In Proceedings of VLDB'95: The International Conference on Very Large Databases, Zurich, Switzerland, September 1995.

Value-cognizant Speculative Concurrency Control

Azer Bestavros
Computer Science Department
Boston University, MA 02215
best@cs.bu.edu

Spyridon Braoudakis
Computer Science Department
Boston University, MA 02215
sb@cs.bu.edu

Abstract

We describe SCC-kS, a Speculative Concurrency Control (SCC) algorithm that allows a DBMS to use efficiently the extra computing resources available in the system to increase the likelihood of timely commitment of transactions. Using SCC-kS, up to *k shadow* transactions execute speculatively on behalf of a given uncommitted transaction so as to protect against the hazards of *blockages* and *restarts*. SCC-kS allows the system to scale the level of speculation that each transaction is allowed to perform, thus providing a straightforward mechanism of trading resources for timeliness. Also, we describe SCC-DC, a *value-cognizant* SCC protocol that utilizes *deadline* and *criticalness* information to improve timeliness through the controlled *deferment* of transaction commitments. We present simulation results that quantify the performance gains of our protocols compared to other widely used concurrency control protocols for real-time databases.

1 Introduction

For DataBase Management Systems (DBMS) with limited resources, performance studies of concurrency control methods (*e.g.* [ACL87]) have concluded that Pessimistic Concurrency Control (PCC) protocols [EGLT76, GLPT76] perform better than Optimistic Concurrency Control (OCC) techniques [BCFF87, KR81]. The main reason for this good performance is that PCC's blocking-based conflict resolution policies result in resource conservation. While abundant resources are usually not to be expected in conventional DBMS, they are more common in Real-Time DataBase Systems (RTDBS), which are designed to cope with rare high-load conditions, rather than normal average-load conditions. RTDBS are engineered not to guarantee a particular throughput, but to ensure that in the rare event of a highly-loaded system, transactions complete before their set deadlines [BMHD89]. These design goals often lead to a computing environment with far more resources than what would be necessary to sustain average loads, thus vanishing the advantage of PCC over OCC algorithms. In particular, OCC algorithms become attractive since computing resources wasted due to restarts do not adversely affect performance [HCL90b, HCL90a].

Real-time concurrency control schemes considered in the literature could be viewed as extensions of either PCC-based or OCC-based protocols, whereby transactions are assigned priorities that reflect the urgency of their timing constraints. These priorities are used with PCC-based techniques [AGM88, ACL87, SZ88, HSTR89, Sin88, SRL88, SRSC91] to make it possible for urgent transactions to abort conflicting, less urgent ones (thus avoiding the hazards of blockages); and are used with OCC-based techniques [Kor90, HCL90b, HCL90a, HSRT91, KS91, LS90, SPL92] to favor urgent transactions when conflicting, less urgent ones attempt to validate and commit (thus avoiding the hazards of restarts).

In a recent study [Bes92], we proposed an approach to concurrency control that combines the advantages of both OCC and PCC protocols while avoiding their disadvantages. Our approach relies on the use of *redundant* computations to start on alternative schedules, as soon as 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). SCC protocols are particularly suitable for RTDBS because they reduce the negative impact of blockages and rollbacks, which are characteristics of PCC and OCC techniques. In our previous SCC studies, we did not make any use of transaction deadline or criticalness information. Nevertheless, our performance studies [BB94] demonstrated the superiority of

Permission to copy without fee all or part of this material is granted by the VLDB Endowment for purposes other than direct commercial advantage. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proc. of the 21st VLDB Conference, Zurich, Swizerland, 1995

SCC-based protocols to real-time OCC-based and PCC-based protocols, which use such information.

In this paper, we argue that SCC protocols provide for a very natural (and elegant) way of incorporating transaction deadline and criticalness information into concurrency control for RTDBS. They introduce a new dimension (namely redundancy) that can be used for that purpose: By allowing a transaction to use more (redundant) resources, it can achieve better *speculation* and hence improve its chances for a timely commitment. Thus, the problem of incorporating transaction deadline and criticalness information into concurrency control is reduced to the problem of rationing system resources amongst competing transactions, each with a different payoff to the overall system. In section 2, we introduce the basic idea behind speculation. Next, the SCC-kS protocol, a practical speculative technique that operates under a limited speculation (resources) assumption, is presented. In section 3, we present the SCC-DC protocol, which extends SCC-kS to allow the use of deadline and criticalness information to improve timeliness. Also, SCC-VW, a simplified, efficient version of the SCC-DC protocol is presented. In section 4, we present simulation results that show the improvements achievable by SCC-based algorithms over other widely used protocols.

2 Speculative Concurrency Control

A major disadvantage of basic OCC [KR81] when used in RTDBS is that transaction conflicts are not detected until the validation phase, at which time it may be too late to restart. The Broadcast Commit (OCC-BC) variant of classical OCC [MN82, Rob82] attempts to solve this problem by a notification process, whereby a committing transaction notifies all concurrently running, conflicting transactions about its commitment. All such conflicting transactions are immediately restarted. OCC-BC detects conflicts earlier than the basic OCC algorithm resulting in less wasted resources and earlier restarts. The SCC approach proposed in [Bes92] 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 additional (redundant) resources to start on *speculative* corrective measures as soon as the conflict in question develops. By starting on such measures as early as possible, the likelihood of meeting set timing constraints is greatly enhanced.

To elucidate this point, consider 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 a SCC algorithm allows T_2 to have two shadows to account for the conflict with T_1, whereby one of these is committed depending on the time needed for transaction T_2 to reach its validation phase. In figure 2, T_2 reaches its validation phase before T_1, resulting in the validation and commitment of T_2 without any need to disturb T_1. Obviously, once T_2 commits, the shadow transaction T_2' has to be aborted. In figure 3, T_1 reaches its validation phase before T_2. With OCC-BC, T_2 is restarted when T_1 validates and commits as illustrated in figure 1. The SCC protocol instead of restarting T_2, simply aborts T_2 and adopts its shadow transaction T_2', thus improving the chances of meeting T_2's deadline.

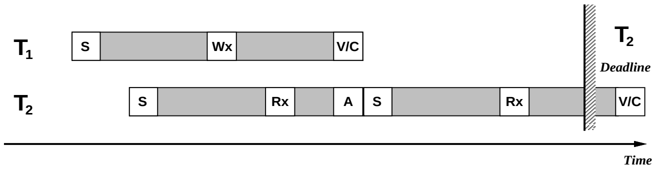

Figure 1: OCC-BC: Illustrative scenario

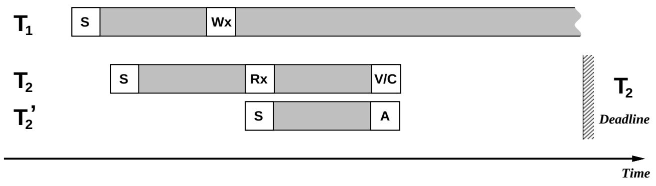

Figure 2: SCC: Illustrative scenario #1

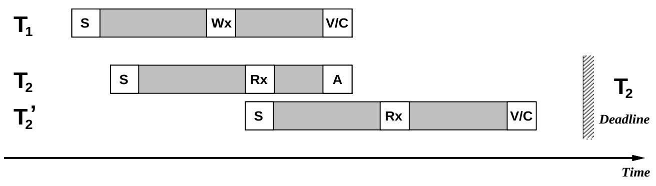

Figure 3: SCC: Illustrative scenario #2

The above notion of "speculation" could be generalized, whereby we associate with each transaction T_r as many shadows as there are *Speculated Orders of Serialization* (SOS). This leads to what we have termed the Order-Based SCC (SCC-OB). A SCC-OB algorithm requires a large amount of redundancy. If transaction T_r is one of n pairwise conflicting transactions, then SCC-OB may require T_r to fork an exponential number of shadows [Bra94], namely: $\sum_{i=1}^{n} \frac{(n-1)!}{(n-i)!} = \mathcal{O}\left((n-1)!\right)$.

