa *et al.* [Hari90b, Hari90a] investigated the behavior of both PCC and OCC schemes in a real-time environment. The study showed that for a RTDBMS with firm deadlines (where late transactions are immediately discarded) OCC outperforms PCC, especially when resource contention is low. The key result of this study is that, if low resource utilization is acceptable (*i.e.* a large amount of wasted resources can be tolerated) and there is a large number of transactions available to execute, then a restart-oriented algorithm that allows a higher degree of concurrent execution becomes a better choice.

With classical OCC [Kung81], the execution of a transaction consists of three phases: *read*, *validation*, and *write*. The key component in OCC algorithms is the validation phase where the transaction's fate is determined. A transaction is allowed to execute unhindered (during its read phase) until it reaches its commit point, at which time a validation test is applied. This test checks that there are no conflicts between the actions of the transaction being validated and those of any other committed transaction. A transaction is restarted at its commit point if it fails its validation test, otherwise it commits by going through its write phase, in which modifications to the database (updates or writes performed by the transaction during its read phase) are made visible. One disadvantage of this basic OCC scheme is that when a conflict is detected the transaction being validated is always the one to be aborted. In RTDBMS, however, we want conflicts to be resolved according to the priority associated with the real-time transactions. Thus, more flexibility for conflict resolution is needed.

An even more serious problem of classical OCC, which may have a negative impact on the number of timing constraint violations, is that conflicts are not detected until the validation phase, at which time it might be too late to restart. PCC two-phase locking algorithms do not suffer from this problem because they detect potential conflicts as they occur. They suffer, however, from the possibility of unnecessarily missing set deadlines as a result of unbounded waiting due to blocking.

The Broadcast Commit variant (OCC-BC) of classical OCC [Mena82, Robi82] remedies this problem partially. When a transaction commits, it notifies those concurrently running transactions that conflict with it. Those transactions are immediately restarted. Note that there is no need to check for conflicts with already committed transactions since any such transaction would have, in the event of a conflict, restarted the validating transaction at its (the committed transaction's) own earlier commit time. This also means that the validating transaction is always guaranteed to commit. The broadcast commit method detects conflicts earlier than the basic OCC algorithm resulting in less wasted resources and earlier restarts.

To better illustrate this point, consider the following example. Assume that we have 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. Adopting the basic OCC algorithm means restarting transaction T_2 when it enters its validation phase because it conflicts with the already committed transaction T_1 on data item x. This scenario is illustrated in figure 1. Obviously, the likelihood of the restarted transaction T_2 meeting its timing constraint decreases.

In the example illustrated in figure 1, restarting T_2 after reaching its validation phase is wasteful of resources and – more importantly in real-time applications – it is wasteful of irrecoverable time! It is important to notice that the conflict between T_1 and T_2 developped when T_2 performed the read operation on x. This conflict, however, became prohibitive of both T_1 and T_2 committing their actions when T_1 was allowed to commit. In other words, before the commitment of T_1 the conflict over x was only a *potential* consistency threat. It *materialized* when T_1 was allowed to commit.

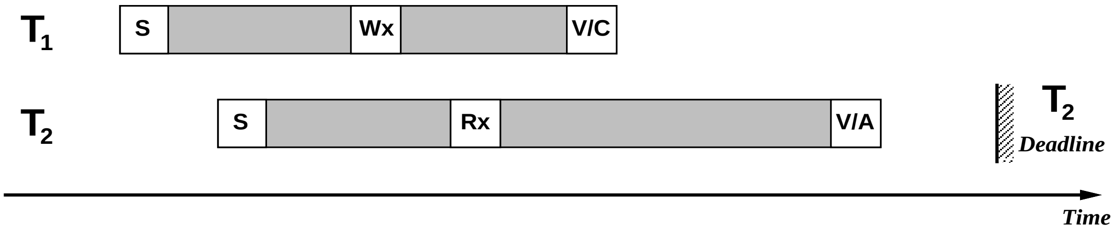

Figure 1: Transaction management under the basic OCC algorithm.

The OCC-BC algorithm avoids waiting unnecessarily for a transaction's validation phase in order to restart it. In particular, a transaction is aborted if any of its conflicts with other transactions in the system becomes a materialized consistency threat. This is illustrated in figure 2.

