Timeliness via Speculation for Real-Time Databases*

Azer Bestavros (best@cs.bu.edu)
Spyridon Braoudakis (sb@cs.bu.edu)

Computer Science Department
Boston University
Boston, MA 02215

Abstract

Various concurrency control algorithms differ in the time when conflicts are detected, and in the way they are resolved. In that respect, the Pessimistic and Optimistic Concurrency Control (PCC and OCC) alternatives represent two extremes. PCC locking protocols detect conflicts as soon as they occur and resolve them using *blocking*. OCC protocols detect conflicts at transaction commit time and resolve them using *rollbacks* (restarts). For real-time databases, blockages and rollbacks are hazards that increase the likelihood of transactions missing their deadlines. We propose a *Speculative* Concurrency Control (SCC) technique that minimizes the impact of blockages and rollbacks. SCC relies on the use of added system resources to *speculate* on potential serialization orders and to ensure that if such serialization orders materialize, the hazards of blockages and roll-backs are minimized. We present a number of SCC-based algorithms that differ in the level of speculation they introduce, and the amount of system resources (mainly memory) they require. We show the performance gains (in terms of number of satisfied timing constraints) to be expected when a representative SCC algorithm (SCC-2S) is adopted.

Keywords: real-time databases; concurrency control; performance evaluation; simulation; client-server distributed databases.

1 Introduction

Traditional concurrency control algorithms can be classified broadly as either *pessimistic* or *optimistic*. Pessimistic Concurrency Control (PCC) algorithms [Eswa76, Gray76] avoid any concurrent execution of transactions as soon as *potential* conflicts between these transactions are detected. Optimistic Concurrency Control (OCC) algorithms [Boks87, Kung81] allow such transactions to proceed at the risk of having to restart them in case these potential conflicts *materialize*.

Most real-time concurrency control schemes considered in the literature and used in commercial systems combine Two-Phase Locking (2PL), which is a PCC strategy, with a priority scheme to guarantee that the more urgent transactions are not blocked out waiting for less urgent ones [Abbo88, Stan88, Huan92, Sing88, Sha88, Sha91]. Despite its widespread use, 2PL has some properties such as the possibility of deadlocks and long and unpredictable blocking times that damage its appeal for real-time environments. This led to a large body of research on alternatives to 2PL for RTDBS [Huan91].

*This work has been partially supported by GTE Labs (fund number 3658-3) and by NSF (grant CCR-9308344).

For conventional DataBase Management Systems (DBMS) with limited resources, performance studies have concluded that PCC blocking-based conflict resolution policies result in throughputs higher than those achievable by OCC restart-based conflict resolution policies [Agra87]. However, for Real-Time DataBase Systems (RTDBS) throughput (or maximum concurrency) ceases to be an appropriate measure of performance. Rather, the number of transactions completed before their set deadlines becomes the decisive performance measure [Buch89]. Haritsa *et al.* [Hari92] investigated the behavior of both PCC and OCC schemes in a real-time environment and showed that for a RTDBS with firm deadlines (where late transactions are discarded immediately) OCC outperforms PCC, especially when resource contention is low.

The main disadvantage of classical OCC [Kung81] is that transaction conflicts are not detected until the validation phase, at which time it may be too late to restart.[1] The Broadcast Commit variant of the classical OCC (OCC-BC) [Mena82, Robi82] attempts to solve this problem by requiring that a committing transaction notifies all uncommitted, conflicting transactions for an immediate restart. OCC-BC detects conflicts earlier than the basic OCC algorithm resulting in less wasted resources and earlier restarts. However, like the classical OCC approach, it is not sensitive to transactions' priorities or deadlines. This has been partially remedied by introducing waiting [Hari90] and blocking [Lin90, Son92] to OCC-based algorithms.

Recently Bestavros proposed a categorically different approach to concurrency control for RTDBS [Best92]. His approach relies on the use of standby processes to speculate on alternative schedules (serialization order of transactions), once conflicts that threaten the consistency of the database are detected. These alternative schedules are adopted *only if* suspected inconsistencies materialize; otherwise, they are abandoned. Due to its nature, this approach has been termed *Speculative Concurrency Control* (SCC).

SCC algorithms use added processes 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.

[1] PCC 2PL 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 remainder of this paper is organized as follows. In section 2, we overview the basic *order-based* SCC algorithm. Despite its impracticality (in terms of the number of speculative processes it requires), this algorithm serves as a reference point for subsequent SCC-based algorithms. In section 3, we describe the SCC-kS class of algorithms, which restricts the number of processes available for a transaction to a constant k. The SCC-2S—a member of the SCC-kS class—is presented. SCC-2S is the simplest SCC-based algorithm. We use it in our simulation studies as a representative of SCC algorithms. In section 4, we compare the performance of SCC-2S to that of OCC-BC. We describe the client-server RTDBS model and the workload model used in our experiments, and we discuss our simulation results in detail. Finally, in section 5, we conclude by summarizing our findings.

2 Basic Speculative Concurrency Control

Let $T_1, T_2, \ldots T_m$ be the set of active transactions in the system. A transaction T_i consists of a sequence of actions $a_{i1}, a_{i2}, \ldots a_{il}$, where each a_{ij}, $j = 1, 2, \ldots l$, is either a *read* or a *write* operation on one of the shared objects of the database. Write and subsequent read operations of an object x by an uncommitted transaction T_i are performed on a private copy of x in the local workspace of T_i. The updated value of object x is made visible to other transactions (*i.e.* reflected in the shared database) only when T_i is committed. 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 the consistency of the database [Papa79, Bern87].

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 validation, when all database updates are made permanent, all write requests are buffered by the data manager and serialized according to their transaction validation order [Bern87].

SCC algorithms allow several processes to co-exist on behalf of the same transaction. Each one of these processes makes different assumptions with regard to the *Speculated Order of Serialization* (SOS). For a transaction T_i, we call each one of these processes a *shadow* of T_i.

Similar to the OCC-BC algorithm, we adopt a forward validation method, in which validation is done against active transactions only. In particular, when a transaction T_r enters its validation phase, the algorithm must check that the ReadSets of all active transactions do not intersect with

the WriteSet of T_r, otherwise any such transactions are aborted. This forward validation method implies that transaction aborts result from *reading* an object that a validating transaction wrote. This is why we shadow the reader of an object and not the writer.