The SCC-OB algorithm can be optimized so as to reduce significantly the number of shadows that may be required per transaction. In particular, if we allow a shadow to account for multiple serialization orders (i.e. the relationship between shadows and SOS is *one-to-many*), then it can be shown that only a linear number of shadows is sufficient to yield all the power of SCC-OB. Such an optimized algorithm, called Conflict-Based SCC (SCC-CB), is detailed in [Bra94]. At any point in time, SCC-CB needs no more than n shadows per transaction, and over the course of a transaction execution, no more than $\sum_{i=1}^{n}(n-i)$, or $\frac{n(n-1)}{2}$ shadows are created.

2.1 The K-Shadow SCC (SCC-kS) Algorithm

SCC-kS is a class of SCC algorithms that operate under a *limited resources* assumption, allowing no more than k shadows to execute on behalf of any given uncommitted transaction in the system. 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 $i = 0, \ldots, k-1$.

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.

Each speculative shadow T_r^s executes with the assumption that it will finish before any conflicts with other uncommitted transactions materialize, except for one conflict which is *speculated* to materialize before the commitment of T_r. Thus, T_r^s remains blocked on a shared object (say 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, (*i.e.* 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 followed a wrong SOS.

The value of k (the upper limit on the number of shadows allowed per transaction) does not have to be the same for all transactions. Foe a particular transaction, k reflects the amount of speculation that this transaction is allowed to perform (and thus the amount of resources it is allowed to consume). Thus, k is set to a value that reflects the transaction's urgency (how tight is the deadline) and criticalness. The value of k may change within the course of a transaction execution to reflect changes in the relative importance of that transaction compared to all other transactions in the system. For simplicity of presentation, and without loss of generality, we assume that k is constant and identical for all transactions.

Let $\mathcal{T} = T_1, T_2, T_3, \ldots, T_m$ be the set of uncommitted transactions in the system. Let $\mathcal{T}^{\mathcal{O}}$, and $\mathcal{T}^{\mathcal{S}}$ be the sets of optimistic, and speculative shadows executing on behalf of the transactions in the set $\mathcal{T}$, respectively. We use the notation $\mathcal{T}_r^{\mathcal{S}}$ to denote the set of speculative shadows executing on behalf of transaction T_r, and $SpecNumber(T_r)$ to denote the number of these shadows. 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. We use the notation: $(X, \_) \in ReadSet(T_r^i)$ to mean that shadow T_r^i read object X. $WriteSet(T_r^i)$ contains a list of all objects X written by shadow T_r^i.

For each speculative shadow T_r^i, 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. 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. The SCC-kS algorithm is described by the following set of five rules.

Start Rule: When a transaction T_r is started, an optimistic shadow T_r^o is created and the $SpecNumber(T_r)$, $ReadSet(T_r^o)$, and $WriteSet(T_r^o)$ are initialized.

Read Rule: When a read-after-write conflict is detected, if the maximum number of speculative shadows for the transaction, T_r, is not exhausted, a new speculative shadow T_r^s is started (by forking it off T_r^o) to account for this new conflict. Otherwise, this conflict is ignored. The Commit Rule below insures that corrective measures are taken, should this conflict materialize.

Write Rule: When a write-after-read conflict is detected, speculative shadows cannot be forked off, as before, from the reader transaction's optimistic shadow. This is because the conflict is detected on another transaction's write operation. Therefore, since its optimistic shadow already read that database object, we must either create a new copy of the reader transaction or choose another point during its execution from which we can fork. Figure 4 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 is not an appropriate shadow to fork off because, like the optimistic shadow, it already read X.

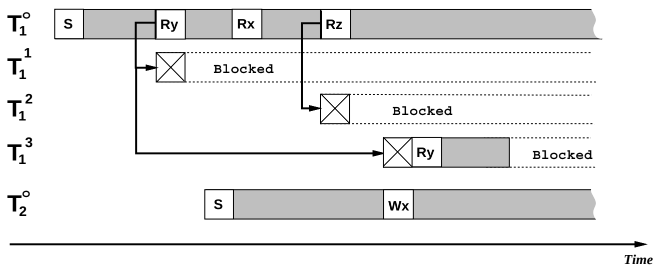

Figure 4: T_1^3 is forked off T_1^1.

When a new conflict implicates transactions that already conflict with each other, some adjustments may be necessary. In figure 5, 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.

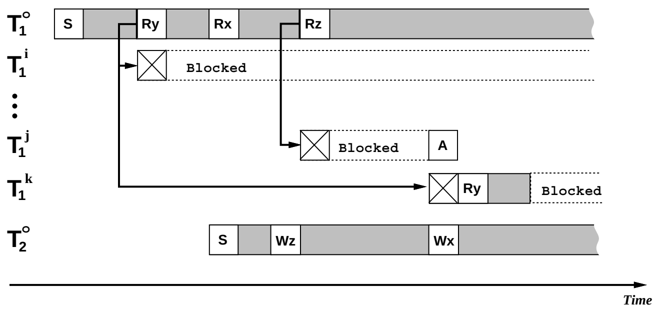

Figure 5: Example of multiply conflicting transactions.

The limit of at most $k-1$ speculative shadows per transaction does not preclude a transaction T_r from developing more than $k-1$ conflicts at any point during its lifetime. Rather, this limit is on the number of conflicts that SCC-kS will be ready to deal with in a timely manner. Choosing Which conflicts should be accounted for by speculative shadows is an interesting problem. In [BB94] we have adopted a *Latest-Blocked-First-Out* (LBFO) *shadow replacement policy* that 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. LBFO is one of several policies that could be adopted. In [Bra94] some alternative policies that account for the *most probable* serialization orders based on deadline and priority information are described and evaluated.

Blocking Rule: This 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 SOS.

Commit Rule: This rule is used when it is decided to commit an optimistic shadow T_r^o on behalf of a transaction T_r. First, all shadows in $\mathcal{T}_r^{\mathcal{S}}$ are aborted. Next, each transaction T_u that conflicts with T_r is considered. Two cases exists: either there is a speculative shadow, T_u^i, waiting for T_r's commitment, or not. The first case is illustrated in figure 6, where T_1^2—having anticipated the correct SOS—is promoted to become the new optimistic shadow of T_1, replacing the old optimistic shadow which had to be aborted. Speculative shadow T_1^3—which like the optimistic shadow made an incorrect SOS—is aborted as well. The second case is illustrated in figure 7, where the commitment of T_2^o on behalf of transaction T_2 was not accounted for by any speculative shadow of T_1.[1] In this case, the shadow with the latest possible blocking point (before the (T_2, Z) conflict) is chosen 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.