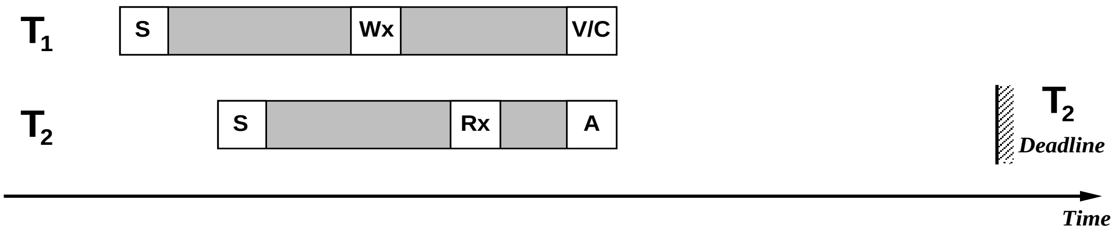

Figure 2: Transaction management under the OCC-BC algorithm.

The SCC aproach we are proposing in this paper 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, we use redundant resources to start on *speculative* corrective measures as soon as the conflict in question develops. By starting on such corrective measures as early as possible, we argue that the likelihood of meeting any set timing constraints will be greatly enhanced.

As we have hinted before, the underlying assumption in the rest of this paper is that the RTDBMS is operating with an abundance of computing resources. In other words, utilization is not an important performance parameter. Instead, we evaluate performance based on the timely commitment of transactions. In a real-time environment, both our abundant resources assumption and our performance measure seem appropriate.

3 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 the example of figures 1 and 2 once more. 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 3 and figure 4 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.

In figure 3 T_2 reaches its validation phase before T_1. Thus, T_2 will be validated[2] 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.

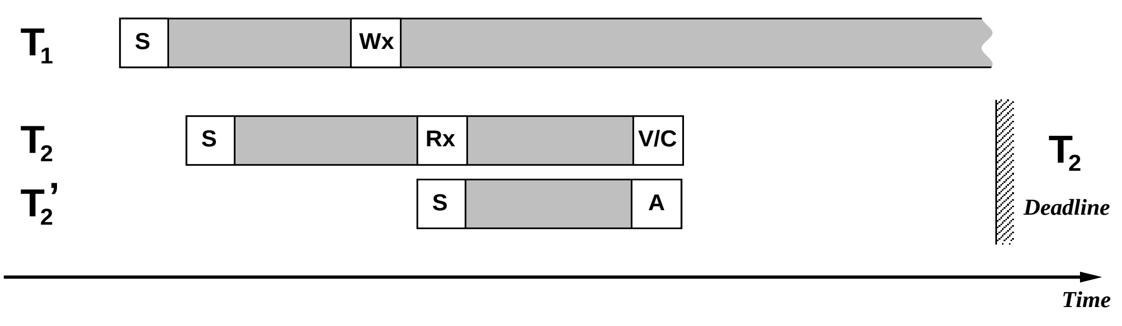

Figure 3: Schedule with an undeveloped potential conflict.

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,

[2]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).

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 4.

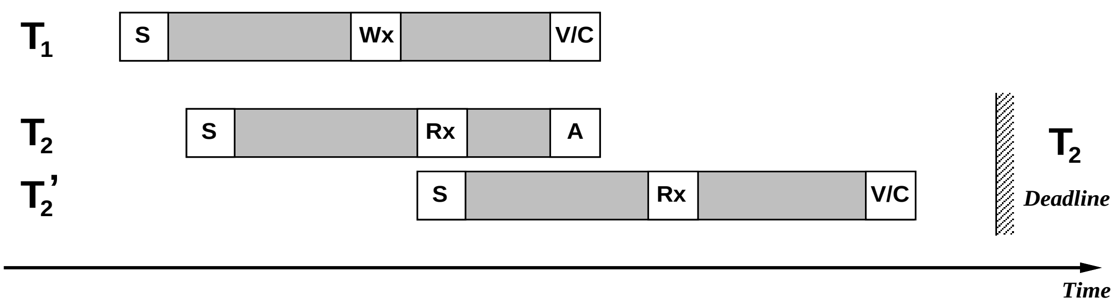

Figure 4: Schedule with a developed conflict.

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.

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 4). This, however, is the same price that OCC and OCC-BC protocols would have had to incur anyway (see figure 2).[3] On the other hand, as we depicted in figure 3, 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.

