

Value-cognizant Speculative Concurrency Control*

AZER BESTAVROS
(best@cs.bu.edu)

SPYRIDON BRAOUDAKIS
(sb@cs.bu.edu)

Computer Science Department
Boston University
Boston, MA 02215

February 20, 1995

Abstract

A problem with Speculative Concurrency Control algorithms and other common concurrency control schemes using forward validation is that committing a transaction as soon as it finishes validating, may result in a value loss to the system. Haritsa showed that by making a lower priority transaction wait after it is validated, the number of transactions meeting their deadlines is increased, which may result in a higher value-added to the system. SCC-based protocols can benefit from the introduction of such delays by giving optimistic shadows with high value-added to the system more time to execute and commit instead of being aborted in favor of other validating transactions, whose value-added to the system is lower. In this paper we present and evaluate an extension to SCC algorithms that allows for commit deferments.

*This work has been partially supported by NSF (grant CCR-9308344).

1 Introduction

Various concurrency control algorithms differ in the time when conflicts are detected, and in the way they are resolved. Pessimistic Concurrency Control (PCC) protocols [Eswa76, Gray76] detect conflicts as soon as they occur and resolve them using *blocking*. Optimistic Concurrency Control (OCC) protocols [Boks87, Kung81] detect conflicts at transaction commit time and resolve them using *rollbacks*.

For a conventional DataBase Management System (DBMS) with limited resources, performance studies of concurrency control methods (*e.g.* [Agra87]) have concluded that PCC locking protocols perform better than OCC techniques. The main reason for this good performance is that PCC's blocking-based conflict resolution policies result in resource conservation, whereas OCC's restart-based conflict resolution policies waste more resources. While abundant resources are usually not to be expected in conventional database systems, they are more common in real-time environments [Fran85], which are engineered to cope with rare high-load conditions, rather than normal average-load conditions. For example, Real-Time DataBase Systems (RTDBS) are engineered not to guarantee a particular throughput, but to ensure that in the rare event of a highly-loaded system, transactions (critical ones in particular) complete before their set deadlines [Buch89]. This often leads to a computing environment with far more resources than what would be necessary to sustain average loads. In such environments, the advantage that PCC blocking-based algorithms have over OCC restart-based algorithms vanishes. In particular, under such conditions, OCC algorithms become attractive since computing resources wasted due to restarts do not adversely affect performance. Haritsa *et al.* [Hari90b, Hari90a] investigated the behavior of both PCC and OCC schemes in a real-time environment. The study showed that for a RTDBS with firm deadlines (where late transactions are immediately discarded) OCC outperforms PCC, especially when resource contention is low. The key result of this study is that, if low resource utilization is acceptable (*i.e.* a large amount of wasted resources can be tolerated) then a restart-oriented algorithm that allows a higher degree of concurrent execution becomes a better choice.

Real-time concurrency control schemes considered in the literature could be viewed as extensions of either PCC-based or OCC-based protocols. In particular, transactions are assigned priorities that reflect the urgency of their timing constraints. These priorities are used in conjunction with PCC-based techniques [Abbo88, Agra87, Stan88, Huan89, Sing88, Sha88, Sha91] to make it possible for more urgent transactions to abort conflicting, less urgent ones (thus avoiding the hazards of blockages); and are used in conjunction with OCC-based techniques [Kort90, Hari90b, Hari90a, Huan91, Kim91, Lin90, Son92] to favor more urgent transactions when conflicting, less urgent ones attempt to validate and commit (thus avoiding the hazards of restarts).

In a recent study [Best92], we proposed a categorically different 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 [Best94] demonstrated the superiority of SCC-based protocols to OCC-based and PCC-based real-time concurrency control 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. In particular, SCC protocols 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 and we describe briefly SCC-OB, the most general SCC protocol. Next, the SCC-kS protocol, a practical speculative technique that operates under a limited speculation (resources) assumption, is presented. SCC-kS allows the system to constrain the level of speculation that each transaction is allowed to perform. This provides a straightforward mechanism for trading resources for timeliness. 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 our simulation results, which show that SCC-based algorithms provide significant performance gains over other widely used protocols.

2 Speculative Concurrency Control

A major disadvantage of the classical OCC [Kung81] 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 variant of the classical OCC (OCC-BC) [Mena82, Robi82] attempts to solve this problem by a notification process. When a transaction commits, it notifies all concurrently running, conflicting transactions about its commitment. All those conflicting transactions are immediately restarted. The broadcast commit variant detects conflicts earlier than the basic OCC algorithm resulting in less wasted resources and earlier restarts.

To illustrate 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 the basic OCC algorithm means restarting transaction T_2 when it enters its validation phase because it conflicts with the already committed transaction T_1 on data item x . This scenario is illustrated in figure 1(a). Obviously, the likelihood of the restarted transaction T_2 meeting its timing constraint decreases considerably. The OCC-BC algorithm avoids waiting unnecessarily for a transaction’s validation phase in order to restart it. A transaction is aborted immediately if any of its conflicts with other transactions in the system becomes a materialized consistency threat. This is illustrated in figure 1(b).