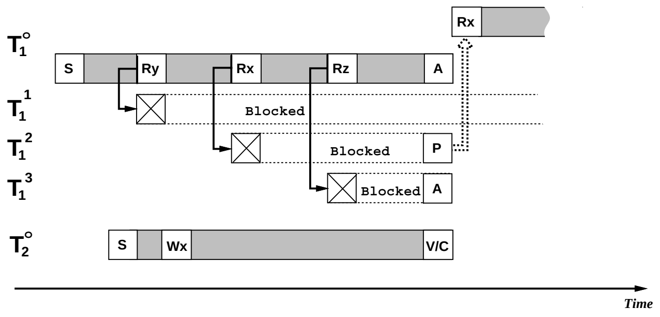

Figure 6: Applying the Commit Rule (case 1).

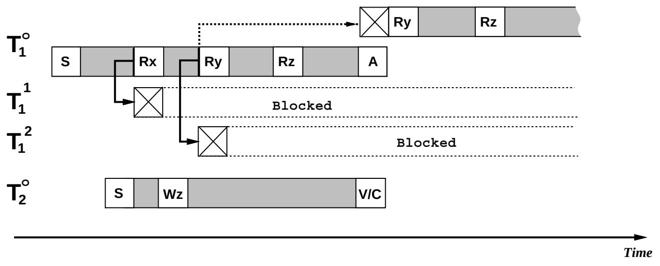

Figure 7: Applying the Commit Rule (case 2).

2.2 Two-Shadow SCC (SCC-2S)

SCC-2S allows a maximum of two shadows per uncommitted transaction to exist at any point in time: an *optimistic* shadow and a *pessimistic* shadow. The optimistic shadow runs under the assumption that it will be the first (among all other conflicting transactions) to commit, thus executing without incurring any blocking delays. The pessimistic shadow, on the contrary, is subject to blocking and restarts. It is kept ready to replace the optimistic shadow, should such a replacement be necessary. The pessimistic shadow runs under the assumption that it will be the last (among all other conflicting transactions) to commit.

SCC-2S resembles OCC-BC 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.

[1]Figure 7 makes the implicit assumption that transaction T_1 is limited to having at most two speculative shadows at any point during its execution.

3 Value Cognizant SCC

SCC-kS incorporates deadline and criticalness information into SCC by relating the relative *worth* of transactions to the amount of speculation (and thus resources) they are allotted. Nevertheless, SCC-kS is not *value-cognizant* because it does not make use of deadline and priority information in resolving data conflicts, or in making other scheduling decisions.

Previous concurrency control studies considered RT-DBSs where all transactions are of equal worth. The major performance objectives were to minimize the number of missed *firm* deadlines or to minimize *tardiness*—the time by which late transactions miss their *soft* deadlines. Under this approach all system transactions are assigned the same *value*. However, there exist real-time applications where different transactions may be assigned different values [SZ88, HSTR89] to reflect their relative worth to the system upon successful completion. For such systems the attention shifts to maximizing the *value-added* to the system by the transactions' commitment; minimizing tardiness or the number of missed deadlines becomes of secondary importance. Notice that a transaction's value and its deadline are two orthogonal properties [BSR88, HSTR89]. The fact that a transaction has a tight deadline does not in any way imply that it has a high value, nor does the fact that it has a loose deadline imply that it has a low value. Transactions with similar values may have different deadlines, while those with similar deadlines may have different values.

3.1 Transaction Value

The relationship between a transaction's value and the value-added to the system can be captured by the notion of *value functions* introduced by Jensen, Locke, and Tokuda [JLT85, Loc86]. Each transaction T_u is associated with a value function $V_u(t)$, which represents the value of T_u as a function of its completion (commit) time. A real-time application cashes on the full value of a transaction if it is committed on time. Otherwise, a penalty is assessed. We define the *penalty gradient*, to be the rate at which a transaction loses its value when it commits past its deadline.

Definition 1 *The* penalty gradient *of a transaction T_u with a value function of $V_u(t)$ and a deadline D_u is:*

$$\frac{d}{dt}V_u(t), \quad \text{for } t > D_u.$$

The penalty gradient is an important factor in RT-DBS performance studies because it indicates *how soft* deadlines are relative to each other. In this paper, we consider the case where the penalty gradients of transactions follow the formula: Penalty Gradient of $T_u = \tan\alpha_u$, for $t > D_u$. The penalty gradient of T_u may vary from infinity for a very critical transaction ($\alpha_u = \pi/2$), to zero for a non-critical transaction ($\alpha_u = 0$). Figure 8 depicts a typical value function. Transaction T_u has an arrival time of A_u and a soft deadline of D_u. If T_u completes its execution before its set deadline D_u its value-added to the system is v_u. On the other hand, if T_u misses its deadline the value-added to the system diminishes according to its penalty gradient $\tan\alpha_u$.

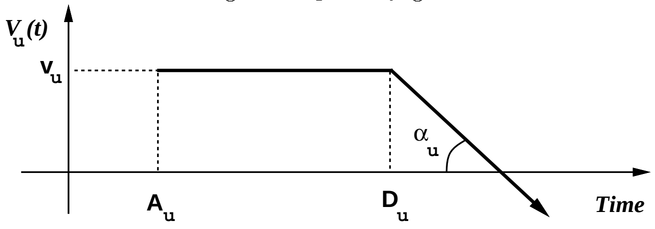

Figure 8: A typical value function for a transaction T_u

Definition 2 *The* value function $V_u(t)$ *of transaction T_u with arrival time A_u and soft deadline D_u is:*

$$V_u(t) = \begin{cases} v_u & \text{if } A_u \le t \le D_u \\ v_u - [(t - D_u)\tan\alpha_u] & \text{if } t > D_u \end{cases}$$

where v_u is the value-added to system if T_u completes its execution before its set deadline D_u, and $\tan\alpha_u$ *is its penalty gradient.*

3.2 SCC with Deferred Commit (SCC-DC)

Committing a transaction as soon as it validates may result in a value loss to the system. In figure 9, committing T_1 as soon as it is validated causes T_2 to miss its deadline and a value penalty to be assessed to the system. In [HCL90b], Haritsa showed that by delaying the commitment of a lower priority transaction, the number of transactions meeting their deadlines is increased. SCC-based protocols can benefit from the introduction of such delays by giving optimistic shadows more time to execute and commit instead of being aborted in favor of other validating transactions of lesser worth. Figure 10 shows the increased value-added to the system that results from delaying the commitment of T_1, thus allowing T_2^o to commit before its deadline and contribute a higher value to the system.

