A Family of Speculative Concurrency Control Algorithms for Real-Time Databases[*]

AZER BESTAVROS SPYRIDON BRAOUDAKIS
(best@cs.bu.edu) (sb@cs.bu.edu)
Computer Science Department
Boston University
Boston, MA 02215

Abstract

Speculative Concurrency Control (SCC) is a new concurrency control approach, especially suited for real-time databases [4]. SCC uses redundancy to ensure that serializable executions are discovered and adopted as early as possible, thus increasing the likelihood of the timely commitment of transactions with strict timing constraints. We present SCC-nS, a generic algorithm that characterizes a family of SCC-based algorithms. Under SCC-nS, shadows executing on behalf of a transaction are either optimistic *or* speculative. *Optimistic shadows execute under an assumed serialization order, which requires them to wait for* no *other conflicting transactions. They execute unhindered until they are either aborted or committed. Alternately, speculative shadows execute under an assumed serialization order, which requires them to wait for* some *conflicting transactions to commit.*

1 Introduction

Traditional concurrency control algorithms can be broadly classified as either *pessimistic* or *optimistic*. Pessimistic Concurrency Control (PCC) algorithms [9, 10] avoid any concurrent execution of transactions as soon as *potential* conflicts between these transactions are detected. On the contrary, Optimistic Concurrency Control (OCC) algorithms [7, 17] allow such transactions to proceed at the risk of having to restart them in case these suspected conflicts *materialize*.

For Real-Time DataBase Management Systems (RTDBMS) where transactions execute under strict timing constraints, maximum concurrency (or throughput) ceases to be an expressive measure of performance. Rather, the number of timely-commited transactions becomes the decisive performance measure [8]. Most real-time concurrency control schemes considered in the literature [1, 2, 28, 13, 26, 24, 25] are based on Two-Phase Locking (2PL), which is a PCC strategy. Despite its widespread use in commercial systems, 2PL's long and unpredictable blocking times damage its appeal for real-time environments, where the primary performance criterion is meeting time constraints and not just preserving consistency requirements. Over the last few years, several alternatives to 2PL for RTDBMS have been explored [16, 12, 11, 14, 15, 18, 27].

In a recent study [4], Bestavros proposed a categorically different approach to concurrency control for RTDBMS. His approach relies on the use of redundant computation to start on alternative schedules, once conflicts that threaten the consistency of the database are detected. These alternative schedules are adopted *only if* the suspected inconsistencies materialize; otherwise, they are abandoned. Due to its nature, this approach has been termed *Speculative Concurrency Control* (SCC). This paper examines a family of SCC algorithms and their implementations.

SCC algorithms use redundancy to 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 timing constraints – should these alternative schedules be needed (due to restart as in OCC). On the other hand, SCC resembles OCC in that it allows conflicting transactions to proceed concurrently, thus avoiding unnecessary delays (due to blocking as in PCC) that may jeopardize their timely commitment.

The remainder of this paper is organized as follows. In section 2, we review some of the problems encountered with traditional concurrency control in RTDBMS, and we overview the basic idea behind the SCC-based approach. In section 3, SCC-nS, a generic SCC algorithm is described, and its superiority for real-time database applications is demonstrated. In section 4, three members of the SCC-nS family (namely SCC-1S, SCC-2S, and SCC-MS) are singled out and contrasted. In section 5, we conclude with a description of our current and future research work.

---

[*] This work has been partially supported by GTE Labs.

2 Concurrency Control for RTDBMS

A disadvantage of classical OCC when used in RTDBMS is that transaction conflicts are not detected until the validation phase, at which time it might be too late to restart. This may have a negative impact on the number of timing constraint violations. PCC two-phase locking algorithms do not suffer from this problem because they detect potential conflicts as they occur.

The Broadcast Commit variant (OCC-BC) [19, 22] of the classical OCC remedies this problem partially. When a transaction commits, it notifies all concurrently running, conflicting transactions about its commitment. All those conflicting transactions are immediately restarted. The broadcast commit method detects conflicts earlier than the basic OCC algorithm resulting in less wasted resources and earlier restarts.

To 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 considerably.

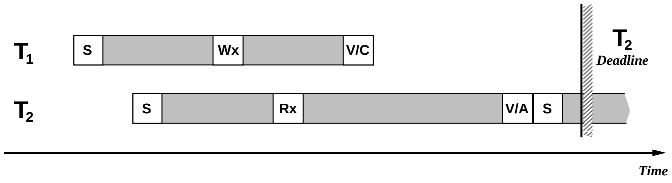

Figure 1: Transaction management under basic OCC.

The OCC-BC algorithm avoids waiting unnecessarily for a transaction's validation phase in order to restart it. 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.

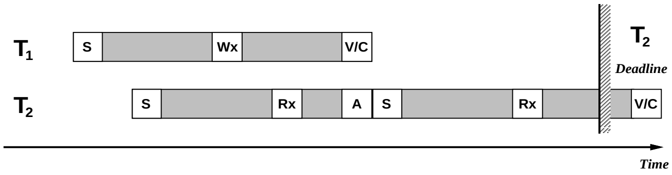

Figure 2: Transaction management under OCC-BC.

The SCC-based Approach:

The SCC approach proposed in [4] 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, an SCC algorithm uses 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, the likelihood of meeting any set timing constraints will be greatly enhanced. 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. In figure 3, T_2 reaches its validation phase before T_1. T_2 will be validated and committed without any need to disturb T_1. 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.

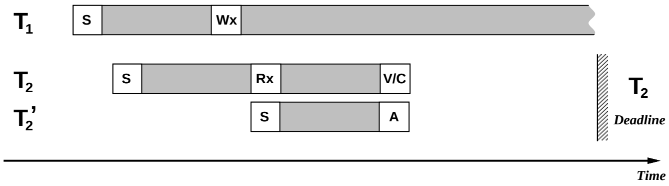

Figure 3: An undeveloped potential conflict.