In the remainder of this section, we consider two SCC-based algorithms. The first (SCC-basic) represents the most general description of a particular class of SCC algorithms, whereas the second (SCC-2S) represents a specialization of SCC-basic that uses the minimum possible amount of redundancy. SCC-basic and SCC-2S represent the two extremes of a family of algorithms, each corresponding to a particular level of *computation redundancy* and *real-time performance*.

3.1 SCC Basic Algorithm

In this section we present the most general SCC algorithm. Despite its impracticality (in terms of the amount of redundancy it requires), this algorithm will serve as a reference point for all SCC-based algorithms.

A transaction T_i consists of a sequence of actions $a_1, a_2, \ldots a_n$, where each a_i, $i = 1, 2, \ldots n$, is either a read or a write operation on one of the shared objects of the database. Each transaction in the system is assumed to preserve the consistency of the shared database. Therefore, *any* sequential (or serializable) execution of any collection of transactions will also preserve consistency [Bern87].

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

Given a concurrent execution of transactions, action a_{ir} of transaction T_i conflicts with action a_{js} of T_j, if they access the same object and either a_{ir} is a read operation and a_{js} is a write operation (*read-write* conflict), or a_{ir} is a write operation and a_{js} is a read operation (*write-read* conflict).

Let $T = T_1, T_2, T_3, \ldots, T_m$ be the set of uncommitted transactions in the system. As we have hinted before, our technique relies on allowing several processes to concurrently execute on behalf of the same transaction. Each one of these processes corresponds to a different *speculated serialization order*. For a transaction T_r, we call each one of these processes a *shadow* of T_r. We associate with each shadow T_r^i of a transaction T_r a relation $\Psi(T_r^i) \subseteq T \times T$, such that $(T_u, T_v) \in \Psi(T_r^i)$ if the speculated serialization order for T_r^i implies that T_u commits *before* T_v. We call this set the *Speculated Order of Serialization* (SOS). We denote by $\Psi(T_r^i)^*$ the transitive closure of $\Psi(T_r^i)$. The description of the basic SCC algorithm follows.

a. When the execution of a new transaction T_r is requested, a shadow T_r^0 is created such that $\Psi(T_r^0) = \phi$.

b. Whenever a shadow T_r^i wishes to write an object that has been read by another shadow T_s^j, then:

   1. If $(T_r, T_s) \in \Psi(T_s^j)^*$ then T_s^j is simply restarted without changing $\Psi(T_s^j)$, otherwise

   2. A new shadow T_s^x for T_s is started, where $\Psi(T_s^x) = \Psi(T_r^i) \cup (T_r, T_s)$.

c. Whenever a shadow T_r^i wishes to read an object that has been written by a one of the shadows of a transaction T_s, then:

   1. If $(T_s, T_r) \in \Psi(T_r^i)^*$ then T_r^i must block waiting for the commitment of T_s, otherwise

   2. A new shadow T_r^y for T_r is forked, where $\Psi(T_r^y) = \Psi(T_r^i) \cup (T_s, T_r)$. T_r^i is allowed to proceed, whereas T_r^y must block waiting for the commitment of T_s.

d. Whenever it is decided to commit a shadow T_r^i on behalf of transaction T_r, then any other shadow of transaction T_r is discarded. In addition any shadow T_s^j, for which $\Psi(T_r^i) \cup \Psi(T_s^j)$ is not a partially ordered set.

Theorem 1 *The SCC-basic algorithm guarantees serializability.* [proof by induction]

Theorem 2 *The SCC-basic algorithm is deadlock-free.* [proof by induction]

3.2 Two-Shadow SCC Algorithm

The SCC-basic algorithm we have presented in section 3.1 allows a large number of shadows for each uncommitted transaction in the system to co-exist. Each one of these shadows assumes a different serialization order. In this section, we present another SCC-based algoritm (SCC-2S), which can be thought of as a special case of SCC-basic. In particular, it 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 5. 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.[4] 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.