To illustrate the basic premise of SCC, we compare it to OCC-BC using an 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 OCC-BC algorithm means restarting transaction T_2 when T_1 enters its validation phase (figure 1(a)). This restart may be too late for T_2 to meet its deadline. The SCC approach remedies this hazard by requiring T_2 to fork-off a *shadow transaction* T_2^1 immediately before the reading of item x (which has been modified by the uncommitted transaction T_1). Two possible scenarios may develop depending on the time needed for T_2 to reach its validation phase. If T_2 reaches its validation phase before T_1, then T_2 will be validated and committed without any need to disturb T_1. Once T_2 commits, the shadow T_2^1 is aborted. However, if T_1 reaches its validation phase first, the SCC protocol, instead of restarting T_2, simply replaces T_2 by its shadow T_2^1 (figure 1(b)). T_2 "speculated" that it will commit ahead of T_1. When this speculation turned out to be wrong, T_2', which "speculated" correctly that T_1 will commit first, was adopted. From this example, we notice that the waiting pattern is determined dynamically, based on the shadow's Speculated Order of Serialization (SOS).

As illustrated in the example of figure 1, the basic idea of SCC is to keep enough shadows (standby alternate processes) for each SOS. Such shadows will be blocked at appropriate points in time so as to be ready to resume execution, if needed. Figure 2 demonstrates this concept by showing all shadows and SOS's for a transaction T_3, which conflicts with two other transactions T_1 and T_2. In [Best93a], an *Order-Based* SCC (SCC-OB) algorithm, which generalizes this idea, is proposed. The SCC-OB algorithm requires an exponential number of shadows, namely $\sum_{i=1}^{n} \frac{(n-1)!}{(n-i)!} = \mathcal{O}((n-1)!)$, to account for all the possible orderings of any n uncommitted, conflicting transactions. Fortunately, it can be shown that $\sum_{i=1}^{n}(n-i) = \frac{n(n-1)}{2}$ shadows per transaction are actually sufficient, whereby each shadow accounts for *multiple* SOS instead of a single one. Furthermore, at any point in time only a maximum of n such shadows per transaction is necessary. This reduction in complexity can be achieved by observing that standby shadows do not read dirty data, and thus transactions need not consider their relative position in the different serialization orders. Accordingly, they don't need to keep a number of shadows accounting for each one of these serialization orders. Rather, they need only to be concerned about the immediate future—which transaction will commit next—accounting for *conflicts* encountered with other uncommitted transactions. The *Conflict-Based* SCC (SCC-CB) variant presented in [Brao94] makes use of the aforementioned improvements over the SCC-OB algorithm.

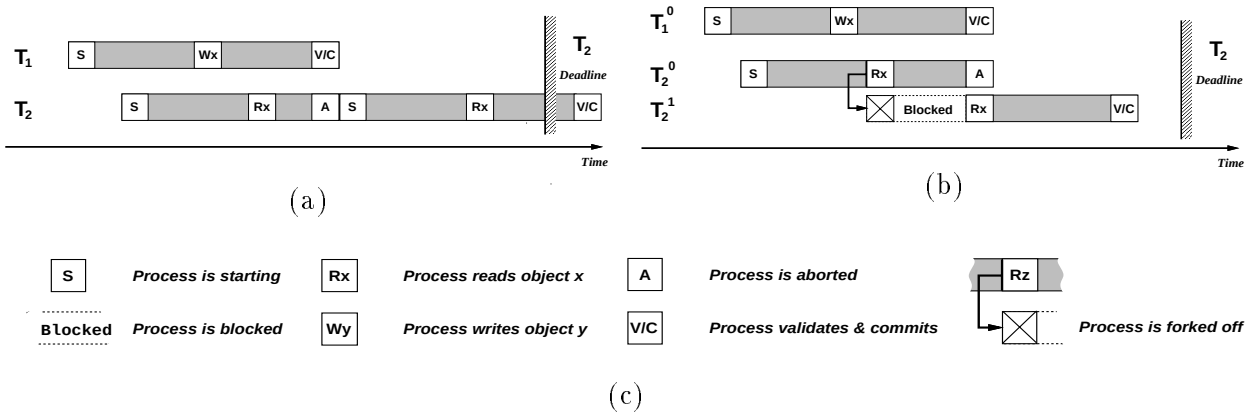

Figure 1: Transaction management under (a) OCC-BC (b) SCC (c) Legend

3 The SCC-kS Class of Algorithms

In this section, we describe a class of SCC algorithms that operate under a *limited resources* assumption. This assumption restricts to k the number of shadows alloted for each uncommitted transaction. A formal description of the SCC-kS class of algorithms can be found in Appendix A.1.

3.1 Algorithm Overview

Under the SCC-kS algorithm, shadows executing on behalf of a transaction are either *optimistic* or *speculative*. Optimistic shadows execute unhindered, whereas speculative shadows are kept ready to replace a defunct optimistic shadow, if such a replacement is deemed necessary. At any point during its execution, a transaction T_r has exactly one optimistic shadow T_r^o. In addition, T_r may have i speculative shadows T_r^i, for $i = 0, \ldots, k-1$.

Optimistic shadow behavior:
The optimistic shadow T_r^o executes under the assumption that T_r 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 it reaches its validation phase, 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 all the other

uncommitted transactions in the system with which it conflicts, *except* for one particular transaction T_u, which is *speculated* to commit before T_r. T_r^s remains blocked on the shared object X, on which the conflict with T_u developed, waiting to read the value that T_u will assign to X when it commits. If T_r^s's speculation becomes true (*i.e.* T_u commits before T_r), 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 with T_u.

The number of speculative shadows maintained by SCC-kS (namely $k-1$) may 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. In this paper we have adopted a particular solution, which requires the speculative shadows of SCC-kS to account for the *first* $l \leq k-1$ conflicts (whether read-after-write or write-after-read) encountered by a transaction. This is implemented by the *Latest-Blocked-First-Out* (LBFO) shadow replacement policy, which replaces the shadow with the latest blocking point. LBFO is one of several policies that could be adopted. We are currently investigating alternatives to this policy, which utilize information about deadlines and priorities of the conflicting transactions to account for the *most probable* serialization orders [Brao94].

Figure 3 illustrates the LBFO shadow replacement policy when only two speculative shadows are alloted for transaction T_1. The presumption that the first two conflicts in which T_1 participated (by accessing objects Y and Z), is revised when transaction T_2 writes object X. In this case, the newly detected conflict (T_2, X) becomes the earliest conflict of T_1. T_1^2, the *latest* shadow of T_1 is aborted and replaced by a new speculative shadow, T_1^3, accounting for the new (T_2, X) conflict.