However, if transaction T_1 reaches its validation phase first, then transaction T_2 cannot continue to execute due to the 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 close. The SCC protocol (see figure 4), instead of restarting T_2, simply aborts T_2 and adopt its shadow transaction T_2'.

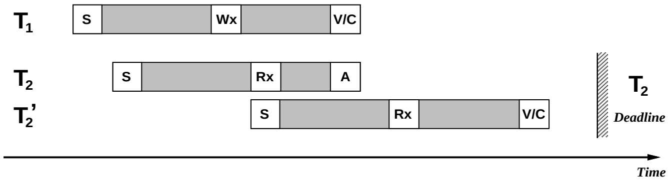

Figure 4: A developed conflict.

3 A Generic SCC-nS Algorithm

In this section, we present a class of SCC algorithms that operate under a *limited redundancy* assumption. In particular, we present a generic SCC algorithm which does not allow more than n shadows to execute on behalf of any given uncommitted transaction in the system.

3.1 Preliminaries

A transaction T_i consists of a sequence of actions $a_{i1}, a_{i2}, \ldots a_{im}$, where each a_{ij}, $j = 1, 2, \ldots m$, 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 these shared objects. Therefore, *any* sequential (or serializable) execution of any collection of transactions will also preserve the consistency of the database [20, 3].

Write operations are performed on private data copies in the local workspace of transactions instead on the shared database objects directly. They will be made permanent in the shared database only during the transactions commit time. Each transaction T_i has its own local workspace, where updates are being performed. Subsequent read operations by T_i on previously updated database objects retrieve the value from its local workspace. Any other transaction is not aware of this value, since it other reads directly from the database, or from its own local workspace.

Given a concurrent execution of transactions, action a_{ir} of transaction T_i conflicts with action a_{js} of transaction 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).

Write-write conflicts (when both a_{ir} and a_{js} actions are write operations) are treated using the Thomas' Write Rule (TWR). At commit time, when all database updates are made permanent, all write requests are buffered by the data manager and serialized according to their transaction order. A timestamp is being assigned to every committing transaction for that purpose. With the TWR every write request arriving out of timestamp order (late) is being *ignored* rather than being *rejected* [3]. In other words, all write requests are granted, whether or not the targeted data object is being updated by another uncommitted transaction.

As we have hinted before, SCC-based algorithms allow several shadows (processes or tasks) to execute concurrently on behalf of the same transaction. Each one of these processes corresponds to a different *speculated serialization order*. For a transaction T_r, each one of these processes is called a *shadow* of T_r. In this paper, a shadow can be in one of two modes: *optimistic* or *speculative*. Each transaction T_r has, at any point in its execution, exactly one optimistic shadow T_r^o. In addition, T_r may have i speculative shadows T_r^i, for $i = 0, \ldots, n-1$. Accordingly, each transaction can have *at most* n shadows executing on its behalf at any point in its lifetime.

One point that we should make here is that only the reader transactions need to be shadowed. Because of the forward validation method adopted in our protocol, validation is done only against active transactions. All conflicting transactions are notified of their data access conflicts and are aborted immediately. It follows that to ensure serializability we must check that the ReadSets of all active transactions do not intersect with the WriteSet of the transaction being validated. Thus, only transactions that perform read operations are in danger of being aborted and need to be shadowed.

For each transaction T_r we keep a variable $\mathit{SpecNumber}(T_r)$, which counts the number of the speculative shadows currently executing on behalf of T_r. With each shadow T_r^i of a transaction T_r – whether optimistic, or speculative – we maintain two sets: $\mathit{ReadSet}(T_r^i)$ and $\mathit{WriteSet}(T_r^i)$. $\mathit{ReadSet}(T_r^i)$ records pairs (X, t_x), where X is an object read by T_r^i, and t_x represents the order[1] in which this operation was performed. We use the notation: $(X, \_) \in \mathit{ReadSet}(T_r^i)$ to mean that shadow T_r^i read object X. $\mathit{WriteSet}(T_r^i)$ contains a list of all objects X written by shadow T_r^i.

For each speculative shadow T_r^i in the system, we maintain a set $\mathit{WaitFor}(T_r^i)$, which contains pairs of the form (T_u, X), where T_u is an uncommitted transaction and X is an object of the shared database. $(T_u, X) \in \mathit{WaitFor}(T_r^i)$ implies that T_r^i must wait for T_u before being allowed to Read object X. We use $(T_u, \_) \in \mathit{WaitFor}(T_r^i)$ to denote the existence of at least one tuple (T_u, X) in $\mathit{WaitFor}(T_r^i)$, for some object X.

3.2 Algorithm Overview

Under the SCC-nS algorithm, shadows executing on behalf of a transaction are either *optimistic* or *speculative*. Optimistic shadows execute unhindered, whereas speculative shadows are maintained so as to be ready to replace a defunct optimistic shadow, if such a replacement is deemed necessary.

Optimistic shadow behavior:

For a transaction T_r, the optimistic shadow T_r^o executes with the *optimistic* assumption that it will commit *before* all the other uncommitted transactions in the system with which it conflicts. T_r^o records any conflicts found during its execution, and proceeds uninterrupted until one of these conflicts materializes (due to the commitment of a competing transaction), in which case T_r^o is aborted – or else until its validation phase is reached, in which case T_r^o is committed.