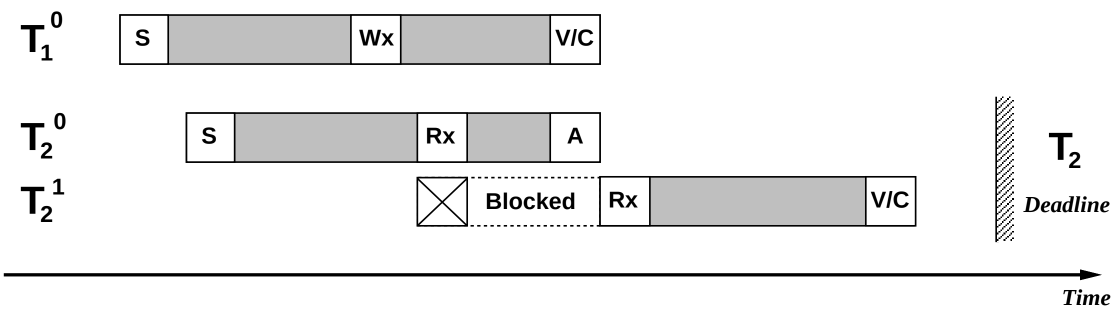

Figure 5: Schedule with a standby shadow promotion.

It is possible that multiple conflicts develop between executing transactions. Figure 6 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,[5] T_2^1 is promoted to become the primary shadow and is, thus, allowed to resume.

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

[5]Or as a result of its conflict with T_1 (as was the case in figure 5).

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 7 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.

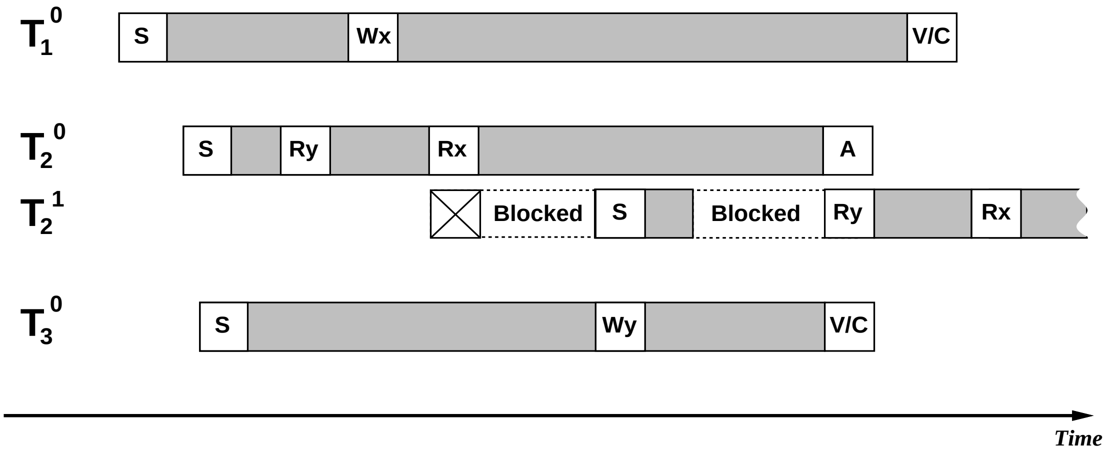

Figure 6: Schedule with a standby shadow restart and promotion.

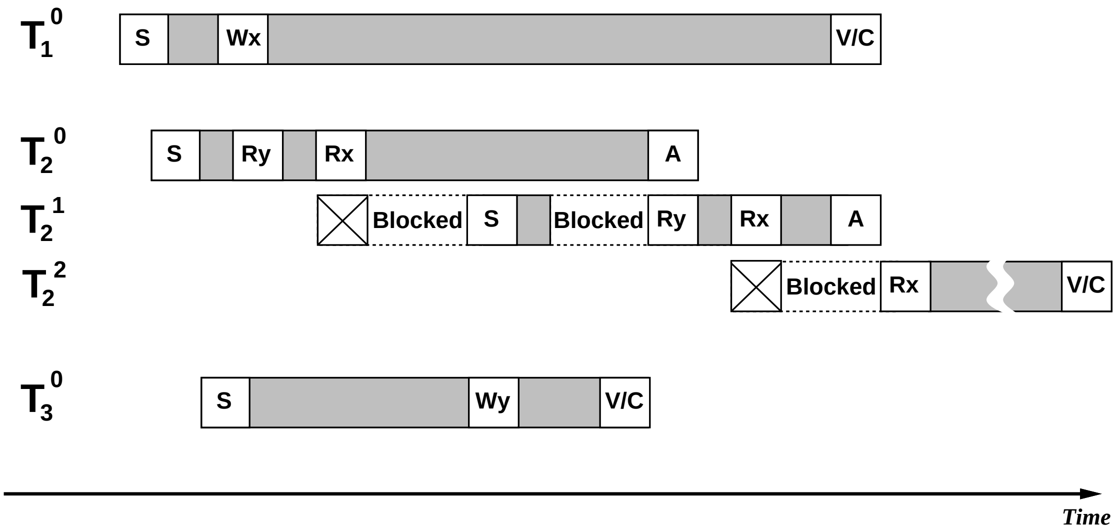

Figure 7: Schedule with two standby shadows.