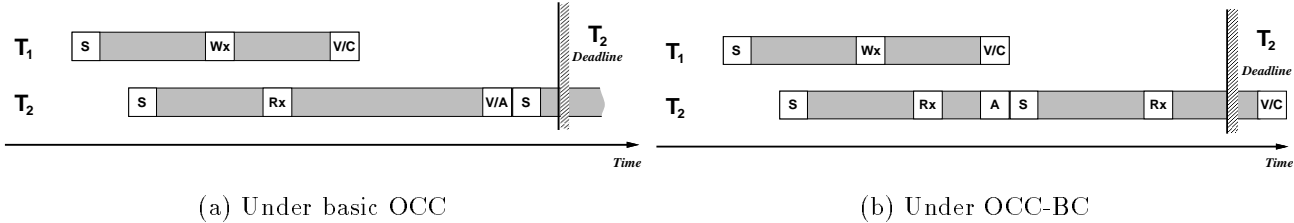


Figure 1: Transaction management illustration.

The SCC approach proposed in [Best92] 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 corrective measures as early as possible, the likelihood of meeting set timing constraints is greatly enhanced. Figure 2(a) and figure 2(b) show two possible scenarios that may develop depending on the time needed for transaction T_2 to reach its validation phase. In figure 2(a), T_2 reaches its validation phase before T_1 . T_2 will be validated and committed without any need to disturb T_1 . This schedule will be serializable with transaction T_2 preceding transaction T_1 . Obviously, once T_2 commits, the shadow transaction T_2' has to be aborted. However, if transaction T_1 reaches its validation phase first, then transaction T_2 cannot continue to execute due to the conflict over x ; T_2 must abort. With OCC-BC algorithms, T_2 would have had to restart when T_1 commits. This might be too late if T_2 's deadline is close. The SCC protocol (see figure 2(b)), instead of restarting T_2 , simply aborts T_2 and adopt its shadow transaction T_2' .

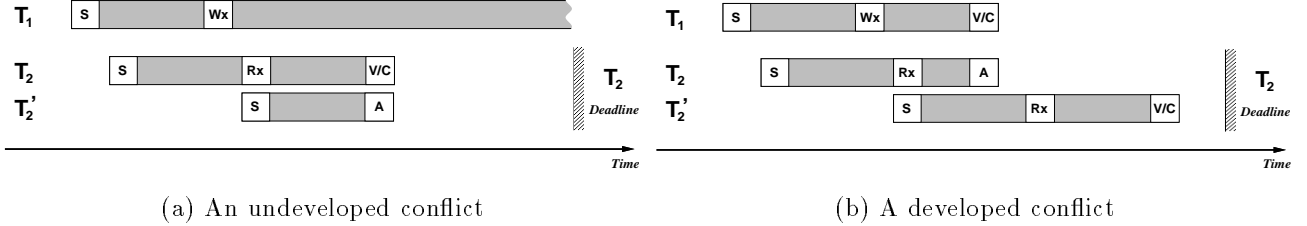


Figure 2: Transaction management under SCC.

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 may require a large amount of redundancy. In particular, 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 [Brao94], namely: $\sum_{i=1}^n \frac{(n-1)!}{(n-i)!} = \mathcal{O}((n-1)!)$.

As an illustration of the relationship between shadows and SOS, consider figure 3, which shows a schedule for three, pairwise conflicting transactions T_1 , T_2 , and T_3 . The SCC-OB algorithm requires, among other things, the maintenance of five shadows on behalf of T_3 , each with different SOS. For instance, T_3^0 (the optimistic shadow) speculates that T_3 will commit before both T_1 and T_2 . T_3^1 speculates that T_1 will commit before T_3 , which in turn will commit before T_2 . T_3^2 speculates that T_3 will commit before T_1 , but after T_2 . T_3^3 speculates that T_1 will commit first, followed by T_2 , followed by T_3 . Finally, T_3^4 speculates that T_2 will commit first, followed by T_1 , followed by T_3 . Notice that figure 3 illustrates the shadows maintained on behalf of T_3 only. Similar shadows (not shown) will be maintained for T_1 and T_2 to account for their conflicts with other transactions in the system (including T_3).

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 serialization orders is *on-to-many*), then it can be shown that only a linear number of shadows is sufficient to yield all the power of the basic SCC algorithm. Such an optimized algorithm, called Conflict-Based SCC (SCC-CB) is detailed in [Brao94]. In the illustration of figure 3, SCC-CB would require the maintenance of only *three* shadows, namely: T_3^0 (covering all SOS in which T_3 commits first amongst all conflicting transactions), T_3^1 (covering all SOS in which T_3 commits after T_2), and T_3^2 (covering all SOS in which T_3 commits after T_1). It can be shown that at any point in time, SCC-CB needs no more than n shadows per transaction. Moreover, 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.