Our approach for introducing delays is similar to those proposed in [AAJ92, HCL90a, SPL92]. Whenever a shadow T_u^o finishes its execution, we evaluate if it is advantageous to defer T_u^o's commitment. Finding the *best* point in time to commit a finished shadow T_u^o is a very hard optimization problem, since it requires the consideration of all possible serialization orders of active transactions. To avoid the exponential nature of this problem, we propose a protocol, SCC with Deferred Commit (SCC-DC), which estimates the value-added to the system at *discrete* points in time (*e.g.* periodically). SCC-DC compares the estimated value-added to the system if the finished shadow T_u^o is committed at time t, to

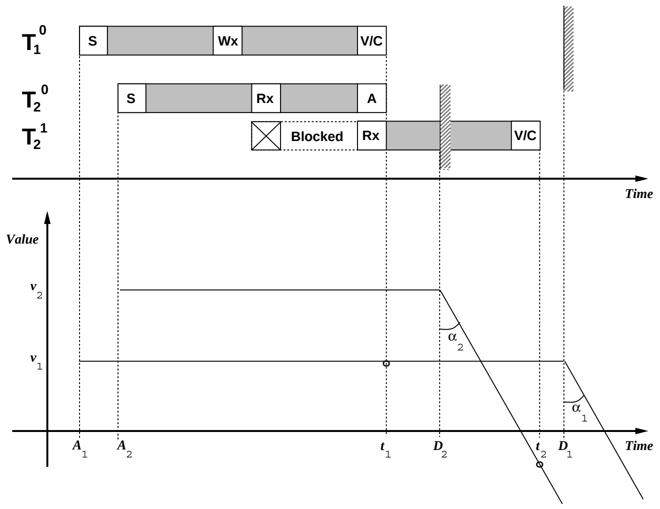

Figure 9: Value-added to the system without deferment

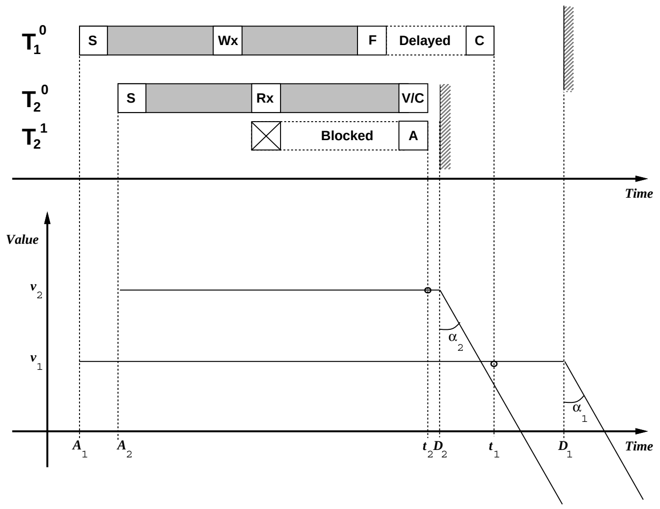

Figure 10: Value-added to the system with deferment.

the estimated value-added to the system if T_u^o is committed at time $t + \delta$, where δ is some constant delay. Because of its discrete nature, this algorithm does not always provide us with the *best* point in time to commit a shadow. This *optimal* point in time may well lie anywhere inside those time intervals.

Basic Definitions and Assumptions

Each transaction in the system T_u has an arrival time A_u and a deadline D_u. We classify transactions according to their run-time characteristics. We denote with C_u the *class* of transaction T_u. We assume that for each such class the profile of the execution time—how long it takes to finish a transaction of that class—is known. Such a profile (figure 11) can be obtained from collected statistics of the previous history of the system.

Definition 3 *The* finish probability density function *$F_u(x)$ denotes the probability that the execution time for a transaction in class C_u will not exceed x. We use E_{C_u} to denote the expected execution time for class C_u.*

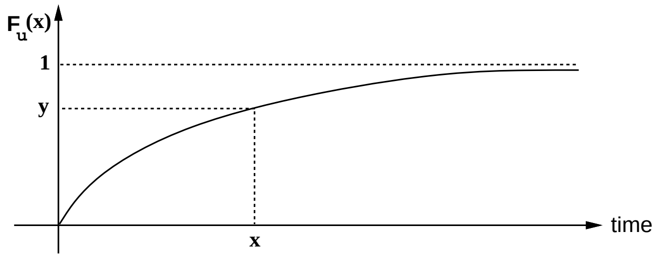

Figure 11: Typical finish probability density for C_u

3.3 Transaction Committment Protocol

We assume that a special clock exists to signal the points in time, when transactions may be committed. At each tick, we decide for each transaction shadow T_u^o that finished its execution whether to proceed and commit T_u^o, or defer its commitment for an additional clock tick δ. If the clock ticks at time t and T_u^o is a transaction shadow which has finished its execution, then:

- If T_u^o does not conflict with any other uncommitted transaction, then we commit it on behalf of T_u.
- Otherwise, if T_u conflicts with uncommitted transactions $T_1, \ldots, T_m$, then we compute the expected value-added, V_{now}, should we commit T_u^o at the current clock tick t, and the expected value-added, V_{later}, should we defer T_u^o's commitment to a later clock tick $t + k\delta$, for $k \in \mathcal{N}^*$. If $V_{now} \geq V_{later}$ then we commit T_u^o, otherwise we defer it.

Since more than one shadow may exist on behalf of an uncommitted transaction, the computation of the expected value-added to the system by that transaction depends on *which* shadow is committing and at *what* time. We define two measures, the *shadow finish probability* and the *shadow adoption probability*, which we use to assist in these computations.

Definition 4 *The shadow finish probability function $F_u^i(x)$ of shadow T_u^i denotes the probability of T_u^i finishing its execution by time x.*

$$F_u^i(x) = \text{Prob}[T_u^i \text{ will finish by time } x].$$

Assuming that shadow T_u^i has already executed for τ time units, then using the probability density function $F_u(x)$, the finish probability can be computed at time t_{now} by applying Baye's Theorem as follows:

$$\begin{aligned} F_u^i(x) &= \frac{\text{Prob}[T_u^i \text{ will finish before } x \text{ and after } \tau]}{\text{Prob}[T_u^i \text{ will finish after } \tau]} \\ &= \frac{F_u(x) - F_u(\tau)}{1 - F_u(\tau)}, \quad \text{for } x \geq \tau. \end{aligned}$$

In our model we favor transactions that have a high value-added to the system by using the transaction value functions in resolving data conflicts and making other scheduling decisions. This implies that a transaction shadow created to account for a conflict with a higher valued transaction is more likely to be adopted in the future than a shadow which is created to account for a conflict with a lesser valued transaction.

Definition 5 *The shadow adoption probability function $P_u^i(t)$ of shadow T_u^i of transaction T_u denotes, at time t, the probability that shadow T_u^i will be adopted in the future—i.e. the probability that the conflict that called for the creation of T_u^i will materialize.*

The shadow adoption probability functions capture the relative importance of the shadows of a transaction as a function of time. At time t, for a transaction T_u, they are computed as follows:

a. If T_u has no speculative shadows then $P_u^o(t) = 1$.

b. If T_u conflicts with $T_{r_1}, T_{r_2}, \ldots, T_{r_m}$ then:

$$\begin{aligned} P_u^o(t) &= \frac{V_u(t)}{V_u(t) + \sum_{j=1}^{m} V_{r_i}(t) P_{r_i}^o(t)} \\ P_u^i(t) &= \frac{V_i(t) P_i^o(t)}{V_u(t) + \sum_{j=1}^{m} V_{r_i}(t) P_{r_i}^o(t)}, r_1 \leq i \leq r_m \end{aligned}$$

where T_u^i is the shadow of T_u that accounts for the conflict between T_u and T_i.

Description of the SCC-DC Algorithm

We add to the SCC-kS protocol an additional rule which controls the commitment of transactions. The *Termination Rule* is invoked periodically by the system with a period of δ time units.