Let $T = T_1, T_2, T_3, \ldots, T_m$ be the set of uncommitted transactions in the system. Furthermore, let $T^{primary}$ and $T^{standby}$ be the sets of primary and standby shadows executing on behalf of the transaction set T, respectively. For each standby shadow T_s^i in the system, we maintain a set

$WaitFor(T_s^i)$, which contains a list of tuples of the form (T_r, X), where $T_r \in T$ and X is an object of the shared database. $(T_r, X) \in WaitFor(T_s^i)$ implies that T_s^i must wait for T_r before being allowed to read object X. We use the notation $(T_r, \_) \in WaitFor(T_s^i)$ to imply that there exists at least one tuple $(T_r, X) \in WaitFor(T_s^i)$, for some object X. The details of the SCC-2S algorithm follow:

a. When the execution of a new transaction T_r is requested, a primary shadow $T_r^0 \in T^{primary}$ is created and executed.

b. Whenever a primary shadow T_r^i wishes to read an object X that has been written by another shadow T_s^j, then:

   1. If there is no standby shadow for T_r, then a new shadow T_r^{i+1} for T_r is forked off, such that $WaitFor(T_r^{i+1}) = \{(T_s, X)\}$, otherwise

   2. Let T_r^k be the standby shadow executing on behalf of T_r. If $(T_s, X) \notin WaitFor(T_r^k)$, then $WaitFor(T_r^k) = WaitFor(T_r^k) \cup \{(T_s, X)\}$.

c. Whenever a primary shadow T_r^i wishes to write an object Y that has been read by another shadow T_s^j, then:

   1. If there is no standby shadow for T_s, then a new shadow T_s^{j+1} for T_s is created and executed, such that $WaitFor(T_s^{j+1}) = \{(T_r, Y)\}$, otherwise

   2. Let T_s^k be the standby shadow executing on behalf of T_s. If $(T_r, Y) \notin WaitFor(T_s^k)$, then T_s^k is aborted and a new standby shadow T_s^{k+1} is started with $WaitFor(T_s^{k+1}) = WaitFor(T_s^k) \cup \{(T_r, Y)\}$.

d. A standby shadow T_r^i is blocked whenever it wishes to read any object that has been written on behalf of any of the transactions in $WaitFor(T_r^i)$.

e. Whenever it is decided to commit a primary shadow T_r^i on behalf of transaction T_r, then

   1. If $(T_r, \_) \in WaitFor(T_s^i)$ then the primary shadow of T_s is aborted, T_s^i is promoted to become a primary shadow of T_s, and a new backup shadow T_s^{i+1} is forked off T_s^i, such that $WaitFor(T_s^{i+1}) = WaitFor(T_s^i) - \{(T_r, \_)\}$.

   2. Any standby shadow of T_r is aborted.

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 system. Figure 8 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 9 depicts the tardiness[6] 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

[6]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.

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 and various levels of data conflicts. They are discussed in [Best93].

3.3 Other SCC-based Algorithms

The SCC-basic algorithm and the SCC-2S algorithm represent two extremes regarding the amount of redundant computations they introduce. In the presence of M uncommitted transactions in the system, SCC-basic allows at most one shadow per M-tupple orderings (an upper bound of $M!$ shadows per transaction), whereas SCC-2S allows at most one shadow per 2-tupple orderings (an upper bound of 2! shadows per transaction. It is conceivable to think about other alternatives, in which instead of considering M-tupple orderings or 2-tupple orderings, one could consider N-tupple orderings, $1 \leq N \leq M$. In [Best92b] we present a generic SCC-based algorithm, which allows the redundancy level for the individual transactions in the system to be different and to vary dynamically. In [Best92c], we use this feature to express the priority (or urgency) of transactions in real-time databases.