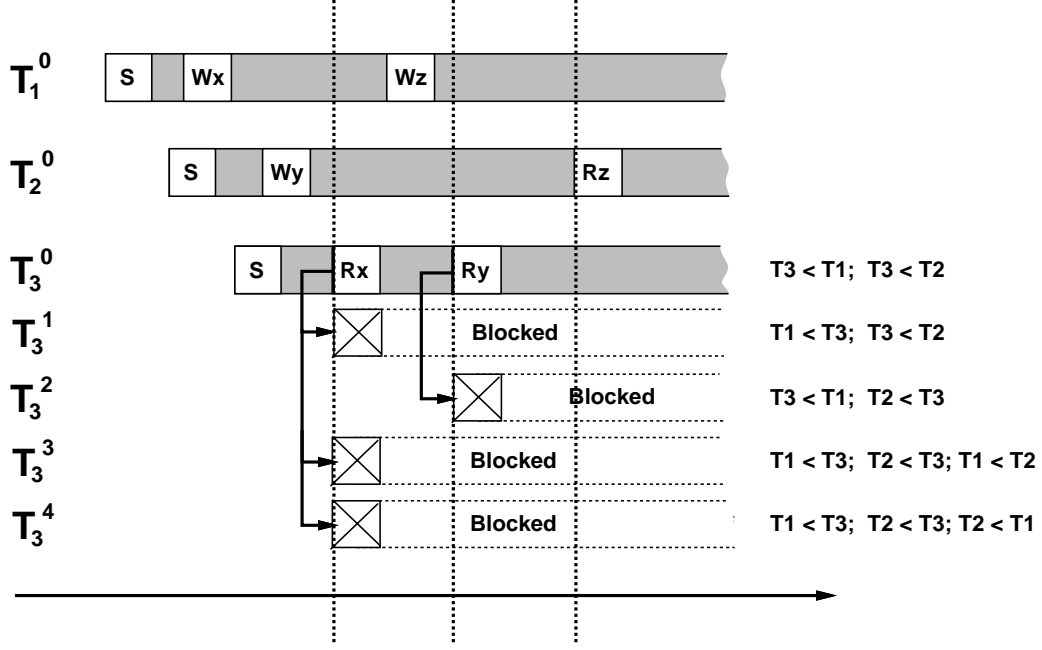


Figure 3: SOS coverage for a schedule with three conflicting transactions.

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

In this section, we describe briefly¹ a class of SCC algorithms that operate under a *limited resources* assumption. In particular, we present a generic SCC algorithm which does not allow 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 $i = 0, \dots, k - 1$.

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

Speculative shadow behavior: Each speculative shadow T_r^s executes with the assumption that it will finish before the materialization of any detected conflict with any other uncommitted transaction, except for one particular conflict which is *speculated* to materialize before the commitment of T_r . Thus, T_r^s remains blocked on 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 made the wrong assumption with respect to the serialization order.

¹A thorough description of SCC-kS and proof of its correctness are detailed in [Brao94].

In our protocol, k (the upper limit on the number of shadows allowed per transaction) does not have to be the same for all transactions. The value of k for a particular transaction 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 reflect the transaction’s urgency (how tight is the deadline) and criticalness. The value of k does not have to be constant for a given transaction; it 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 for all transactions.

Let $\mathcal{T} = T_1, T_2, T_3, \dots, T_m$ 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 order² 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 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 . 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 as a set of five rules, which we describe below.

Start Rule:

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

Read Rule:

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

²This can be a special read timestamp, implemented by maintaining for each shadow T_r^i in the system a counter that is atomically incremented every time a read operation is performed by T_r^i .

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 , even though in a later stage, is not an appropriate shadow to fork off because, like the optimistic shadow, it already read X .

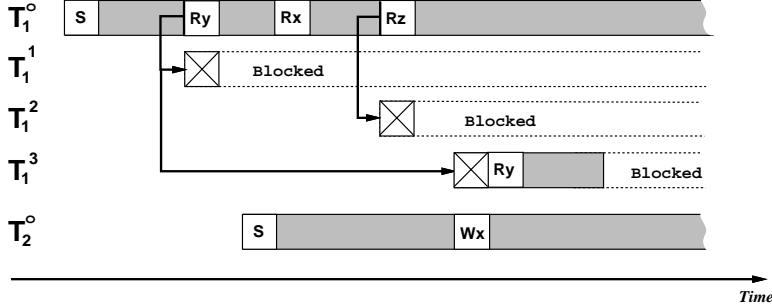


Figure 4: T_1^3 is forked off T_1^1 .

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. All other speculative shadows of T_1 remain unaffected.

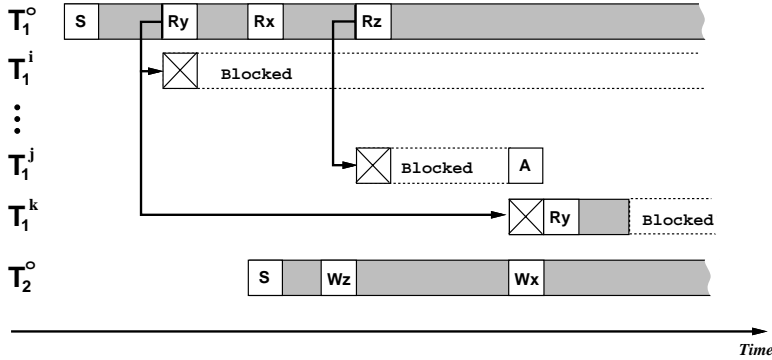


Figure 5: Example of multiply conflicting transactions.