Speculative shadow behavior:

Each speculative shadow T_r^s executes with the assumption that it will finish before the materialization of any detected conflict with any other uncommitted transaction, except for one particular conflict which is *speculated* to materialize before the commitment of T_r. Thus, T_r^s remains blocked on the shared object X, on which this conflict has developed, waiting to read the value that the conflicting transaction, T_u will assign to X when it commits. If this speculated assumption becomes true, (*e.g.*, T_u commits before T_r enters its validation phase), T_r^s will be unblocked and *promoted* to become T_r's optimistic shadow, replacing the old optimistic shadow which will have to be aborted, since it made the wrong assumption with respect to the serialization order.

At any point during the execution of our algorithm, the first k speculative shadows of a transaction T_r ac-

---

[1] This can be a special read timestamp, implemented by maintaining for each shadow T_r^i in the system a counter that is atomically incremented every time a read operation is performed by T_r^i.

count for the first k detected conflicts in which T_r participated. These may not be the first k conflicts that transaction T_r will develop during the course of its execution. To illustrate this point, consider the condition depicted in figure 5. Transaction T_1 may detect at some point in its execution a conflict over some object X, which it had read earlier. In particular, when the read operation for object X was requested by the optimistic shadow T_1^o, there was no conflict to be detected. Such a conflict appeared later when transaction T_3 requested to update that same object X.

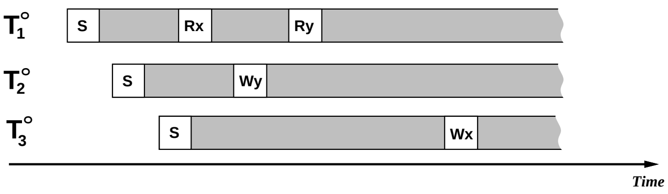

Figure 5: T_1 detects conflict (T_3, X) after T_3 writes X.

The *shadow replacement algorithm* we are using in this paper is one of several algorithms that could be adopted. In [5] some alternatives to this policy are discussed and evaluated. In particular, information about deadlines and priorities of the conflicting transactions can be utilized so as to account for the *most probable* serialization orders.

It is very important to realize that the imposed limit of at most $n-1$ speculative shadows per transaction does not prohibit a transaction T_r from developing more than $n-1$ conflicts at any point during its lifetime. Rather, this limit is on the number of potential hazards that our algorithm will be ready to *optimally* deal with (by using the speculative shadows). Every *extra* hazard that develops after this limit is reached will be accounted for only *suboptimally*[2] (since no such speculative shadow will be available). In that sense, we can view the aforementioned description as encompassing a hierarchy of algorithms. Going down a level in this hierarchy (by reducing n) can compromise only performance not correctness.

3.3 Description of SCC-nS

Let $\mathcal{T} = T_1, T_2, T_3, \ldots, T_m$ be the set of uncommitted transactions in the system. Furthermore, let $\mathcal{T}^{\mathcal{O}}$, and $\mathcal{T}^{\mathcal{S}}$ be, respectively the sets of optimistic, and speculative shadows executing on behalf of the transactions in the set $\mathcal{T}$. We use the notation $\mathcal{T}_r^{\mathcal{S}}$ to denote the set of speculative shadows executing on behalf of transaction T_r. The SCC-nS algorithm is described as a set of five rules, which we describe below.

Start Rule:
The *Start Rule*, is followed whenever a new transaction T_r is submitted for execution, in which case an optimistic shadow T_r^o is created. In the absence of any conflicts this shadow will run to completion (the same way as with the OCC-BC algorithm). The *SpecNumber*(T_r), *ReadSet*(T_r^o), and *WriteSet*(T_r^o), are, also, initialized.

Read Rule:
The *Read Rule* is activated whenever a read-after-write conflict is detected. The processing that follows is straightforward. In particular, if the maximum number of speculative shadows of the transaction in question, say T_r, is not exhausted, a new speculative shadow T_r^s is created (by forking it off T_r^o) to account for the newly detected conflict. Otherwise, in the absence of any new speculative shadow for transaction T_r, this potential conflict will have to be ignored at this point. The Commit Rule (see below) deals with the corrective measures that need to be taken, should this conflict materializes.

Write Rule:
The *Write Rule* is activated whenever a write-after-read conflict is detected. Speculative shadows cannot be forked off as before from the transaction's optimistic shadow. This is because the conflict is detected on some other transaction's write operation. Therefore, since its optimistic shadow already read that database object, we must either create a new copy of this transaction or choose another point during its execution from which we can fork it off. For performance reasons, this second choice was adopted. The algorithm makes use of the function *BestShadow* (discussed later) to find the most appropriate speculative shadow, if such a shadow indeed exists. In the absence of such a shadow a restarted copy of the transaction is created. Figure 6 illustrates this point. When the new conflict (T_2, X) is detected, the speculative shadow T_1^3 is forked off T_1^1 to accommodate it. Notice that if a copy of T_1 was instead created, all the operations before R_y (reading the database object Y) would have had to be repeated. T_1^2, even though in a later stage, is not an appropriate shadow to fork off because, like the optimistic shadow, it already read X.

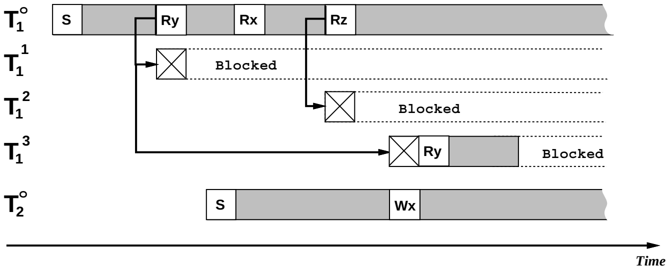

Figure 6: T_1^3 is forked off the *BestShadow* (T_1, X), T_1^1.

---

[2] We can still use the presense of other speculative shadows to improve those decisions (see the Commit Rule below).