3.2 Description of SCC-kS

Let $\mathcal{T} = T_1, T_2, T_3, \ldots, T_m$ be the set of uncommitted transactions in the system. For each transaction T_r we keep a variable $SpecNumber(T_r)$, 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: $ReadSet(T_r^i)$ and $WriteSet(T_r^i)$. $ReadSet(T_r^i)$ records pairs (X, t_x), where X is an object read by T_r^i, and t_x represents the order in which this operation was performed. $WriteSet(T_r^i)$ 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 $WaitFor(T_r^i)$, 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. $(T_u, X) \in WaitFor(T_r^i)$ implies that T_r^i must wait for T_u before being allowed to read object X. The SCC-kS algorithm is presented 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, this conflict is ignored since no more shadows for T_r could be created. The Commit Rule (see below) deals with the corrective measures that need to be taken, should this conflict materialize.

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 [Best93b].

When the new conflict implicates transactions that already conflict with each other, some adjustments may be necessary. In figure 4, 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.

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, when this read is in conflict with any transaction that T_r^i must wait for according to its SOS.

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, awaiting T_r's commitment, or not. The first case is illustrated in figure 5, where the speculative shadow T_1^2 of transaction T_1—having anticipated 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 6, where the commitment of the optimistic shadow T_2^o on behalf of transaction T_2 was not accounted for by any speculative shadow.[2] In this case, a shadow is forked off T_1^2, the *latest* shadow of T_1, to become the new optimistic shadow of transaction T_1. This is the best we can do in the absence of a speculative shadow accounting for the (T_2, Z) conflict.

3.3 Correctness of SCC-kS

Having described its basic concepts, we now present a proof of correctness for the SCC-kS algorithm. First, we define the notions of *history* and *serialization graph* (SG). A *history* H is a partial order of operations that represents the execution of a set of transactions $\mathcal{T}$. Any two conflicting operations in H must be ordered. The *serialization graph* for history H, denoted by SG(H), is a directed graph whose nodes, $T_i \in \mathcal{T}$, are committed transactions in H. Its edges are all $T_i \leftarrow T_j$, for $i \neq j$, such that one of T_i's operations precedes and conflicts with one of T_j's operations in H. A history H is serializable if and only if its serialization graph SG(H) is acyclic [Bern87].

Lemma 1 *Let T_1 and T_2 be two committed transactions in a history H produced by the SCC-kS algorithm. If there is an edge $T_1 \leftarrow T_2$ in SG(H), then the commitment of transaction T_1 precedes that of transaction T_2, denoted by $T_1 \prec T_2$, in the serialization order.*

Proof: Because of the edge $T_1 \leftarrow T_2$ in SG(H) the two transactions must have at least one conflicting operation, over some object X. Without loss of generality, assume that T_1^k and T_2^l are the two shadows that committed on behalf of transactions T_1 and T_2, respectively. Notice that, by the Commit Rule, *at most one* such shadow may exist for each executing transaction. There are three cases to be examined:

[2] Figure 6 makes the implicit assumption that transaction T_1 is limited to having at most two speculative shadows at any point during its execution.

$\Rightarrow T_1^k$'s read operation on X precedes T_2^l's write operation on X (Read-Write):
Suppose $T_2 \prec T_1$. Then according the SCC-kS protocol the commitment of T_2^l precedes that of T_1^k. But, by the Commit Rule, when T_2^l reaches its commit phase, T_1^k will have to be aborted. By assumption, however, T_1^k commits on behalf of transaction T_1 – a contradiction.

$\Rightarrow T_1^k$'s write operation on X precedes T_2^l's read operation on X (Write-Read):
Two cases exist:

1. T_2^l is an optimistic shadow: In this case, as soon as T_1^k reaches its commit phase, T_2^l would have to be aborted (Commit Rule). By assumption, however, T_2^l commits on behalf of transaction T_2 – a contradiction.
2. T_2^l is a speculative shadow: In this case, T_2^l must have been forked off some other shadow T_2^m, executing on behalf of transaction T_2, which requested to R_x at some point during its execution. Two cases need to be examined:
   a. T_2^m **read object X after T_1^k wrote object X**: T_2^l is forked off T_2^m and (T_1, X) is appended to *WaitFor*(T_2^l), as soon as T_2^m requests to read object X (Read Rule).
   b. T_2^m **read object X before T_1^k wrote object X**: T_2^l is forked off some shadow of T_2 and (T_1, X) is appended to its *WaitFor*(T_2^l), as soon as T_1^k requests to write object X (Write Rule).

   In both cases, T_2^l cannot reach its commit phase before transaction T_1 commits, because its *WaitFor*(T_2^l) cannot be empty while T_1 is still in progress (Commit Rule). Therefore, again $T_1 \prec T_2$ in the serialization order.

$\Rightarrow T_1^k$'s write operation on X precedes T_2^l's write operation on X (Write-Write):
Suppose $T_2 \prec T_1$. Then according the SCC-kS protocol T_2^l, enters its commit phase before T_1^k. T_2^l's write operation on X is sent to the data manager first. It will either be processed before T_1^k's write operation on X, or it will be discarded when the data manager receives T_1^k's write operation on X (TWR). Therefore, T_1^k's write operation on X is never processed before that of T_2^l's. Then this conflict (implying the edge $T_1 \leftarrow T_2$ in the SG(H)) is impossible – a contradiction.

■

Theorem 1 *Every history H produced by the SCC-kS algorithm is serializable.*

Proof: The proof of the theorem is by contradiction. In particular, suppose that there is a cycle $T_1 \leftarrow T_2 \leftarrow \cdots \leftarrow T_n \leftarrow T_1$ in SG(H). Then, by the above argument, it must be the case that $T_1 \prec T_2 \prec \cdots \prec T_n \prec T_1$, which leads to a contradiction. Therefore no cycle can exist in SG(H) and thus the SCC-kS algorithm produces only serializable histories.

■

3.4 Two-Shadow SCC (SCC-2S)

In this section, we present, SCC-2S, a member of the SCC-kS class, which allows a maximum of two shadows per uncommitted transaction to exist in the system at any point in time: an *optimistic* shadow and a *pessimistic* shadow. SCC-2S is used in the following sections as a representative of SCC algorithms for simulation purposes.

Let T_i be any uncommitted transaction in the system. The optimistic shadow for T_i 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 pessimistic shadow for T_i, on the contrary, is subject to blocking and restart. It is kept ready to replace the optimistic shadow, if necessary. The pessimistic shadow runs under the 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 optimistic 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 backup 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 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.

To illustrate how SCC-2S works, consider the schedule shown in figure 1(b). Both transactions T_1 and T_2 start with one optimistic 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. The optimistic 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 optimistic 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 7 illustrates the behavior of SCC-2S when a second conflict develops between T_2 and another transaction T_3. In particular, the optimistic 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, T_2^1 is promoted to become the optimistic shadow and is, thus, allowed to resume.