The imposed limit of at most $k - 1$ speculative shadows per transaction does not prohibit a transaction T_i from developing more than $k - 1$ conflicts at any point during its lifetime. Rather, this limit is on the number of potential hazards that our algorithm will be ready to *optimally* deal with (by using the speculative shadows). Every *extra* hazard that develops after this limit is reached will be accounted for only *suboptimally*³ (since no such speculative shadow will be available). The selection of the conflicts to be accounted for by speculative shadows is an interesting problem with many possible solutions. In [Best94] we have adopted a particular solution 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. The *Latest-Blocked-First-Out* (LBFO) *shadow replacement*

³We can still use the presence of other speculative shadows to improve those decisions.

policy implements this by replacing the shadow with the latest blocking point. LBFO is one of several policies that could be adopted. In [Brao94] some alternatives to this policy are discussed and evaluated. In particular, information about deadlines and priorities of the conflicting transactions can be utilized so as to account for the *most probable* serialization orders.

To illustrate this point, consider the scenario depicted in figure 6, where the assumption that the first two conflicts in which transaction T_1 participated (by accessing objects Y , and Z , respectively), is revised when transaction T_2 writes object X . In particular, the newly detected conflict (T_2, X) becomes the first conflict of T_1 . If it is the case that T_1 is restricted so as not to have more than two speculative shadows at any point during its execution, then a shadow replacement is necessary. T_1^2 , the *latest* shadow of T_1 has to be aborted, and a new speculative shadow, T_1^3 , accounting for the new (T_2, X) conflict should replace it.

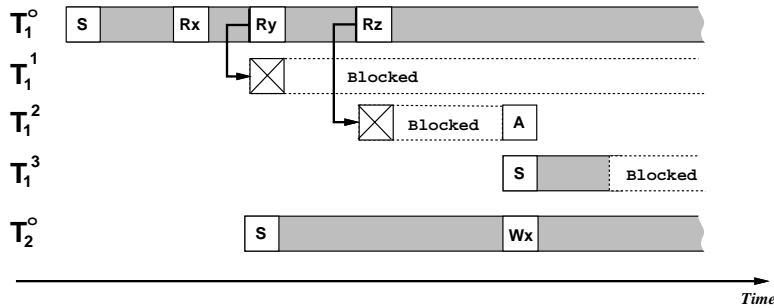


Figure 6: Example of LBFO shadow replacement.

Blocking Rule:

The *Blocking Rule* is used to control when a speculative shadow T_r^i must be blocked. This rule assures that T_r^i is blocked the *first* time it wishes to read an object X in conflict with any transaction that T_r^i must wait for according to its 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 , waiting for T_r 's commitment, or not. The first case is illustrated in figure 7, where the speculative shadow T_1^2 of transaction T_1 – having anticipated (assumed) the correct serialization order – is promoted to become the new optimistic shadow of transaction T_1 , replacing the old optimistic shadow which had to be aborted. Speculative shadow T_1^3 — which like the old optimistic shadow exposed itself by reading the old value of object X — had to be aborted as well. On the contrary, the speculative shadow T_1^1 , which did not read object X , remains unhindered. The second case is illustrated in figure 8, where the commitment of the optimistic shadow T_2^o on behalf of transaction T_2 was not accounted for by any speculative shadow.⁴ 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

⁴Figure 8 makes the implicit assumption that transaction T_1 is limited to having at most two speculative shadows at any point during its execution.

of transaction T_1 . This, even though not optimal, is the best we can do in the absence of a speculative shadow accounting for the (T_2, Z) conflict.

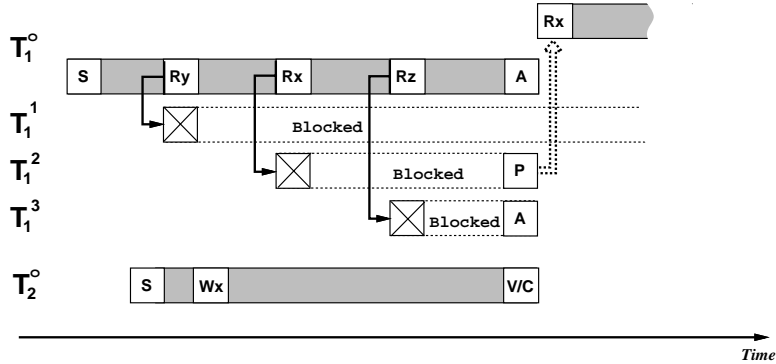


Figure 7: Applying the Commit Rule (case 1).