Let the *Termination Rule* be invoked at time t. For each transaction shadow T_u^o that has finished its execution there are two cases to be examined. If T_u does not conflict with any other uncommitted transaction, then T_u^o is committed on behalf of transaction T_u. Otherwise, if T_u conflicts with transactions T_1, T_2, ..., T_m, then V_{now} (the expected value-added to the system should T_u be committed at time t) is compared to V_{later} (the expected value-added to the system should T_u be committed at a later time $t + k\delta$, for $k \in \mathcal{N}^*$). If $V_{now} \geq V_{later}$ then T_u^o is committed on behalf of T_u, otherwise its commitment is deferred. We use two functions to compute V_{now} and V_{later}: the *Expected Finish probability* and the *Expected Value-added.*

Definition 6 *The Expected Finish probability function,* $\mathrm{EF}_u(x)$*, of transaction T_u at time t, is defined as the probability that some shadow of T_u will be able to finish its execution by time x.* $\mathrm{EF}_u(x)$ *is computed as the summation below over all j shadows of T_u.*

$$\mathrm{EF}_u(x) = \sum_j F_u^j(x) P_u^j(t)$$

Definition 7 *We denote by* $\mathrm{EV}_u(x)$ *the Expected Value-added to the system if transaction T_u commits at time x.*

$$\mathrm{EV}_u(x) = V_u(x)\mathrm{EF}_u(x)$$

V_{now} is the expected value-added from the commitment of shadow T_u^o at time t plus the expected value added from the commitment of T_1, T_2, ..., and T_m at a later time $t + k\delta$, for $k = 1$ to infinity. V_{later} is the expected value added from the commitment of T_u, T_1, T_2, ..., and T_m at some later time $t + k\delta$, for $k = 1$ to infinity.

$$\begin{aligned} V_{now} &= V_u(t) + \sum_{i=1}^{m} \sum_{k=1}^{\infty} EV_i(t + k\delta) \\ V_{later} &= \sum_{k=1}^{\infty} EV_u(t + k\delta) + \sum_{i=1}^{m} \sum_{k=1}^{\infty} EV_i(t + k\delta) \end{aligned}$$

The infinite summations above can be bounded by observing that for each transaction T_i there exist a time $t + k\delta$, for some $k = l_i$, where the expected finish probability of T_i, $EF_i(l_i\delta) = 1 - \epsilon$, where ϵ is an arbitrarily small number. We, therefore, bound these summations with appropriate $k = l_i$ values, introducing arbitrarily small errors. We are now ready to augment SCC-kS with a *Termination Rule* to be invoked periodically, every δ units of time.

Termination Rule: For each shadow T_u^o that finished executing:

- ◇ If T_u conflicts with no other uncommitted transactions, then invoke the *Commit Rule* to commit T_u^o.
- ◇ If T_u conflicts with T_1, T_2, ..., T_m, then:

$$\begin{aligned} V_{now} &= V_u(t) + \sum_{i=1}^{m} \sum_{k=1}^{l_i} EV_i(t + k\delta) \\ V_{later} &= \sum_{k=1}^{l_u} EV_u(t + k\delta) + \sum_{i=1}^{m} \sum_{k=1}^{l_i} EV_i(t + k\delta) \end{aligned}$$

- ◇ If $V_{now} \geq V_{later}$ then invoke the *Commit Rule* for T_u.

Two modifications to the rules of the SCC-kS algorithm are necessary. The first affects the *Commit Rule.* Under SCC-DC, transactions *do not* commit as soon as they finish execution. Rather, they wait (at least) until the next periodical invocation of the *Termination Rule.* Thus, the *Commit Rule* is invoked *only when* the *Termination Rule* decides to commit a shadow. The second modification affects the *Read* and *Write Rules.* Under SCC-DC, an optimistic shadow, T_r^o, can finish executing, yet its commitment may be deferred. While T_r^o is awaiting commitment, a conflict may develop with another shadow T_u^o. If T_u^o, also, finishes its execution, then it is possible under SCC-DC (depending on their relative worth), that T_u^o be committed, thus resulting in the abortion of the finished T_r^o shadow. To accomodate for this possibility, the *Read Rule* (*write Rule*) is extended, so as to be invoked when an optimistic shadow T_r^o wishes to read (write) an object X, which is written (read) by another shadow T_u^o, whether T_u^o is currently executing or has already finished its execution and is awaiting commitment.

3.4 SCC with Voted Waiting (SCC-VW)

SCC-DC requires a substantial computing overhead to determine whether or not to defer a transaction's commitment. SCC with Voted Waiting (SCC-VW) is an approximation heuristic that reduces that overhead of SCC-DC significantly. The main idea of the VW mechanism is to allow uncommitted transactions to *vote* for or against the commitment of a finished transaction (say T_u^o) based on the expected value-added to the system as a result of such a commitment. The votes are weighed based on the relative values of the participating transactions. The resulting measure is called the *commit indicator*, CI_u, for T_u^o. If $CI_u > \alpha$ (in this paper $\alpha = 0.5$) then T_u^o is committed, otherwise it waits.

Two measures are used in the computation of the commit indicator for a finished transaction shadow: the *commit vote*, cv_i^u, of a transaction T_i regarding the commitment of a finished conflicting transaction shadow T_u^o, and the relative *weight* function, $w_i(t)$, of T_i at time t.

Definition 8 *We define the commit vote,* cv_u^i*, of an executing transaction* T_i *with respect to a finished conflicting transaction shadow* T_u^o *to be:*

$$\mathrm{cv}_u^i = \begin{cases} 1 & \textit{if } T_i \textit{ votes to commit } T_u^o \\ 0 & \textit{if } T_i \textit{ votes not to commit } T_u^o \end{cases}$$

Definition 9 *The weight function,* $w_i(t)$*, of a transaction* $T_i \in \mathcal{T}^u$*, is a function of time given by the formula:*

$$w_i(t) = \frac{V_i(t)}{\sum_{T_k \in \mathcal{T}^u} V_k(t)},$$

where $\mathcal{T}^u$ *is the set of transactions that conflict with* T_u^o*, and* $V_k(t)$ *is the value function of* T_k*.*

Definition 10 *The commit indicator,* CI_u*, for a shadow* T_u^o *at time t, is the weighed summation of the commit votes of all conflicting transactions* $T_i \in \mathcal{T}^u$*.*

$$\mathrm{CI}_u(t) = \sum_{T_i \in \mathcal{T}^u} w_i(t) \times \mathrm{cv}_u^i.$$

Description of the SCC-VW Algorithm

Let T_i^u be the shadow of T_i that accounts for the conflict with T_u and E_{C_i} be the average execution time of a transaction from class C_i. Assuming that T_i^u has already executed for τ_i^u time units, the expected value-added to the system if T_i votes to commit T_u^o at the current time t is given by the addition of the expected value-added from the commitment of T_u^o at time t plus the expected value-added from the commitment of the T_i^u shadow of T_i at time $t + (E_{C_i} - \tau_i^u)$.

$$V_{now} = V_u(t) + V_i(t + E_{C_i} - \tau_i^u)$$

For the computation of the expected value-added to the system if T_u's commitment is to be delayed, we distinguish between two cases.