Some interesting issues that must be dealt with in this case are discussed below. When the new conflict implicates transactions that already conflict with each other, some adjustments may be necessary. In figure 7, the speculative shadow T_1^j of transaction T_1, accounting for the conflict (T_2, Z), must be aborted as soon as the new conflict, (T_2, X), involving the same two transactions is detected. Since T_1 read object X before object Z, (T_2, X) is the *first* conflict between those two transactions. Therefore, the speculative shadow accounting for the possibility that transaction T_2 will commit before transaction T_1 must block before the read operation on X is performed. Speculative shadow T_1^k is forked off T_1^1 for that purpose. All other speculative shadows of T_1 remain unaffected.

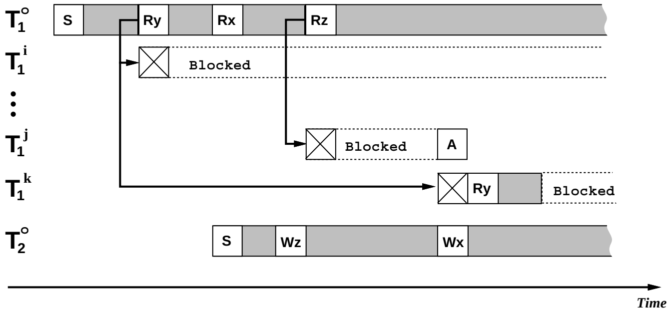

Figure 7: T_1^j, which accounts for the (T_2, Z) conflict, is aborted and replaced by T_1^k when an *earlier* conflict, (T_2, X), with T_2 is detected.

The number of speculative shadows maintained by SCC-nS (namely $n - 1$) might not be enough to account for all the conflicts that develop during a transaction's lifetime. The selection of the conflicts to be accounted for by speculative shadows is an interesting problem with many possible solutions [5]. In this paper we have adopted a particular solution that requires the speculative shadows of SCC-nS to account for the *first* $k \leq n - 1$ conflicts (whether read-after-write or write-after-read) encountered by a transaction. Because such conflicts are not necessarily detected *in order*, a *shadow replacement* might be necessary.

To illustrate this point, consider the scenario depicted in figure 8, where the assumption that the first two conflicts in which transaction T_1 participated (by accessing objects Y, and Z, respectively), is revised when transaction T_2 writes object X. In particular, the newly detected conflict (T_2, X) becomes the first conflict of T_1. If it is the case that T_1 is restricted so as not to have more that two speculative shadows at any point during its execution, then a shadow replacement is necessary. T_1^2, the *latest* shadow of T_1 has to be aborted, and a new speculative shadow, T_1^3, accounting for the new (T_2, X) conflict should replace it. The *LastShadow* function (explained below) is used to find this *latest* speculative shadow.

Blocking Rule:

The *Blocking Rule* is used to control when a speculative shadow T_r^i must be blocked. This rule assures that T_r^i is blocked the *first* time it wishes to read an object X in conflict with any transaction that T_r^i must wait for according to its speculated serialization order.

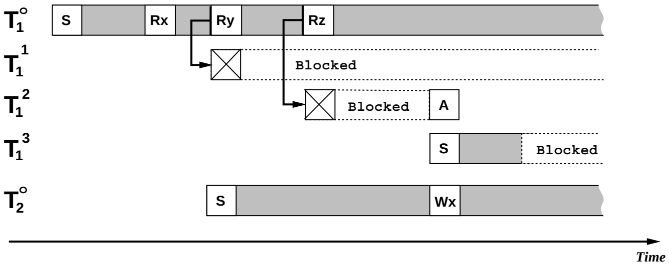

Figure 8: Detecting conflict (T_2, X) causes the abortion of *LastShadow*(T_1) (T_1^2), and its replacement by T_1^3.

Commit Rule:

Whenever it is decided to commit an optimistic shadow T_r^o on behalf of a transaction T_r, the *Commit Rule* is activated. First, all other shadows of T_r become obsolete and are aborted. Next, all transactions conflicting with T_r are considered. For each such transaction T_u there are two cases: either there is a speculative shadow, T_u^i, waiting for T_r's commitment, or not.

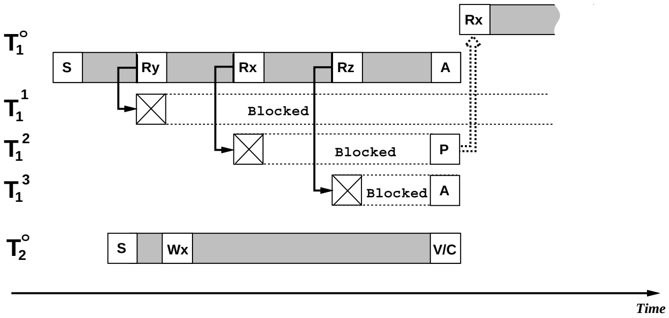

Figure 9: T_1^2, accounting for the developed conflict (T_2, X), is promoted to replace the optimistic shadow of T_1. T_1^3 is aborted, while T_1^1 remains unaffected.

The first case is illustrated in figure 9, where the speculative shadow T_1^2 of transaction T_1 – having anticipated (assumed) the correct serialization order – is promoted to become the new optimistic shadow of transaction T_1, replacing the old optimistic shadow which had to be aborted. Speculative shadow T_1^3, which like the old optimistic shadow exposed itself by reading the old value of object X had to be aborted as well. On the contrary, the speculative shadow T_1^1, which did not read object X, remains unhindered.

The second case is illustrated in figure 10, where the commitment of the optimistic shadow T_2^o on behalf of transaction T_2 was not accounted for by any speculative shadow.[3] In this case, a shadow is forked off the *LastShadow*(T_1) to become the new optimistic shadow of transaction T_1. This, even though not optimal, is the best we can do in the absence of a speculative shadow accounting for the (T_2, Z) conflict. A complete and formal description of the SCC-nS algorithm can be found in Appendix A.