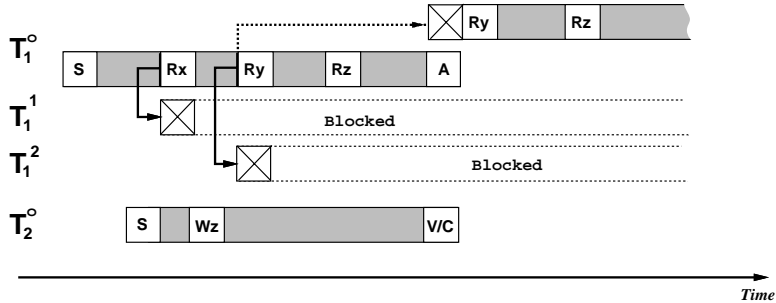


Figure 8: Applying the Commit Rule (case 2).

2.2 Two-Shadow SCC (SCC-2S)

SCC-2S allows a maximum of two shadows per uncommitted transaction to exist in the system at any point in time: an *optimistic* shadow and a *pessimistic* shadow. 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, should such a replacement be 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.

3 Incorporation of Deadline and Criticalness Information

In the previous section, we have discussed one way of incorporating deadline and criticalness information into the SCC methodology—namely by relating the relative *worth* of transactions to the amount of speculation (and thus resources) they are allotted. Nevertheless, these algorithms are not value-cognizant algorithms because they do not make any use of deadline or criticalness information in resolving data conflicts or in making other scheduling decisions. While this protects them from problems related to priority dynamics (*e.g.* priority inversion and starvation [Sha90]), it also prevents them from making better scheduling decisions. In this section, we discuss one way of incorporating deadline and criticalness information within the SCC methodology.

Most of the previous studies on the performance of real-time concurrency control algorithms considered RTDBS that operate under the assumption that all transactions in the system are of equal worth. Their major performance objectives were to minimize the number of missed firm deadlines or 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 [Stan88, Huan89] to reflect their relative worth to the system upon their successful completion. For such systems the attention shifts to maximizing the *value-added* to the system by the transactions’ commitment. Minimizing tardiness and the number of missed deadlines become of secondary importance. Notice that a transaction’s value and its deadline are two orthogonal properties [Biya88, Huan89]. 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.

Performance analysis studies that incorporated transaction values include those of Huang *et al.*[Huan89, Huan91] and Haritsa *et al.*[Hari91]. In [Huan89] several PCC-based algorithms are investigated for resource scheduling and data conflict resolution. This work is extended in [Huan91] to include OCC-based methods. Both consider a soft RTDBS where transaction values, after missing their deadlines, decrease at a rate inversely proportional to the values that these transactions had before their deadlines. In [Hari91] the special case in which all transactions have step-shaped value functions—the system is operating under a firm deadlines assumption—is investigated. Our work differs from the work of Huang *et al.* in that we do not relate the value that a transaction has before its deadline to the rate with which this value decreases after its deadline. We investigate value-based variants of optimistic methods which are extended to include SCC-based algorithms. Our work differs from Haritsa *et al.*’s work in that we consider soft deadline RTDBS where transactions may sustain some (diminished) value when they are completed after their deadlines. Another difference is that in our work the rates at which transactions are penalized for missing their deadlines are an integral part of the transaction’s value functions.

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 the *value function* introduced in the work of Jensen, Locke, and Tokuda [Jens85, Lock86]. Each system transaction T_u is associated with a value function $V_u(t)$, which represents the value of transaction 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 soft RTDBS 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 9 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 $\tan \alpha_u$.

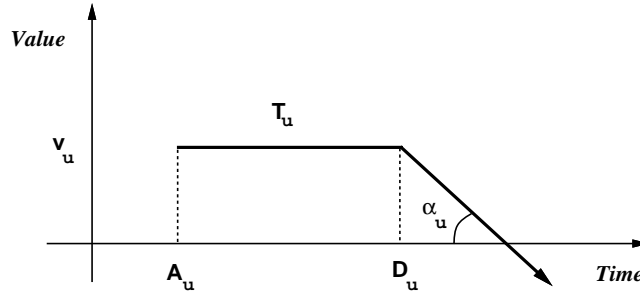


Figure 9: A typical value function for the soft RTDBS transaction T_u .

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

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

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 Value-Based SCC with Deferred Commit (SCC-DC)

A problem with SCC algorithms and other common concurrency control schemes is that committing a transaction as soon as it finishes validating, may result in a value loss to the system. For example, in figure 10(a), 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 [Hari90b], Haritsa showed that by making a lower priority⁵ transaction wait after it is validated, the number of transactions meeting their deadlines is increased, which results in a higher value-added to the system. SCC-based protocols can benefit from the introduction of such delays by giving optimistic shadows with high value-added to the system more time to execute and commit instead of being aborted in favor of other validating transactions, whose value-added to the system is lower. Figure 10(b) shows the increased value-added to the system that results from delaying the commitment of the validated transaction T_1 , thus allowing T_2^o to commit on behalf of transaction T_2 on time to meet its deadline and contribute a higher value to the system.

In the remainder of this section we present one way of incorporating deferred transaction commitment in SCC-based protocols to exploit any potential additional concurrency in the system. Our approach is similar

⁵The notion of transaction value as an integral part of a transaction's priority was not used in that study.