The first case occurs if T_u has no read-after-write conflict with T_i. In this case, the finished (optimistic) shadow of T_u can be committed as soon as the optimistic shadow of T_i completes its execution. This event is estimated to happen at time $t + E_{C_i} - \tau_i^o$. Assuming that, at time t, T_i^o has already executed for τ_i^o time units, we get:

$$V_{later} = V_i(t + E_{C_i} - \tau_i^o) + V_u(t + E_{C_i} - \tau_i^o),$$

The second case occurs if there exists a speculative shadow T_u^i of T_u accounting for a read-after-write conflict with T_i. In this case, the commitment of T_i^o at time $t + E_{C_i} - \tau_i^o$ will result in the abortion of T_u^o and its replacement by T_u^i. Assuming that T_u^i has already executed for τ_u^i time units, we get:

$$V_{later} = V_i(t + E_{C_i} - \tau_i^o) + V_u(t + E_{C_i} - \tau_i^o + E_{C_u} - \tau_u^i)$$

Termination Rule: When an optimistic shadow T_u^o finishes its execution, evaluate whether it is advantageous, to delay T_u^o's commitment.

- ◇ If T_u conflicts with no other uncommitted transactions, then invoke the *Commit Rule* to commit T_u^o.
- ◇ If T_u^o conflicts with the set $\mathcal{T}^u$, then:
  1. For every transaction $T_i \in \mathcal{T}^u$
     a. Compute V_{now} and V_{later}, and
     b. Determine the *commit vote*, cv_u^i, of T_i:

$$cv_u^i = \begin{cases} 1 & \text{if } V_{now} \geq V_{later} \\ 0 & \text{otherwise} \end{cases}$$

  2. Compute the *commit indicator* for T_u^o.

$$CI_u(t) = \sum_{T_i \in \mathcal{T}^u} w_i(t) \times cv_u^i$$

  3. If $CI_u(t) \leq \alpha$, then delay T_u^o's commitment, otherwise invoke the *Commit Rule* on T_u^o.

4 Performance Evaluation

In this section, we present a comparative evaluation of the following protocols: 2PL with Priority Abort (2PL-PA) [AGM88] as a representative of PCC-based protocols, OCC-BC [HCL90b] and WAIT-50 [HCL90a] as representatives of OCC-based protocols, and SCC-2S and SCC-VW as representatives of SCC-based protocols.

The RTDBS model that we used in our experiments consists of a multiprocessor DBMS operating on disk resident data. We assume an environment with abundant resources.[2] We consider that the time spent on performing concurrency control tasks is negligible and that dedicated processors are assigned for these tasks. The system model consists of five main modules as depicted

[2]This assumption allows us to phase out resource contention and measure the most concurrency achievable by each algorithm.

in Figure 12. Transactions which are ready to execute are maintained in a *Transaction Pool.* The *Transaction Manager* (TM) is responsible for making resource and concurrency control requests (*e.g.* read page, write page, request cpu, ... *etc.*) on behalf of active transactions. The *Resource Manager* (RM) allocates and deallocates system resources (e.g. CPU, disk, database pages) to requesting transactions. The *Concurrency Control Manager* (CCM) processes read and write requests from the TM. Once a transaction has either committed or aborted, it is removed from the system and sent at a *Transaction Sink.*

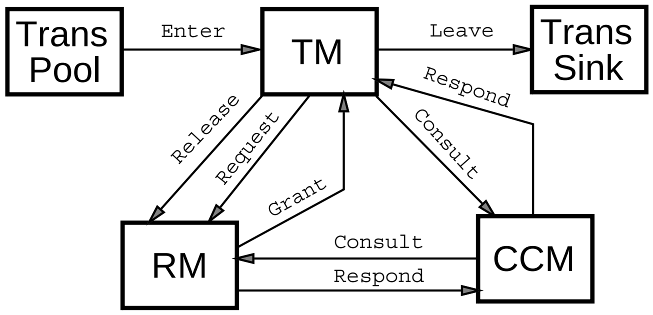

Figure 12: The Logical System Model

The primary performance measures that we employ are the percentage of transactions that miss their deadlines, *Missed Ratio*, and the average time by which late transactions miss their deadlines, *Average Tardiness.* A transaction that commits at or before its deadline has a tardiness of zero. A transaction that completes after its deadline has a tardiness of $T - Deadline$, where T is the transaction's completion time. The simulations also generated a host of other statistical information, including number of transaction restarts, average wasted computation, ... *etc.* These secondary measures were quite helpful in explaining the behavior of the algorithms under investigation.

4.1 Simulation Results

We consider a 1,000-page database from which each transaction accesses 16 randomly selected pages. The probability of a page being updated is set at 25%. The *slack factor* for the computation of transaction deadlines is set up at 2, and the *EDF* policy to assign transaction priorities (for 2PL-PA and Wait-50) is adopted. These parameter settings are comparable to those used in similar studies [HCL92]. Our experiments assumed that transaction deadlines are soft. This entails that late transactions (those missing their deadlines) must complete—nevertheless—with the minimum possible delay. Each simulation runs until at least 4,000 transactions are committed. Enough runs were performed to guarantee a 90% confidence interval. Unless otherwise stated, our figures depict the average over all experiments. Simulations were performed under a wide range of workloads to enable us to characterize the behavior of the protocols under the various conditions that may arise in a real-world RTDBS. For a comprehensive analysis of these simulations, we refer the reader to [Bra94].

Figures 13 and 14 depict the average number of transactions that missed their deadlines, and the extra time needed by late transactions to complete their operations, respectively. All protocols perform well when the number of transactions in the system is small. However, as the arrival rate of transactions in the system increases, their performance degrades at different rates. SCC-2S provides the most stable performance among the studied protocols. Its *Missed Ratio* is the lowest under all system loads. On the other hand, although Wait-50 performs well at low loads, its performance degrades fast, becoming even worse than OCC-BC at the higher system loads. It is remarkable that while at an arrival rate of 70 transactions per second, SCC-2S, Wait-50, and OCC-BC miss 1%, 1.5%, and 2.5% of their deadlines, respectively, at 150 transactions per second their respective *Missed Ratios* become 30%, 92%, and 78%. 2PL-PA showed consistently the worst performance among the tested protocols. Its performance degrades at much lower system loads and with a much higher slope. This is to be expected because the environment at which we performed our simulations (high data contention, tight deadlines) was particularly unfriendly to locking-based protocols.

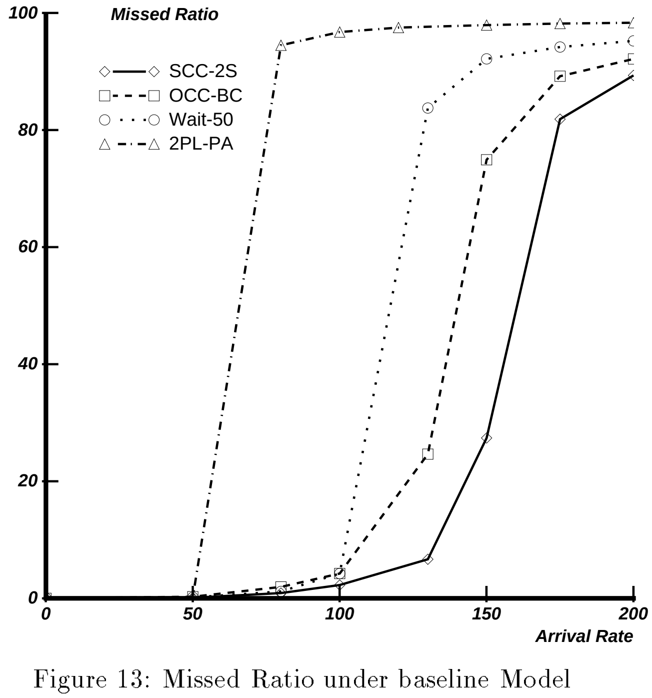

Figure 13: Missed Ratio under baseline Model

The superiority of SCC-2S becomes evident by observing that not only do transactions running under the SCC-2S algorithm make more of their deadlines, but also the amount of time by which late transactions miss their deadlines is considerably smaller. It is worthwhile to point out here that, although SCC-2S outperforms