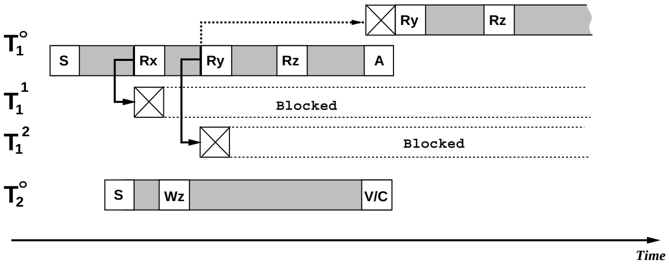

Figure 10: When the unaccounted-for conflict (T_2, Z) materializes, a new optimistic shadow for T_1 is forked off the *LastShadow*(T_1), T_1^2.

As we mentioned above, the algorithm makes use of two functions: *LastShadow*, and *BestShadow*. *LastShadow* is a function from the set of uncommitted transactions $\mathcal{T}$ to the set of speculative shadows $\mathcal{T}^{\mathcal{S}}$. It takes for input a transaction T_r, and returns the *latest* speculative shadow T_r^{last} of T_r in order of read conflict. *BestShadow* is a function from the cross-product of uncommitted transactions and database objects, to the set of speculative shadows $\mathcal{T}^{\mathcal{S}}$. It takes as input a transaction T_r and a database object X read by its optimistic shadow T_r^o. It returns the speculative shadow T_r^{best} of T_r, which did not read object X and accounts for the *latest* conflict (T_u, Y) in which T_r participates. Should such a speculative shadow does not exist, T_r^{best} corresponds to the starting point in the execution of T_r. Appendix B provides a formal definition of these functions.

3.4 Simulation Results

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 [21]. Figure 11 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 committed before their set deadlines. Figure 12 depicts the tardiness[4] 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, with a tardiness of 18 units of time. The above simulations assumed tight deadlines, which explains the high percentage of missed 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 [6].

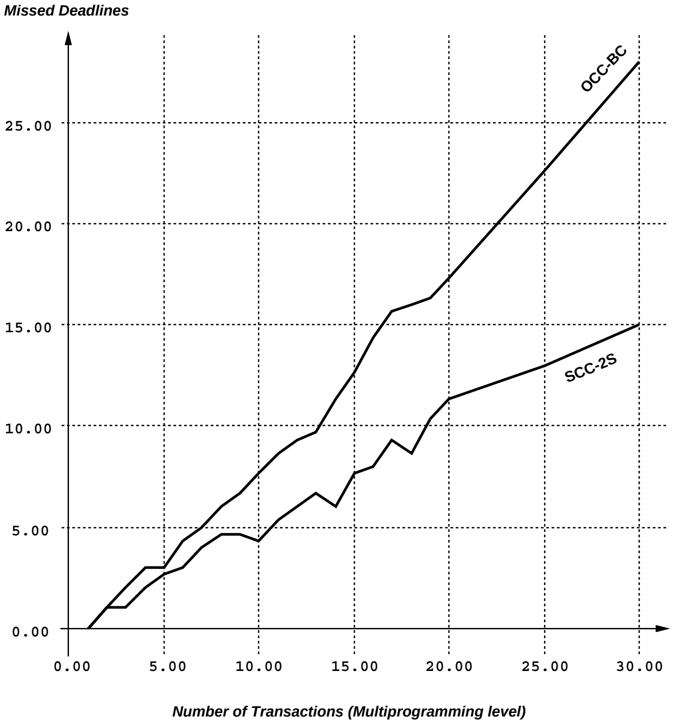

Figure 11: Missed deadlines for OCC-BC vs. SCC-2S

4 Three members of the SCC-nS family

In this section, we consider three SCC-based algorithms: SCC-1S, SCC-2S, and SCC-MS. The first represents a specialization of SCC-nS, which uses the minimum possible amount of redundancy. The second can be seen as the simplest form of a hybrid algorithm, allowing each transaction to have one optimistic and one pessimistic (speculative) shadow. The third represents the most flexible

---

[3] Figure 10 makes the implicit assumption that transaction T_1 is limited to having at most two speculative shadows at any point during its execution.

[4] 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.

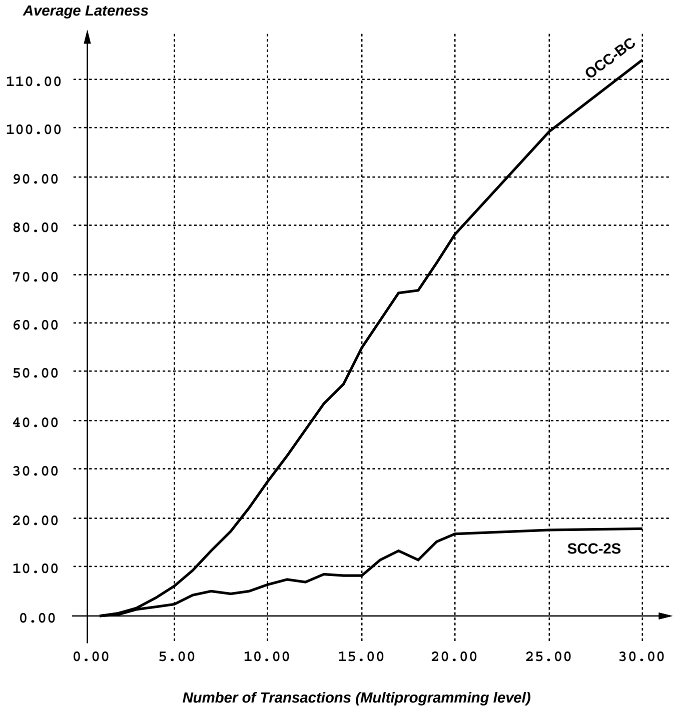

Figure 12: Average tardiness for OCC-BC vs. SCC-2S

of this family of SCC algorithms. SCC-MS and SCC-1S illustrate the two extremes with regard to the level of the *computation redundancy* they introduce and the *real-time performance* they achieve.