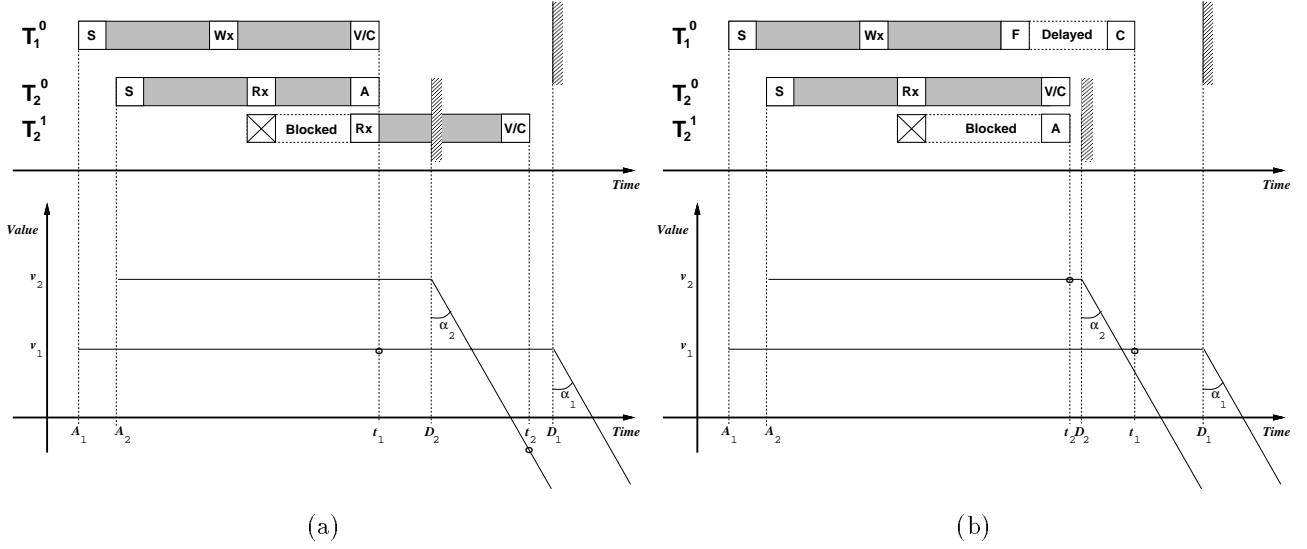


Figure 10: Value-added to the system (a) without deferment (b) with deferment.

to those proposed in [Agra92, Hari90a, Son92]. Whenever a transaction shadow T_u^o (executing on behalf of transaction T_u) finishes its execution, we evaluate if it is advantageous to defer T_u^o 's commitment for a later point in time. Finding the *best* point in time to commit a finished shadow T_u^o is a very hard optimization problem, since it requires to consider all possible serialization orders of active transactions in the system. To avoid the exponential nature of this problem and make this evaluation tractable, we propose an SCC-based protocol, SCC with Deferred Commit (SCC-DC), which estimates the value-added to the system at *discrete* points in time (*e.g.* periodically). In particular, SCC-DC compares the estimated value-added to the system if the finished shadow T_u^o is committed at time t , to the estimated value-added to the system if T_u^o is committed at time $t + \delta$, where δ is some constant delay. Notice that, because of its discrete nature, this simplified algorithm does not always provide us with the *best* point in time to commit a shadow T_u^o on behalf of a transaction T_u . This *optimal* point in time may very 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 in different classes according to their run-time characteristics. We denote with C_u the *class* of transaction T_u . We make the assumption that for each transaction class C_u , we can estimate the *average execution time*, E_{C_u} , of the transactions of this class. This can be obtained off-line from the previous history of the system, or at run-time from collected statistical results. With each transaction class C_u we associate a *probability density function* $F_u(x)$. A typical finish probability density function is depicted in figure 11.

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 exceed x ,*

$$F_u(x) = \text{Prob}[T_u \in C_u \text{ will finish after more than } x \text{ time units}].$$

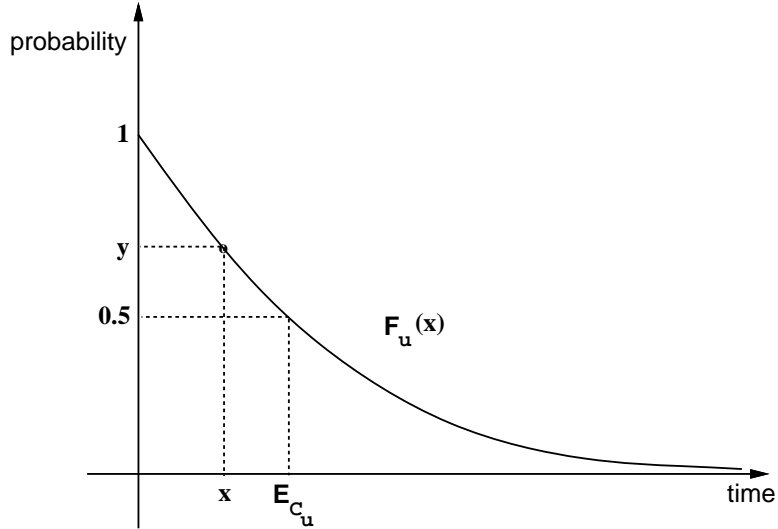


Figure 11: Finish probability density function of transaction class C_u .