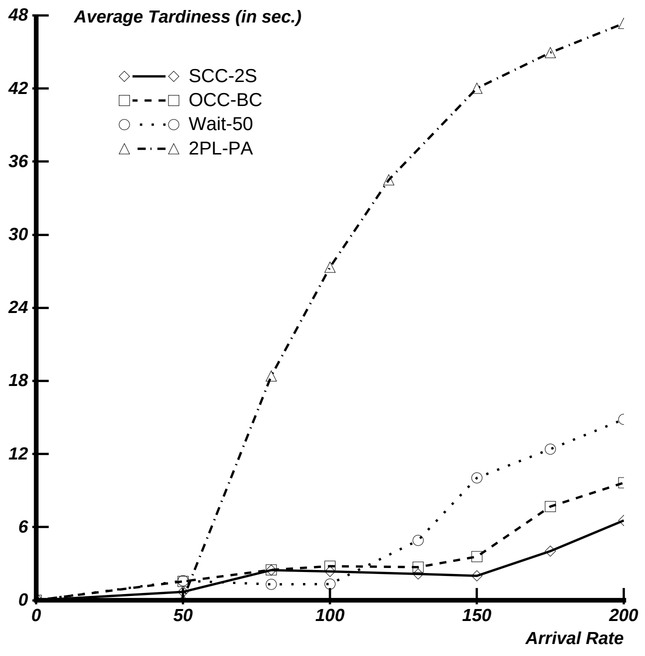

Figure 14: Average tardiness under baseline Model

OCC-BC with respect to *Average Tardiness* under all system loads (as figure 14 suggests), this is not the case when we consider Wait-50. On the contrary, Wait-50 has a relatively better *Average Tardiness* performance for the lower system loads, which it loses only when the system load becomes considerably high (at arrival rates above 125 transactions per second). This result can be attributed to the fact that SCC-2S is not a deadline-cognizant protocol, unlike Wait-50 which utilizes this information to make better decisions regarding "*when* to commit transactions". However, at high loads Wait-50—because of its higher *Missed Ratio* (relative to SCC-2S)—loses this advantage.

4.2 SCC-VW and System Value

Our previous experiments considered a RTDBS which was operating under the assumption that all transactions in the system were equally important. The two major performance objectives were to minimize the *Missed Ratio* and minimize the *Average Tardiness* of the system. In this section, we lift this assumption, allowing transactions to have different *values*, to reflect their relative worth to the system upon commitment. The major performance objective for such a system is to maximize the expected *value-added* to the system by the completed transactions. Minimizing tardiness and the number of missed deadlines becomes of secondary importance. We call the new performance measure the *System Value*.

In the following experiments, we report on the performance of SCC-VW (as an SCC-based protocol which incorporates transaction values in its decision making). Our results suggest only minor improvement over the original SCC-2S protocol. In particular, figure 15 depicts the *System Value* for the protocols in question, where all transactions are assigned the same *value function*.[3] 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 (like the one employed by SCC-VW) is adopted. An interesting observation of our experiments is that although SCC-VW improved the value-added to the system, it misses more deadlines relative to SCC-2S as figure 16 suggests. This is because, as we explained above, SCC-VW's objective is to maximize the expected *System Value*, and not necessarily the number of satisfied timing constraints. This observation is reinforced by viewing the *Average Tardiness* results shown in figure 17. There, SCC-VW provides a smaller *Average Tardiness* result compared with SCC-2S. In other words, although SCC-VW misses more deadlines than SCC-2S, it misses them by a smaller margin.

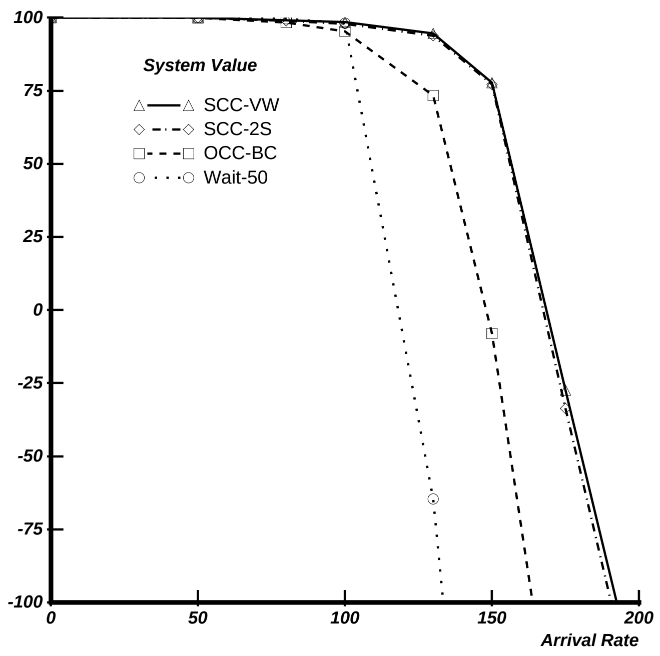

Figure 15: System Value under baseline model (1 class)

We have performed more experiments to evaluate the relative performance of the algorithms in a RTDBS where transactions belong to different classes, each with different *value functions* and different *execution profiles*. Our results show that SCC-VW performs better under such conditions. Figure 18 shows a sample simulation for a RTDBS with two classes of transactions. The first class is characterized by long execution times, tight deadlines, high value-added (when committed on time), and large penalty gradients. Alternately, the second class is characterized by short execution times, lower value-added,

[3]The value added is constant if the deadline is met, otherwise a penalty gradient of -1 is assessed. All other parameters are set to those of the baseline model.