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 than two shadows will be needed over a stretch of time. In Figure 8, after T_2^1 is promoted to become the optimistic shadow for T_2, a pessimistic shadow T_2^2 is forked off to account for the read-write conflict between T_2^1 and T_1.

4 Performance Evaluation

In this section, a comparative evaluation of the performance of SCC-2S (as a representative of SCC-based algorithms) and OCC-BC (as a representative of OCC-based algorithms) in RTDBS is presented. First, we describe the database model, the workload model, the performance measures, and parameters used in our baseline model. Next, we discuss our results and conclusions regarding the impact of data contention, resource contention, deadline tightness, deadline policies, and various loading conditions.

4.1 A Client-Server RTDBS Model

Being interested in measuring the overhead imposed on the system by the implementation of each algorithm, we built our model to closely resemble a real system. In particular, the server's Transaction and Buffer Manager constitute partial implementations, whereas the Disk Manager is simulated. For the same reason, actual rather than simulated time, is measured. This includes the communication delays caused by the messages exchanged between the server and the clients.

The database is modeled as a collection of pages stored on a number of disks. The centralized server is a shared memory multiprocessor which communicates with client transactions by exchanging messages. The Transaction Manager is responsible for keeping track of the pages used by the transactions running on the system. The Buffer Manager is responsible for providing the pages requested by the transactions, as well as storing into the buffer pool the dirty pages received by a committed transaction. The Least Recently Used (LRU) policy is employed for page replacement.

The transactions arrival rate follows a Poisson distribution with each transaction having an associated deadline time. Each transaction consists of a number of read and write operations. Each write operation is being preceded by a corresponding read operation on the same data object. The local transaction managers keep track of the pages accessed by their transactions, as well as their access modes. If a page is not present into the client's local Pool, it is requested from the server. This can cause up to two I/O operations on the server. During commit time, all updated pages are sent to the server. For each such updated page at most one I/O operation is performed.

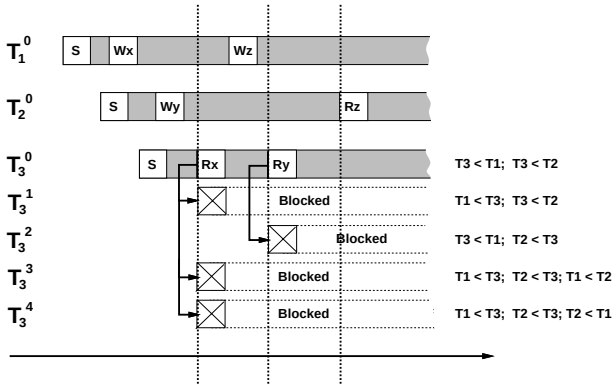

Figure 2: T_3 has five shadows each with a different Speculated Order of Serialization (SOS).

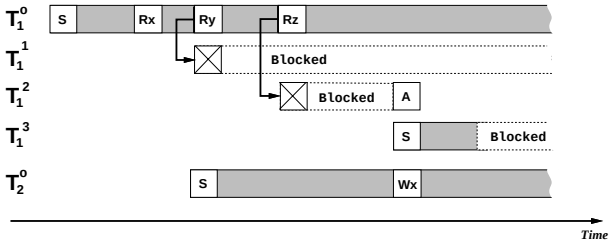

Figure 3: Detecting conflict (T_2, X) causes the replacement of T_1^2 by T_1^3.

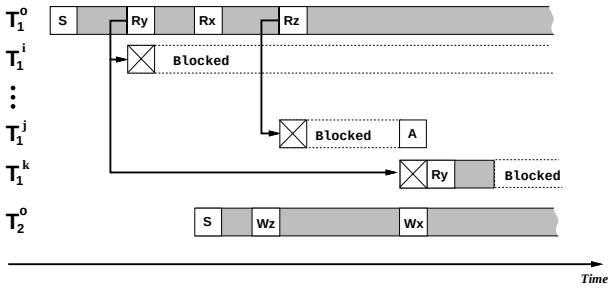

Figure 4: T_1^j is replaced by T_1^k when conflict (T_2, X) is detected.

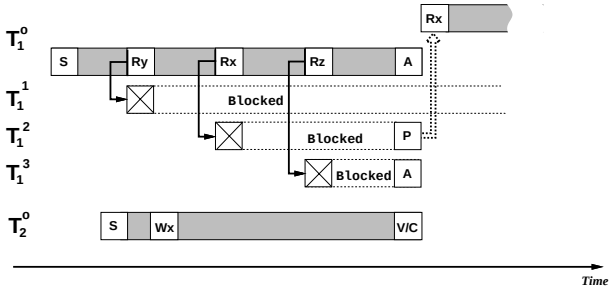

Figure 5: T_1^2 is promoted to replace the optimistic shadow of T_1. T_1^3 is aborted.

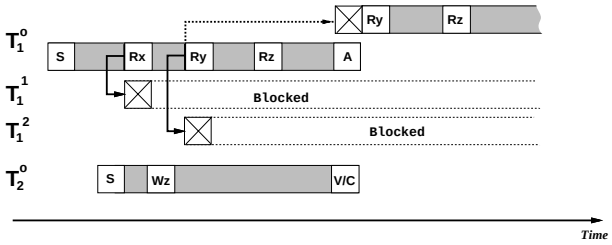

Figure 6: T_1^o is forked off T_1^2 when conflict (T_2, Z) materializes.

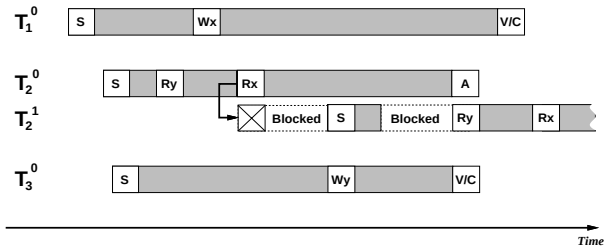

Figure 7: Schedule with a pessimistic shadow restart and promotion.

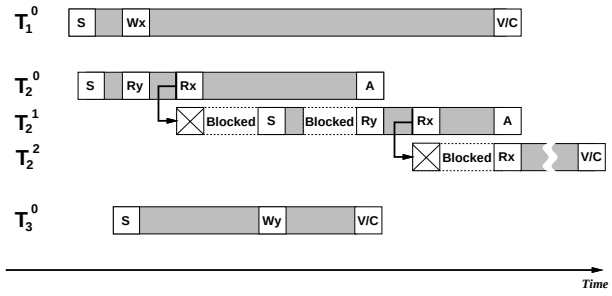

Figure 8: Schedule with two pessimistic shadows.