3.2.1 Deciding on the Commitment of Transactions

As we mentioned above, we estimate the value-added to the system at discrete, periodic points in time. We assume that a special system clock exists that ticks with a period δ , signaling the points in time, when system transactions may be committed. At each tick of this clock 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. In particular, if the clock ticks at time t and T_u^o is a transaction shadow which has finished its execution, then according to the SCC-DC protocol:

- 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, \dots, T_m , then we compute:
 - a. the expected value-added to the system, V_{now} , should we commit T_u^o at the current clock tick t , and
 - b. the expected value-added to the system, V_{later} , should we defer T_u^o 's commitment to a later tick at time $t + k\delta$, for $k \in \mathcal{N}^*$.

If $V_{now} \geq V_{later}$ then we commit T_u^o on behalf of T_u , otherwise we defer T_u^o 's commitment.

In SCC-based protocols more than one shadow may coexist on behalf of a particular transaction. The computation, thus, of the expected value-added to the system on behalf of a transaction T_u depends on *which* shadow of T_u is committing and at *what* time. In the following we define two measures, the shadow *finish probability* and the shadow *adoption probability*, which we use to assist in these computations.

Shadow Finish Probability:

With each shadow T_u^i of a transaction T_u we associate a probability function $F_u^i(x)$.

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 execution⁶ 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, its finish probability can be computed at any point in time t_{now} using the probability density function $F_u(x)$ —the probability that transactions of class C_u will be able to *finish* their execution after time x —as follows:

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

where we made use of the formula:

$$\text{Prob}[A/B] = \frac{\text{Prob}[A \text{ and } B]}{\text{Prob}[B]}$$

Notice that τ is the same as the time elapsed from T_u 's arrival time A_u to the time at which T_u^i was blocked if it is a speculative shadow of T_u , or $(t_{now} - A_u)$ if T_u^i is the optimistic shadow of transaction T_u .

Shadow Adoption Probability:

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 which is created to account for a conflict with a higher valued transaction is more likely to be adopted in the future (more so than a shadow which is created to account for a conflict with a lesser valued transaction). To express the likelihood of a shadow being adopted, we associate with each shadow T_u^i of a transaction T_u , a probability function $P_u^i(t)$.

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 indicate the relative importance of the shadows of a transaction as a function of time. For a transaction T_u they are computed as follows. At time t :

- a. If T_u has no speculative shadows (i.e. conflicts with no other uncommitted transaction), then $P_u^o(t) = 1$.
- b. If T_u conflicts with transactions $T_{r_1}, T_{r_2}, \dots, T_{r_m}$ then:

$$\begin{aligned} P_u^o(t) &= \frac{V_u(t)}{V_u(t) + V_{r_1}(t)P_{r_1}^o(t) + V_{r_2}(t)P_{r_2}^o(t) + \dots + V_{r_m}(t)P_{r_m}^o(t)} \\ P_u^i(t) &= \frac{V_i(t)P_i^o(t)}{V_u(t) + V_{r_1}(t)P_{r_1}^o(t) + V_{r_2}(t)P_{r_2}^o(t) + \dots + V_{r_m}(t)P_{r_m}^o(t)}, \quad \text{for } i = r_1, \dots, r_m. \end{aligned}$$

where we assume that T_u^i is the shadow of transaction T_u that accounts for the conflict of T_u with transaction T_i for $i = r_1, \dots, r_m$.

⁶If T_u^i is a speculative shadow then we assume that it resumes its execution immediately.

Description of the SCC-DC Algorithm

To incorporate the previously discussed deferred commitment technique, we add to the SCC-kS protocol an additional rule which controls the commitment of transactions. We name this new rule the *Termination Rule*, because it deals with transaction shadows that have terminated (finished) their executions and are, thus, ready to be committed. 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 in the system, then T_u^o is committed on behalf of transaction T_u . Otherwise, if T_u conflicts with transactions T_1, T_2, \dots, T_m , then the expected value-added to the system, V_{now} (corresponding to the value-added should T_u be committed now), is compared to the expected value-added to the system, V_{later} (corresponding to the value-added 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.

To compute V_{now} and V_{later} , we utilize two functions: the *Expected Finish probability* and the *Expected Value-added* to the system, defined below.

Definition 6 *The Expected Finish probability, $EF_u(x)$, 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 . $EF_u(x)$ is computed as the summation below over all j shadows of T_u .*

$$EF_u(x) = \sum_j F_u^j(x) P_u^j(t),$$

where $F_u^j(x)$ and $P_u^j(t)$ are the shadow finish and the shadow adoption probability functions of shadows of transaction T_u , respectively.