4.1 One-Shadow SCC

In this case, every uncommitted transaction in the system has only an optimistic shadow. Neither a speculative nor a pessimistic shadow is present. The optimistic shadow for each T_i, then, runs under the assumption that it will be the first (among all the other transactions with which T_i conflicts) to commit. Therefore, it executes without incurring any blocking delays. The SCC-1S algorithm, thus, resembles the OCC-BC algorithm in that optimistic shadows of transactions continue to execute either until they are validated and committed, or until they are aborted (by a validating transaction). This represents the one extreme regarding the amount of redundant computations that SCC algorithms introduce. At their lowest extent, when no redundant computations are allowed, they identify with the optimistic paradigm. The more redundancy they are allowed to use, the better their real-time performance.

4.2 Two-Shadow SCC (SCC-2S)

The SCC-2S allows a maximum of two shadows per uncommitted transaction to exist in the system at any point in time: an optimistic shadow and a speculative shadow. The speculative shadow of a transaction T_i, called here the *pessimistic* shadow T_i^p (in contrast with the optimistic shadow) is subject to blocking and restart. It is kept ready to replace the optimistic shadow T_i^o, should such a replacement be necessary. T_i^p 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 like the SCC-1S algorithm resembles the OCC-BC algorithm in that optimistic shadows of transactions continue to execute either until they are validated and committed or until they are aborted (by a validating transaction). The difference, however, is that SCC-2S keeps a *pessimistic* shadow for each executing transaction to be used if that transaction must abort. The pessimistic shadow is basically a replica of the optimistic 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 pessimistic shadow is promoted to become the new optimistic 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. The detailed algorithm, as well as illustrative examples of its use can be found in [4].

4.3 Multi-Shadow SCC (SCC-MS)

This is an SCC-based algorithm, which allows the redundancy level for individual transactions to differ and vary dynamically. Each transaction T_r has, at each point of its execution, one optimistic shadow T_r^o, and i speculative shadows T_r^i, where i is the number of detected potential conflicts in which T_r participates.

This variant is more powerful than the generic SCC algorithm presented above. Its superior performance results from its flexibility to deal with *any* transaction conflicts. Contrary to the generic SCC algorithm, it does not fix a priori the number of speculative shadows that each transaction in the system is allowed to have at any point in its lifetime. Thus, every time that a new conflict is encountered, a new speculative shadow is created, to accommodate it. Moreover, each individual transaction can have a different degree of redundancy, in the number of shadows it can originate. This flexibility, of course, is gained at the expense of an increased amount of redundant computations that are allowed in the system. See Appendix C for the details of the SCC-MS algorithm.

5 Conclusion

SCC-based algorithms offer a new dimension (namely redundancy) that can be used effectively to improve the responsiveness of RTDBMS. Using SCC, several shadow transactions execute on behalf of a given uncommitted transaction so as to protect against the hazards of blockages and restarts, which are characteristics of Pessimistic and Optimistic Concurrency Control algorithms, respectively.

In this paper, we presented a generic algorithm (SCC-nS) which characterizes a family of algorithms that differ in the total amount of redundancy they introduce. We described SCC-nS both informally and formally. We demonstrated its superiority for RTDBMS through numerous examples. Three members of the SCC-nS family (namely SCC-1S, SCC-2S, and SCC-MS) were singled out and contrasted. SCC-1S does not introduce any additional redundancy and is shown to be equivalent to the OCC-BC algorithm of [19, 22]. SCC-2S allows exactly one additional *pessimistic* shadow in the system and is shown to outperform OCC-BC with respect to the timely commitment of transactions. SCC-MS introduces as many shadows as necessary to account for all possible *pair-wise* conflicts between uncommitted transactions. This is in contrast to the general algorithm described in [4], where conflicts involving more than two transactions are also considered.

An interesting observation is that the SCC-based protocols discussed in this paper do not make use of transaction priorities or deadline information in resolving data conflicts. This property, while it protects our algorithms from problems related to priority dynamics (e.g. *priority inversions* [23]), it also prevents them from making better decisions which could help in decreasing the number of missed deadlines in the system. We are currently working on developing an SCC-based algorithm which allows for the use of deadline information to improve its responsiveness.

References

[1] Robert Abbott and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 1–12, Los Angeles, Ca, 1988.

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

[4] Azer Bestavros. Speculative Concurrency Control: A Position Statement. Technical Report BUCS-TR-92-016, Computer Science Department, Boston University, Boston, MA, July 1992.

[5] Azer Bestavros and Spyridon Braoudakis. A family of Speculative Concurrency Control Algorithms. Technical Report BUCS-TR-92-017, Computer Science Department, Boston University, Boston, MA, July 1992.

[6] Azer Bestavros, Spyridon Braoudakis, and Euthimios Panagos. Performance Evaluation of Two-shadow Speculative Concurrency Control. Technical Report BUCS-TR-93-001, Computer Science Department, Boston University, Boston, MA, January 1993.

[7] C. Boksenbaum, M. Cart, J. Ferrié, and J. Francois. Concurrent certifications by intervals of timestamps in distributed database systems. *IEEE Transactions on Software Engineering*, pages 409–419, April 1987.

[8] A. P. Buchmann, D. C. McCarthy, M. Hsu, and U. Dayal. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency controls. In *Proceedings of the 5th International Conference on Data Engineering*, Los Angeles, California, February 1989.