Parameter	Meaning	Setting
DBSize	Database size in pages	1000 pages
TRANSize	Size of transactions in pages accessed	20 pages
WProb	Probability to update an accessed page	0.25
SRatio	Slack Ratio used in deadline formula	1.5
RTime	Average time to read a page	3 msec
WTime	Average time to update part of a page	15 msec

Table 1: The Workload Parameters

4.2 Workload Model

The workload model characterizes the transactions running in the system according to the number of pages they access (read and/or write) and their execution time. Table 1 summarizes the key workload parameters used in our simulation experiments.

The DBSize parameter fixes the number of pages in the database. The number of pages accessed by a transaction is given by the TRANSize parameter. Page requests are generated from a uniform distribution spanning the entire database. The WProb parameter specifies the probability that a page which is already read will also be updated. The SRatio parameter provides the deadline slack factor in our simulations. By changing its value we can smoothly vary the tightness of transaction deadlines. The value of SRatio ranges from zero to infinity, with zero meaning that transactions have no laxity. The RTime and Wtime parameters are set to the average time that a transaction needs to read and update a page present in its client's local Pool, respectively.

In addition, we denote by R_{size}, and W_{size} the number of pages that a transaction reads, and writes, respectively. The average times needed to read and write a page are denoted by AVG_{read}, and AVG_{write}, respectively. T_{start} is the set-up time needed to start a transaction, and AVG_{end} is the time needed to commit a transaction. The following formula for the average execution time T_{avg} of a transaction can then be obtained:

$$T_{avg} = R_{size} * AVG_{read} + W_{size} * AVG_{write} + T_{start} + AVG_{end}$$

Knowing the average execution time for a transaction of a given size, T, we can calculate the deadline assigned to a transaction based on its Slack Ratio SRatio as follows:

$$D_T = T_{avg} + T_{avg} * \texttt{SRatio}$$

4.3 Performance Measures

Two primary performance metrics used in this paper are the number of transactions that miss their deadlines, *Missed Deadlines*, and the average time by which late transactions miss their deadlines, *Average Tardiness*. A transaction that commits within its deadline has a tardiness of zero. A transaction that completes after its deadline has a tardiness of $C_T - D_T$, where C_T and D_T are the transaction's completion time and deadline time, respectively.

Previous studies have argued that improving *both* of the aforementioned metrics is difficult [Hari90]. Our simulations have shown that by adopting a superior concurrency control algorithm (SCC-2S in this case), *both* metrics can, indeed, be improved.

Our experiments assume that transaction deadlines are soft. This entails that late transactions (those missing their deadlines) must complete – nevertheless – with the minimum possible delay. Even though transaction response time was not explicitly measured in our simulations, the Average Tardiness metric can be used as an approximation. In particular, by reducing the `SRatio` value to 0, it can be shown that the transaction's Average Tardiness and Response Times are related. This observation coupled with our soft deadline assumption allow our simulations to be useful in the evaluation of SCC-2S for conventional DBMS.

The simulations also generated a host of other statistical information, including CPU and disk utilizations, number of transaction restarts, average wasted computations, ... *etc.* These secondary measures (although not presented in this paper for reasons of space) help explain the behavior of the algorithms under various loading conditions.

4.4 Parameter Settings and the Baseline Model

We started our experiments by first developing a baseline model around which we conducted further experiments, varying a few parameters at a time. Table 1 lists, the values assigned to the workload parameters in our baseline model. The database consisted of 1,000 pages from which each transaction accessed 20 pages randomly. The probability of a page been updated was set at 25%. These parameter settings are comparable to those used in similar studies [Hari90].

Figures 9-a and 9-b depict the average number of transactions that missed their deadlines, and the extra time needed by late transactions – those missing their deadlines – to complete their operations, respectively. The performance of both algorithms is identical when the number of transactions in the system is small. But, as the multiprogramming level in the system increases,

the superiority of the SCC-2S becomes evident. Not only do transactions running under the SCC-2S algorithm make most of their deadlines, but also the amount of time by which late transactions miss their deadlines is considerably smaller.

The reason that SCC-2S outperforms OCC-BC can be attributed to the fact that SCC-2S manages to preserve a large portion of the computation performed by each individual transaction. More precisely, when a transaction – say T – has to be aborted because of a conflict with another committing transaction, it does not have to restart from the very beginning, as it does under the OCC-BC algorithm. This means that some of the pages that were read or updated by transaction T will not need to be read or written again. This property of SCC-2S is especially advantageous when the number of data conflicts in the system is high.

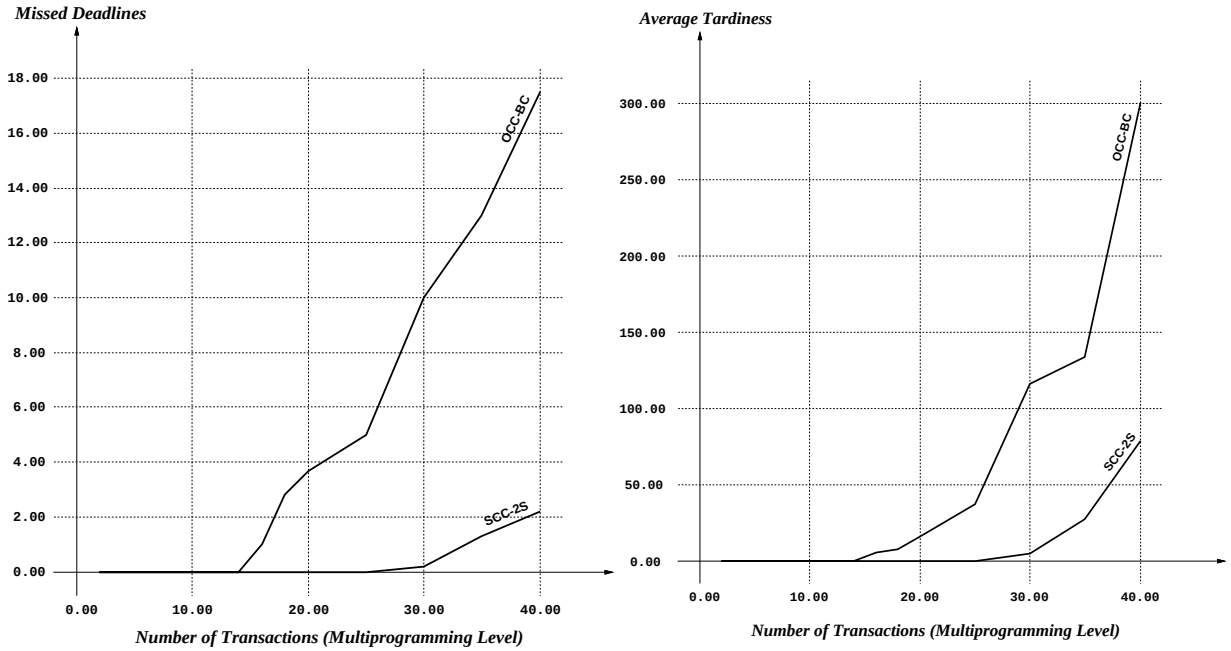

Figure 9: OCC-BC vs SCC-2S. Baseline Model (a) Missed Deadlines (b) Average Tardiness