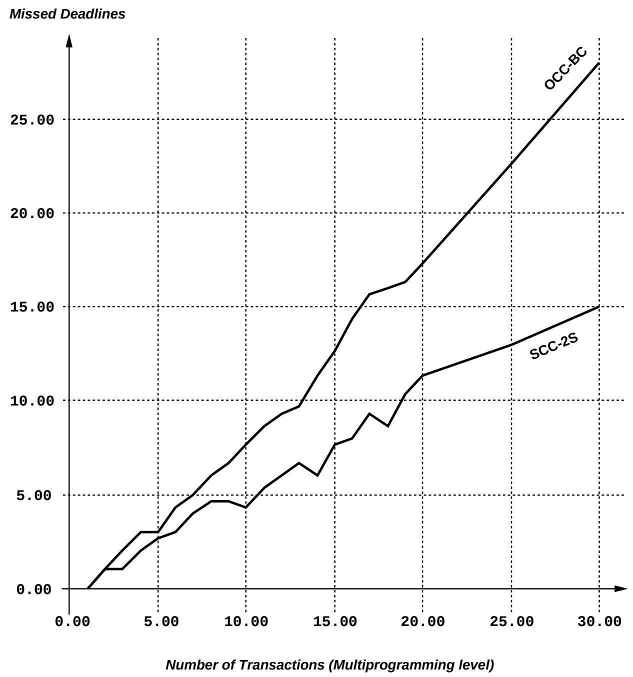

Figure 8: Simulation results for OCC-BC versus SCC-2S (Number of satisfied deadlines)

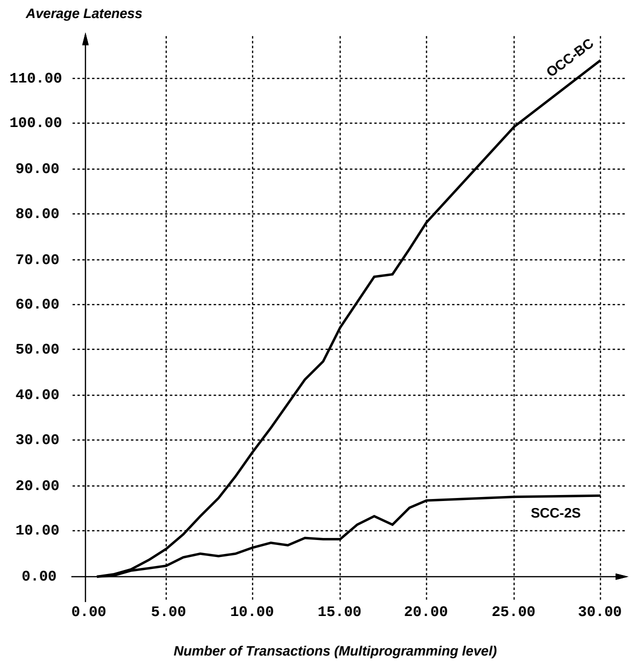

Figure 9: Simulation results for OCC-BC versus SCC-2S (System Tardiness)

4 Conclusion

SCC-based algorithms offer a new dimension (namely redundancy) that can be used effectively in RTDBMS. In this paper, we introduced the basic idea behind SCC algorithms. Many interesting research problems remain to be tackled. In particular, performance metrics suitable for evaluating RTDBMS must be developped. These metrics must reflect how successful a concurrency control algorithm is viz a viz meeting the time constraints (whether soft or hard) imposed on the transactions submitted to the system.

Implementation issues pertinent to SCC-based algorithms must be addressed. In particular, centralized vs. distributed implementations of SCC-based algorithms must be investigated. We are particularly interested in exploiting parallel computing platforms. Also, the fault-tolerance aspects (and potentials) of SCC-based algorithms must be fully examined.

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.

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

[Bern87] A. Bernstein, A. Philip, V. Hadzilacos, and N. Goodman. *Concurrency Control And Recovery In Database Systems*. Addison-Wesley, 1987.

[Best92a] Azer Bestavros. "Performance measures for real-time database management systems." Technical Report (In progress), Computer Science Department, Boston University, Boston, MA, July 1992.

[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.

[Best92c] Azer Bestavros and Spyridon Braoudakis. "Speculative concurrency control algorithms for real-time databases: An alternative expression of transaction priority." Technical Report (In progress), Computer Science Department, Boston University, Boston, MA, September 1992.

[Best93] 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.

[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.