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

[10] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistensy in a shared data base. In G. M. Nijssen, editor, *Modeling in Data Base Management Systems*, pages 365–395. North-Holland, Amsterdam, The Netherlands, 1976.

[11] 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.

[12] 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.

[13] J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of the 10th Real-Time Systems Symposium*, December 1989.

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

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

[16] Henry Korth. Triggered real-time databases with consistency constraints. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, 1990.

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

[18] 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.

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

[20] Christos Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[21] Krithi Ramamritham. Real-time databases. *International journal of Distributed and Parallel Databases*, 1(2), 1993.

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

[23] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical Report CMU-CS-87-181, Carnegie Mellon University, Computer Science Department, December 1987.

[24] Lui Sha, R. Rajkumar, and J. Lehoczky. Concurrency control for distributed real-time databases. *ACM, SIGMOD Record*, 17(1):82–98, 1988.

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

[26] Mukesh Singhal. Issues and approaches to design real-time database systems. *ACM, SIGMOD Record*, 17(1):19–33, 1988.

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

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

Appendices

A The Generic SCC-nS Algorithm

A. The Start Rule: When the execution of a new transaction T_r is requested, an optimistic shadow $T_r^o \in \mathcal{T}^{\mathcal{O}}$ is created and executed.

```
0.      $SpecNumber(T_r) \leftarrow 0$;
1.      $ReadSet(T_r^o) \leftarrow \{\}$;
2.      $WriteSet(T_r^o) \leftarrow \{\}$;
```

B. The Read Rule: Whenever an optimistic shadow T_r^o wishes to read an object X, then:

```
0.      $ReadSet(T_r^o) \leftarrow \{(X,\_)\}$;
          for all $T_u^o$ in $\mathcal{T}^{\mathcal{O}}$, such that $X \in WriteSet(T_u^o)$ do
1.          if $((SpecNumber(T_r) < n-1) \wedge (\forall T_r^i \in \mathcal{T}_r^{\mathcal{S}}, (T_u,\_) \notin WaitFor(T_r^i)))$ then{
1.1             A new speculative shadow $T_r^j$ is forked off $T_r^o$;
1.2             $WaitFor(T_r^j) \leftarrow \{(T_u, X)\}$;
1.3             $SpecNumber(T_r) \leftarrow SpecNumber(T_r) + 1\}$;
```

C. The Write Rule: Whenever an optimistic shadow T_u^o wishes to write an object X, then:

```
0.      $WriteSet(T_u^o) \leftarrow \{X\}$;
          for all $T_r^o$ in $\mathcal{T}^{\mathcal{O}}$, such that $(X,\_) \in ReadSet(T_r^o)$ do
1.          if $(SpecNumber(T_r) < n-1)$ then{
1.1             if $(\forall T_r^i \in \mathcal{T}_r^{\mathcal{S}}, (T_u,\_) \notin WaitFor(T_r^i))$ then{
1.1.1               A new speculative shadow $T_r^j$ is forked off $BestShadow(T_r, X)$;
1.1.2               $WaitFor(T_r^j) \leftarrow \{(T_u, X)\}$;
1.1.3               $SpecNumber(T_r) \leftarrow SpecNumber(T_r) + 1$
1.2             }else if $(\exists T_r^k \in \mathcal{T}_r^{\mathcal{S}}, \exists Y : ((X,\_) \in ReadSet(T_r^k) \wedge (T_u, Y) \in WaitFor(T_r^k)))$ then{
1.2.1               $T_r^k$ is aborted and replaced by $T_r^m$ which is forked off $BestShadow(T_r, X)$;
1.2.2               $WaitFor(T_r^m) \leftarrow \{(T_u, X)\}\}$;
2.          }else if $(SpecNumber(T_r) = n-1)$ then
2.1             if $(\exists T_r^k \in \mathcal{T}_r^{\mathcal{S}} : (X,\_) \in ReadSet(T_r^k))$ then
2.1.1               Abort $LastShadow(T_r)$;
2.1.2               A new speculative shadow $T_r^m$ is forked off $BestShadow(T_r, X)$;
2.1.3               $WaitFor(T_r^m) \leftarrow \{(T_u, X)\}\}$;
```

D. The Blocking Rule: A standby shadow T_r^i is blocked at the *earliest point* at which it wishes to Read an object X that is written by any transaction T_u, such that $(T_u, X) \in WaitFor(T_r^i)$.

E. The Commit Rule: Whenever it is decided to commit an optimistic shadow T_r^o on behalf of a transaction T_r, then:

```
1.          $\forall T_r^i \in \mathcal{T}_r^{\mathcal{S}}, T_r^i$ is aborted;
2.          for all $T_u \in \mathcal{T}$, such that $(\exists T_u^i \in \mathcal{T}_u^{\mathcal{S}} : (T_r, X) \in WaitFor(T_u^i))$ do{
2.1             $T_u^o$ is aborted;
2.2             $T_u^i$ is promoted to become the new optimistic shadow of $T_u$;
2.3             $SpecNumber(T_u) \leftarrow SpecNumber(T_u) - 1$;
2.4             for all $T_u^j \in \mathcal{T}_u^{\mathcal{S}}$, such that $(X,\_) \in ReadSet(T_u^j)$ do{
2.4.1               $T_u^j$ is aborted;
2.4.2               $SpecNumber(T_u) \leftarrow SpecNumber(T_u) - 1$ }};
3.          for all $T_u \in \mathcal{T}$, such that $(\exists X : X \in WriteSet(T_r^o) \wedge (X,\_) \in ReadSet(T_u^o)$
                $(\nexists T_u^i \in \mathcal{T}_u^{\mathcal{S}} : (T_r, X) \in WaitFor(T_u^i)))$ do{
3.1             $T_u^o$ is aborted;
3.2             A new optimistic shadow $T_u^o$ is forked off $LastShadow(T_u)\}$;
```