The performance gained by using SCC-2S does not come for free. The cost incurred to set-up speculative shadows is translated to extra *control* messages that have to be communicated with the server. Our simulations confirmed this fact. A 15%-increase in the average number of messages exchanged with the server was observed for our baseline model. However, it can be shown that although the number of messages exchanged under SCC-2S increases, the total size of the exchanged messages is significantly reduced. This is due to the fact that under SCC-2S the number of pages read or updated decreases (as explained before), and due to the fact that control messages are *much* shorter than data access messages.

It is worthwhile to mention that we reached the same conclusions presented above (vis-a-vis the number of Missed Deadlines, Average Tardiness, and Overhead Messages) when we experimented with a database residing in the main memory of the server's machine.

4.5 Deadline Tightness

In the next set of experiments we examined the effect of deadline tightness on the relative performance of the two algorithms. For this reason we varied the Slack Ratio while keeping all the other parameters the same as those of the baseline model. We present here two experiments for Slack Ratios of 0.7 and 2.0, respectively. The corresponding graphs are shown in figure 10 and figure 11, respectively.

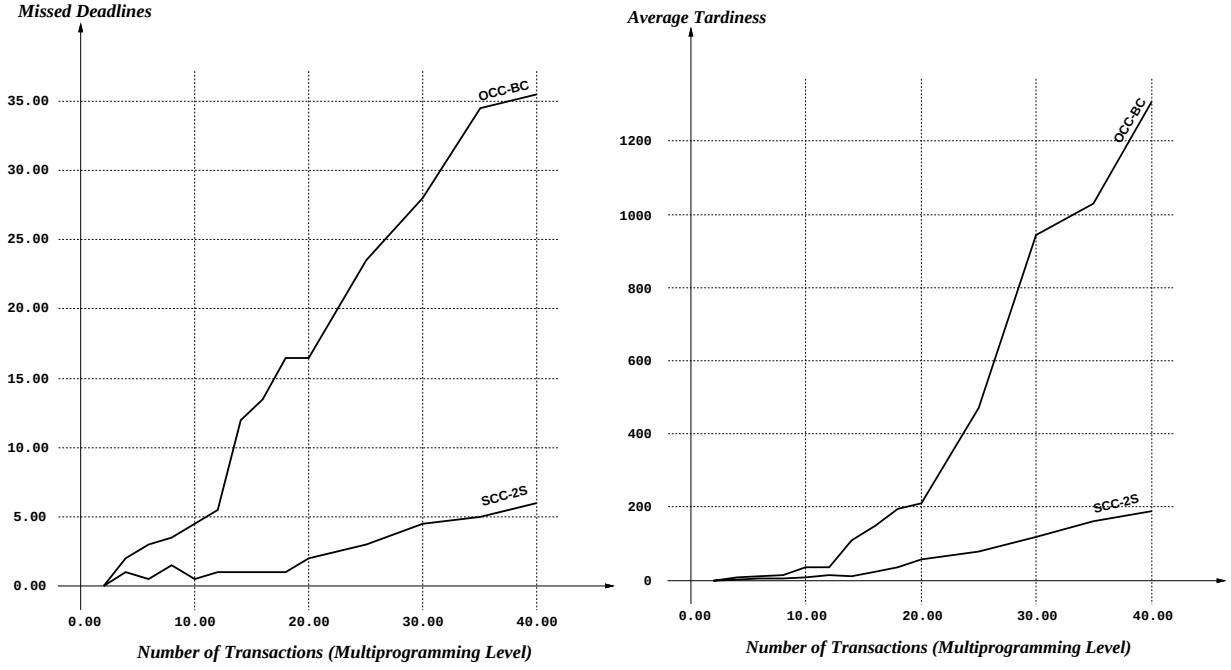

Figure 10: Slack Ratio of 0.7 (a) Missed Deadlines (b) Average Tardiness

At high Slack Ratios, both algorithms miss very few deadlines – with SCC-2S performing consistently better in all multiprogramming levels. However, as the Slack Ratio value decreases, and the system operates under very tight deadlines, the performance of the OCC-BC algorithm degrades rapidly, while the SCC-2S algorithm remains quite stable. Analogous results have been observed for Average Tardiness, with the gap between the two algorithms being even bigger.