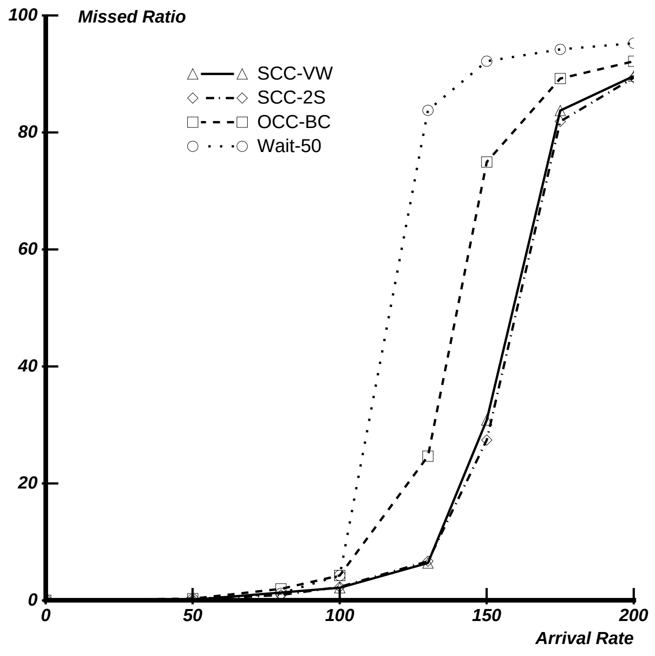

Figure 16: SCC-VW: Missed ratio under baseline model

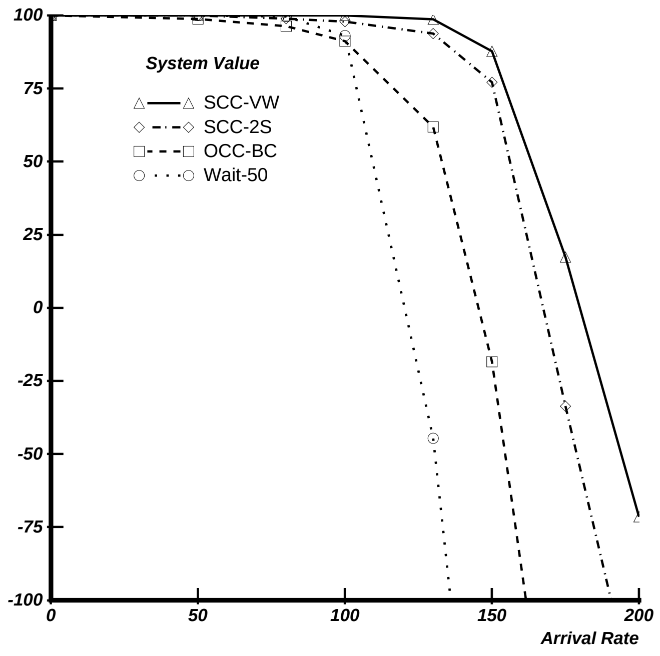

Figure 18: System Value for baseline model (2 classes)

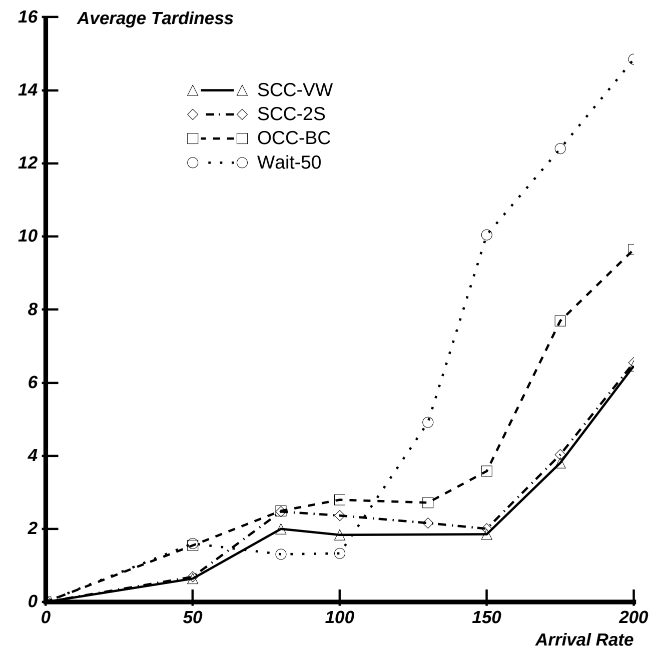

Figure 17: SCC-VW: Tardiness under baseline model

and smaller penalty gradients. The transaction mix was such that only 10% of the transactions in the system were from the first class. This transaction mix, along with the value functions chosen for the two classes were set so as to make the *average* value function identical to the value function when only one class was simulated (see figure 15). The results in figure 18 highlight the superiority of SCC-VW, which can be attributed to its novel incorporation of deadline and criticalness information in concurrency control decisions.

5 Conclusion

SCC protocols introduce a new dimension (namely redundancy) that can be used to improve the timeliness of transaction processing in RTDBS. In particular, by allowing a transaction to use extra resources, it can achieve better *speculation* and hence improve its chances for a timely commitment. In addition, SCC protocols offer a straightforward mechanism for rationing available redundancy amongst competing transactions based on transaction deadline and criticalness information. Thus, the problem of incorporating transaction deadline and criticalness information into concurrency control is reduced to the problem of rationing the available redundant resources amongst competing transactions. Those with higher payoff are allotted more resources so as to achieve better speculation, and hence better timeliness.

Acknowledgments

Benjamin Mandler and Sue Nagy developed the simulation testbed for our experiments. This work has been partially supported by NSF (grant # CCR-9308344) and by GTE.

References

[AAJ92] D. Agrawal, A. El Abbadi, and R. Jeffers. Using delayed commitment in locking protocols for real-time databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, Ca, 1992.

[ACL87] R. Agrawal, M. Carey, and M. Linvy. Concurency control performance modeling: Alternatives and implications. *ACM Transaction on Database Systems*, 12(4), December 1987.

[AGM88] 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.

[BB94] Azer Bestavros and Spyridon Braoudakis. Timeliness via speculation for real-time databases. In *Proceedings of RTSS'94: The 14^{th} IEEE Real-Time System Symposium*, San Juan, Puerto Rico, December 1994.

[BCFF87] 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.

[Bes92] Azer Bestavros. Speculative Concurrency Control: A position statement. Technical Report TR-92-016, Computer Science Department, Boston University, Boston, MA, July 1992.

[BMHD89] 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.

[Bra94] Spyridon Braoudakis. *Concurrency Control Protocols for Real-Time Databases*. PhD thesis, Computer Science Department, Boston University, Boston, MA 02215, expected June 1994.

[BSR88] Sara Biyabani, John Stankovic, and Krithi Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Prooceedings of the 9th Real-Time Systems Symposium*, December 1988.

[EGLT76] 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.

[GLPT76] 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.

[HCL90a] Jayant R. Haritsa, Michael J. Carey, and Miron Linvy. Dynamic real-time optimistic concurrency control. In *Prooceedings of the 11th Real-Time Systems Symposium*, December 1990.

[HCL90b] 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.

[HCL92] 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.

[HSRT91] 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.

[HSTR89] 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.

[JLT85] E. Jensen, C. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proceedings of the 6th Real-Time Systems Symbosium*, December 1985.

[Kor90] Henry Korth. Triggered real-time databases with consistency constraints. In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, 1990.

[KR81] H. Kung and John Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2), June 1981.

[KS91] Woosaeng Kim and Jaideep Srivastava. Enhancing real-time dbms performance with multiversion data and priority based disk scheduling. In *Prooceedings of the 12th Real-Time Systems Symposium*, December 1991.

[Loc86] C. Locke. *Best Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie-Mellon University, Department of Computer Science, May 1986.

[LS90] 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.

[MN82] D. Menasce and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1), 1982.

[Rob82] John Robinson. *Design of Concurrency Controls for Transaction Processing Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1982.

[Sin88] Mukesh Singhal. Issues and approaches to design real-time database systems. *ACM, SIGMOD Record*, 17(1):19–33, 1988.

[SPL92] S. Son, S. Park, and Y. Lin. An integrated real-time locking protocol. In *Prooceedings of the IEEE International Conference on Data Engineering*, Tempe, AZ, February 1992.

[SRL88] Lui Sha, R. Rajkumar, and J. Lehoczky. Concurrency control for distributed real-time databases. *ACM, SIGMOD Record*, 17(1):82–98, 1988.

[SRSC91] Lui Sha, R. Rajkumar, Sang Son, and Chun-Hyon Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, 1991.

[SZ88] John Stankovic and Wei Zhao. On real-time transactions. *ACM, SIGMOD Record*, 17(1):4–18, 1988.