B The LastShadow and BestShadow functions

(a) $\underline{\mathit{LastShadow}()} : \mathcal{T} \rightarrow \mathcal{T}^{\mathcal{S}}$, such that $T_r \in \mathcal{T} \longmapsto T_r^{last} \in \mathcal{T}^{\mathcal{S}}$ **iff** $(\exists X : (X, t_x) \in \mathit{ReadSet}(T_r^o)) \wedge ((\exists T_u \in \mathcal{T} : (T_u, X) \in \mathit{WaitFor}(T_r^{last})) \wedge (\forall Y : ((Y, t_y) \in \mathit{ReadSet}(T_r^o) \wedge (\exists T_v \in \mathcal{T}, \exists T_r^i \in \mathcal{T}_r^{\mathcal{S}} : (T_v, Y) \in \mathit{WaitFor}(T_r^i)))) \Longrightarrow t_y \leq t_x)$.

(b) $\underline{\mathit{BestShadow}()} : (\mathcal{T}, \mathit{object}) \rightarrow \mathcal{T}^{\mathcal{S}}$, such that $(T_r, X) \in (\mathcal{T}, \mathit{Object}) \longmapsto T_r^{best} \in \mathcal{T}^{\mathcal{S}}$ **iff** $(X, t_x) \in \mathit{ReadSet}(T_r^o) \wedge (X, t_x) \notin \mathit{ReadSet}(T_r^{best}) \wedge (\exists T_u \in \mathcal{T}, \exists Y : ((Y, t_y) \in \mathit{ReadSet}(T_r^o) \wedge (T_u, Y) \in \mathit{WaitFor}(T_r^{best}))) \wedge (\forall Z : ((Z, t_z) \in \mathit{ReadSet}(T_r^o) \wedge (\exists T_v \in \mathcal{T}, \exists T_r^i \in \mathcal{T}_r^{\mathcal{S}} : ((T_v, Z) \in \mathit{WaitFor}(T_r^i) \wedge (X, t_x) \notin \mathit{ReadSet}(T_r^i)))) \Longrightarrow t_z \leq t_y)$.

C The Multi-Shadow SCC Algorithm

A. The Start Rule: When the execution of a new transaction T_r is requested, an optimistic shadow $T_r^o \in \mathcal{T}^{\mathcal{O}}$ is created and executed.

```
0.   $\mathit{ReadSet}(T_r^o) \leftarrow \{\}$;
1.   $\mathit{WriteSet}(T_r^o) \leftarrow \{\}$;
```

B. The Read Rule: Whenever an optimistic shadow T_r^o wishes to read an object X, then:

```
0.   $\mathit{ReadSet}(T_r^o) \leftarrow \{(X, \_)\}$;
     for all $T_u^o$ in $\mathcal{T}^{\mathcal{O}}$, such that $X \in \mathit{WriteSet}(T_u^o)$ do
1.       if $(\forall T_r^i \in \mathcal{T}_r^{\mathcal{S}}, (T_u, \_) \notin \mathit{WaitFor}(T_r^i))$ then{
1.1          A new speculative shadow $T_r^j$ is forked off $T_r^o$;
1.2          $\mathit{WaitFor}(T_r^j) \leftarrow \{(T_u, X)\}$;}
```

C. The Write Rule: Whenever an optimistic shadow T_u^o wishes to write an object X, then:

```
0.   $\mathit{WriteSet}(T_u^o) \leftarrow \{X\}$;
     for all $T_r^o$ in $\mathcal{T}^{\mathcal{O}}$, such that $(X, \_) \in \mathit{ReadSet}(T_r^o)$ do
1.       if $(\forall T_r^i \in \mathcal{T}_r^{\mathcal{S}}, (T_u, \_) \notin \mathit{WaitFor}(T_r^i))$ then{
1.1          A new speculative shadow $T_r^i$ is forked off $\mathit{BestShadow}(T_r, X)$;
1.2          $\mathit{WaitFor}(T_r^i) \leftarrow \{(T_u, X)\}$;
2.       }else if $(\exists T_r^k \in \mathcal{T}_r^{\mathcal{S}}, \exists Y : ((X, \_) \in \mathit{ReadSet}(T_r^k) \wedge (T_u, Y) \in \mathit{WaitFor}(T_r^k)))$ then{
2.1              $T_r^k$ is aborted and replaced by $T_r^m$ which is forked off $\mathit{BestShadow}(T_r, X)$;
2.2          $\mathit{WaitFor}(T_r^m) \leftarrow \{(T_u, X)\}\}$;
```

D. The Blocking Rule: A standby shadow T_r^i is blocked at the *earliest point* at which it wishes to Read an object X that is written by any transaction T_u, such that $(T_u, X) \in \mathit{WaitFor}(T_r^i)$.

E. The Commit Rule: Whenever it is decided to commit an optimistic shadow T_r^o on behalf of a transaction T_r, then:

```
1.       $\forall T_r^i \in \mathcal{T}_r^{\mathcal{S}}, T_r^i$ is aborted;
2.       for all $T_u \in \mathcal{T}$, such that $(\exists T_u^i \in \mathcal{T}_u^{\mathcal{S}} : (T_r, X) \in \mathit{WaitFor}(T_u^i))$ do{
2.1          $T_u^o$ is aborted;
2.2          $T_u^i$ is promoted to become the new optimistic shadow of $T_u$;
2.3          for all $T_u^j \in \mathcal{T}_u^{\mathcal{S}}$, such that $(X, \_) \in \mathit{ReadSet}(T_u^j)$ do
2.3.1            $T_u^j$ is aborted};
```