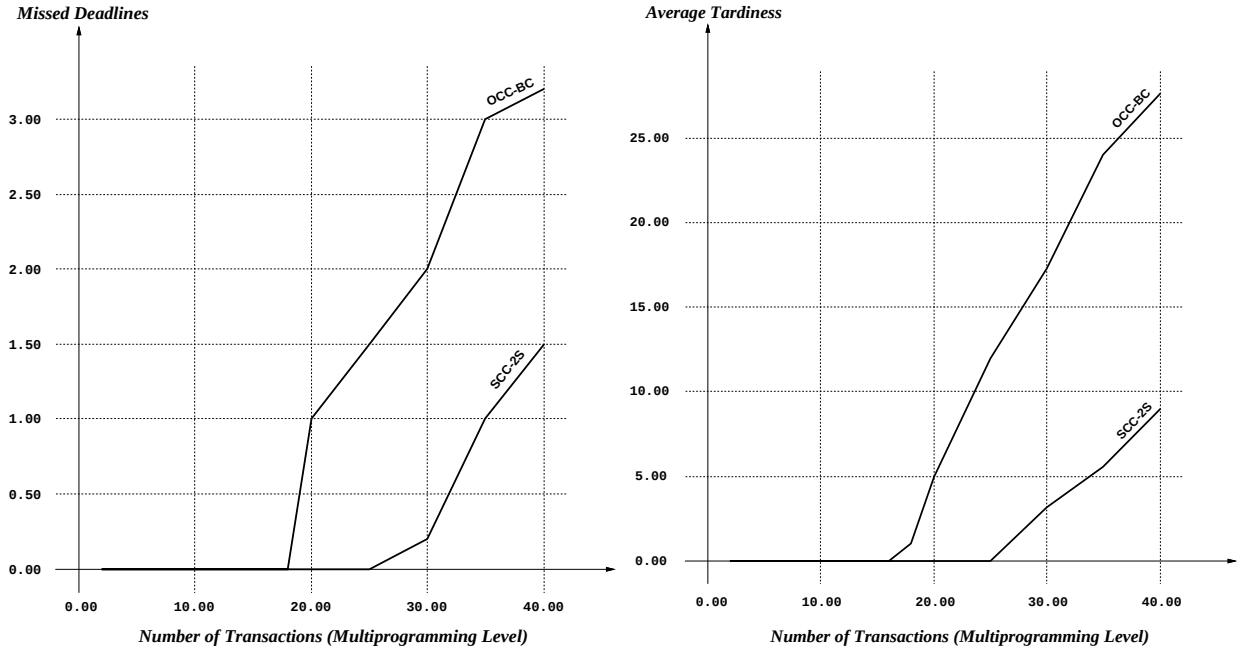

Figure 11: Slack Ratio of 2.0 (a) Missed Deadlines (b) Average Tardiness

4.6 Data Contention

We have experimented with different data contention levels by varying the write probability, `WProb`. The `SRatio` factor was fixed to 1.5 for all the measurements taken. Figure 12-a depicts the number of transactions missing their deadlines when the database consists of 1000 pages and each transaction updates half of the pages it accesses (`DBSize` = 1000 and `WProb` = 50%). As we can see, the OCC-BC algorithm missed almost 50% of its deadlines, whereas its SCC-2S counterpart missed only around 10%. The results obtained with a `DBSize` of 500 pages and a `WProb` of 50% (see Figure 12-b) are even more compelling as OCC-BC misses almost 70% of its deadlines, whereas SCC-2S appears more stable with only 12% of the transactions missing their deadlines.

4.7 Firm Deadlines

All of the previous experiments assumed a soft deadline policy, where all transactions have to be run to completion. When a firm deadline policy is adopted, whereby late transactions are immediately discarded from the system, both algorithms behaved considerably better than before. However, their *relative* performance was similar to that seen in the previous experiments. This improved behavior is due to the fact that discarding transactions that already missed their deadlines results in the availability of more resources for the remaining transactions in the system. This, also, has a positive effect on the system load as well as the degree of data contention exhibited in the system.

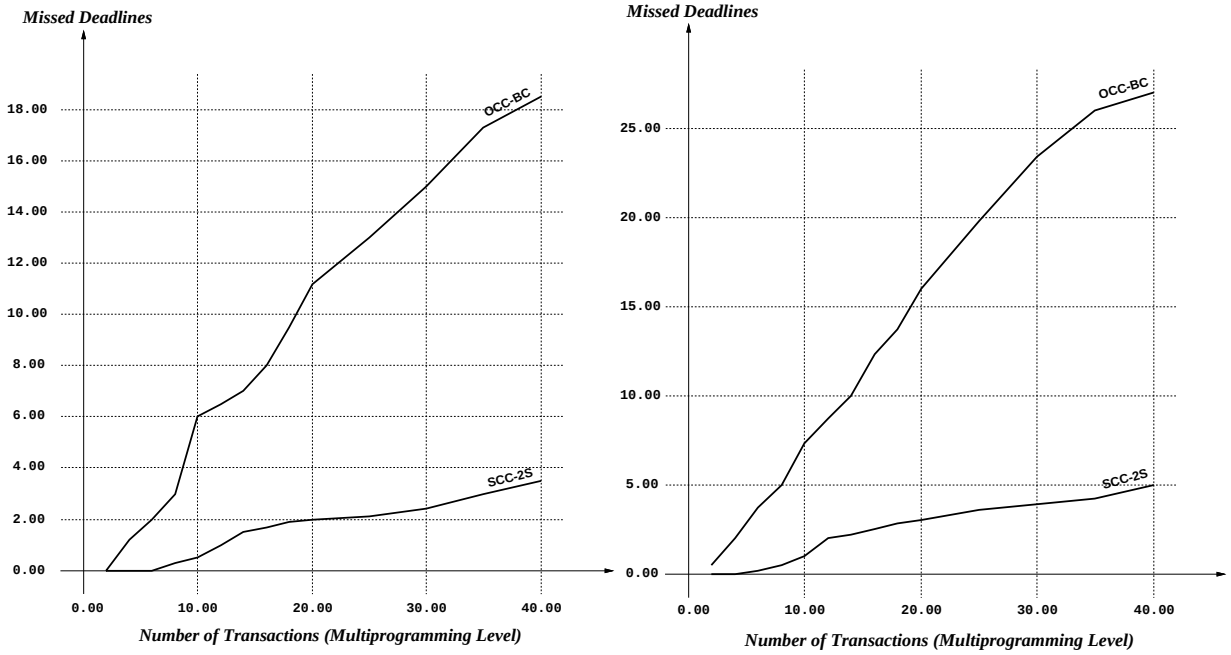

Figure 12: Miss Ratio for `WProb` of 50% (a) `DBSize` = 1000 (b) `DBSize` = 500

4.8 Deadline-cognizant SCC-2S

In RTDBS, traditional concurrency control algorithms are often augmented with heuristics that make such algorithms deadline- and/or priority-cognizant. We have developed an extension of SCC-2S that uses information about transaction deadlines to decide whether a validating transaction should be committed immediately, or whether its commitment should be delayed in favor of more urgent, conflicting transactions. This delay is similar to the waiting introduced in the Wait-50 heuristic [Hari90], except that we apply it to SCC-2S instead of OCC-BC. Initial investigation of this heuristic suggests only minor improvement over the original SCC-2S. The insignificance of the improvement can be explained by noticing that, thanks to speculation, the penalty incurred by a transaction as result of another transaction's commit is smaller. This results in a smaller payoff if delayed commitment is adopted. We are currently investigating other heuristics that combine deadline and priority information into value functions (similar to those suggested in [Huan89]) to be used in an integrated probabilistic scheme for shadow allocation and delayed commitment. The objective of this scheme is to maximize the expected value-added to the system, and not necessarily the number of satisfied timing constraints [Brao94].

5 Conclusion

SCC allows several *shadow* transactions to co-exist on behalf of a given uncommitted transaction so as to protect against the hazards of *blockages* and *restarts*, which are characteristics of PCC-based and OCC-based algorithms, respectively. In this paper, we reviewed a number of SCC-based protocols and described SCC-kS, a protocol that limits the number of processes alloted per transaction to a constant k. To evaluate the premise of SCC-based algorithms, extensive experiments were performed for two representative algorithms: OCC with Broadcast Commit (OCC-BC) and Two-Shadow SCC (SCC-2S). Our experiments indicate that SCC-2S offers significant performance improvements over OCC-BC for a wide range of system loads.[3] Therefore, from a performance standpoint, we argue that SCC-based protocols appear generally better suited than OCC-based protocols for RTDBS.

Speculation can be viewed as a mechanism for the distribution of risk. Instead of relying completely on one serialization order assumption—be it pessimistic or optimistic—a transaction is allowed to probe a host of serialization orders so as to minimize the impact of blockages and rollbacks. In this paper the distribution of risk was done without regard to the *probability* of the risks involved. In particular, if two transactions conflict, then the lower priority transaction has a larger risk of being aborted by the higher priority transaction. Similarily, a transaction with a loose deadline has a larger risk of being aborted by a transaction with a tight deadline. Currently, we are investigating a framework that would tie speculation to hazard probabilities.

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.

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

[Best93a] Azer Bestavros. "Speculative Concurrency Control." Technical Report TR-93-002, Computer Science Department, Boston University, Boston, MA, February 1993.

[3] More experiments on the effects of write page probabilities, database and transaction sizes, under conditions of heavy loading, high data contention, and tight deadlines, were conducted. While not discussed in this paper for space limitations, these experiments reinforced our conclusion regarding the superiority of SCC-2S.

[Best93b] Azer Bestavros and Spyridon Braoudakis. "SCC-nS: A family of Speculative Concurrency Control Algorithms for Real-Time Databases." In *Proceedings of the Third International Workshop on Responsive Computer Systems*, Lincoln, NH, September 1993.

[Boks87] 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.

[Brao94] Spyridon Braoudakis. *Concurrency Control Protocols for Real-Time Databases.* PhD thesis, Computer Science Department, Boston University, Boston, MA 02215, expected June 1994.

[Buch89] 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.

[Eswa76] 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.

[Gray76] 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.

[Hari90] 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.

[Hari92] Jayant R. Haritsa, Michael J. Carey, and Miron Linvy. "Data access scehduling in firm real-time database systems." *The Journal of Real-Time Systems*, 4:203–241, 1992.

[Huan89] 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.

[Huan91] Jiandong Huang, John A. Stankovic, Krithi Ramamritham, and Don Towslwy. "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.

[Huan92] Jiandong Huang, John A. Stankovic, Krithi Ramamritham, and Don Towsley. "Priority inheritance in soft real-time databases." *The Journal of Real-Time Systems*, 4:243–268, 1992.

[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.

[Papa79] Christos Papadimitriou. "The serializability of concurrent database updates." *Journal of the ACM*, 26(4):631–653, October 1979.

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

[Sha88] Lui Sha, R. Rajkumar, and J. Lehoczky. "Concurrency control for distributed real-time databases." *ACM, SIGMOD Record*, 17(1):82–98, 1988.

[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.

[Sing88] Mukesh Singhal. "Issues and approaches to design real-time database systems." *ACM, SIGMOD Record*, 17(1):19–33, 1988.

[Son92] Sang H. Son, Juhnyoung Lee, and Yi Lin. "Hybrid protocols using dynamic adjustment of serialization order for real-time concurrency control." *The Journal of Real-Time Systems*, 4:269–276, 1992.

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

A.1 The SCC-kS Class of Algorithms

For each transaction T_r, we keep a variable $SpecNumber(T_r)$, which counts the number of the speculative shadows currently executing on behalf of T_r. We, also, maintain for each shadow T_r^i of transaction T_r the $ReadSet(T_r^i)$ and the $WriteSet(T_r^i)$, and for each speculative shadow T_r^j the $WaitFor(T_r^j)$ as described in section 3. We use the notation: $(X, \_) \in ReadSet(T_r^i)$ to mean that shadow T_r^i read object X. We use $(T_u, \_) \in WaitFor(T_r^i)$ to denote the existence of at least one tuple (T_u, X) in $WaitFor(T_r^i)$, for some object X.

SCC-kS 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 accepts as 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 accepts 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.

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

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

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 denote by $\mathcal{T}_r^{\mathcal{S}}$ the set of speculative shadows executing on behalf of transaction T_r. The SCC-kS algorithm is described as a set of five rules which are described below.

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.

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

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

1. $ReadSet(T_r^o) \leftarrow \{(X, \_)\}$;
   for all T_u^o in $\mathcal{T}^{\mathcal{O}}$, such that $X \in WriteSet(T_u^o)$ **do**
2. **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**{
2.1 A new speculative shadow T_r^j is forked off T_r^o;
2.2 $WaitFor(T_r^j) \leftarrow \{(T_u, X)\}$;
2.3 $SpecNumber(T_r) \leftarrow SpecNumber(T_r) + 1\}$;

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

1. $WriteSet(T_u^o) \leftarrow \{X\}$;
   for all T_r^o in $\mathcal{T}^{\mathcal{O}}$, such that $(X, \_) \in ReadSet(T_r^o)$ **do**
2. **if** $(SpecNumber(T_r) < n-1)$ **then**{
2.1 **if** $(\forall T_r^i \in \mathcal{T}_r^{\mathcal{S}}, (T_u, \_) \notin WaitFor(T_r^i))$ **then**{
2.1.1 A new speculative shadow T_r^j is forked off $BestShadow(T_r, X)$;
2.1.2 $WaitFor(T_r^j) \leftarrow \{(T_u, X)\}$;
2.1.3 $SpecNumber(T_r) \leftarrow SpecNumber(T_r) + 1$
2.2 }**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**{
2.2.1 T_r^k is aborted and replaced by T_r^m which is forked off $BestShadow(T_r, X)$;
2.2.2 $WaitFor(T_r^m) \leftarrow \{(T_u, X)\}\}$;
3. }**else if** $(SpecNumber(T_r) = n-1)$ **then**
3.1 **if** $(\exists T_r^k \in \mathcal{T}_r^{\mathcal{S}} : (X, \_) \in ReadSet(T_r^k))$ **then**
3.1.1 Abort $LastShadow(T_r)$;
3.1.2 A new speculative shadow T_r^m is forked off $BestShadow(T_r, X)$;
3.1.3 $WaitFor(T_r^m) \leftarrow \{(T_u, X)\}\}$;

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 $LastShadow(T_u)\}$;