Definition 7 *We denote by $EV_u(x)$ the Expected Value-added to the system if the transaction T_u completes its execution (commits) at time x ,*

$$EV_u(x) = V_u(x) EF_u(x).$$

Using these functions, V_{now} is computed as the value-added to the system from the commitment of shadow T_u^o on behalf of transaction T_u at time t , $V_u(t)$, plus the expected value added to the system from the commitment of transactions T_1, T_2, \dots , and T_m at a later time $t + k\delta$, for $k = 1$ to infinity,

$$V_{now} = V_u(t) + \sum_{i=1}^m \sum_{k=1}^{\infty} EV_i(t + k\delta)$$

Similarly, V_{later} is computed as the expected value added to the system from the commitment of transactions T_u, T_1, T_2, \dots , and T_m at some later time $t + k\delta$, for $k = 1$ to infinity,

$$V_{later} = \sum_{k=1}^{\infty} EV_u(t + k\delta) + \sum_{i=1}^m \sum_{k=1}^{\infty} EV_i(t + k\delta)$$

The infinite summations in the above calculations can be bounded, introducing a limited amount of error. To accomplish this we observe that for each transaction T_i there exist a time $t + k\delta$, for $k = l_i$, such that the expected finish probability of transaction T_i , $EF_i(l_i\delta) = 1 - \epsilon$, where ϵ can be an arbitrarily small number. We, therefore, bound these summations with appropriate $k = l_i$ values, introducing arbitrarily small errors ϵ . We are now ready to introduce the *Termination Rule* of our SCC-DC protocol.

Termination Rule:

For each shadow T_u^o that finished executing, evaluate if it is advantageous to defer its commitment.

- ◊ If T_u does not conflict with other uncommitted transactions, then invoke the *Commit Rule* to commit T_u^o .
- ◊ Otherwise, if T_u conflicts with transactions T_1, T_2, \dots, T_m , then:
 - a. Compute the expected value-added to the system, V_{now} , should T_u^o be committed at the current time t ,

$$V_{now} = V_u(t) + \sum_{i=1}^m \sum_{k=1}^{l_i} EV_i(t + k\delta)$$

- b. compute the expected value-added to the system, V_{later} , should T_u^o 's commitment be deferred,

$$V_{later} = \sum_{k=1}^{l_u} EV_u(t + k\delta) + \sum_{i=1}^m \sum_{k=1}^{l_i} EV_i(t + k\delta)$$

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

In addition to the *Termination Rule*, other modifications to the rules of the SCC-kS algorithm are necessary. The first modification affects the *Commit Rule*. Under SCC-DC, transactions *do not* commit as soon as they finish their execution. Rather, they wait (at least) until the next periodical invocation of the *Termination Rule*, before they can commit. Thus, the *Commit Rule* can be invoked *only when* the *Termination Rule* decides to commit a shadow. The second modification affects the *Read* and *Write Rules*. In previously presented SCC-based algorithms a transaction, whose optimistic shadow has finished executing is *always* committed. This follows from our *forward validation* approach. Under SCC-DC, a transaction's 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 values-added to the system), that T_u^o be committed, thus resulting in the abortion of the finished T_r^o shadow. To protect against such situations, the *Read Rule* is extended, so as to be invoked whenever an optimistic shadow T_r^o wishes to read an object X , which is written by another shadow T_u^o , whether T_u^o is currently executing or has already finished its execution and is awaiting commitment. Similarly, the *Write Rule* is extended to allow the creation of a shadow for a transaction, whose optimistic shadow has finished executing. The other two rules of the SCC-OB (or SCC-kS) algorithm, the *Start Rule* and the *Blocking Rule*, remain unaffected.

3.3 SCC with Voted Waiting (SCC-VW)

SCC-DC requires a substantial amount of computations to determine whether or not to defer a transaction's commitment. In this section, we propose an approximation heuristic, Voted Waiting (VW), to reduce the computational complexity of SCC-DC. 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 $CI_u > 50\%$ then T_u^o is committed, otherwise it waits.

Basic Definitions

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:

$$cv_u^i = \begin{cases} 1 & \text{if } T_i \text{ votes to commit } T_u^o \\ 0 & \text{if } T_i \text{ votes not to commit } T_u^o \end{cases}$$

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 we denote by \mathcal{T}^u the set of executing transactions that conflict with the finished shadow T_u^o , and by $V_k(t)$ the value function of transaction T_k .

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

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

Description of the SCC-VW Algorithm

A transaction T_i votes to commit a finished conflicting shadow T_u^o , if by committing T_u^o a higher expected value-added to the system is produced, compared with the one produced by delaying T_u^o 's commitment in favor of T_i 's own optimistic shadow. Otherwise, T_i votes *not* to commit T_u^o . The expected value-added to the system, V_{now} , if T_i votes to commit T_u^o at the current time t is given by the addition of the value-added to the system from the commitment of T_u^o at time t , $V_u(t)$, plus the value-added to the system from the *expected* commitment of the T_i^u shadow of T_i at time $t + (E_{C_i} - \tau_i^u)$. Let T_i^u be the shadow of T_i that accounts for the conflict with transaction 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, we get

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

For the computation of the expected value-added to the system, V_{later} , we distinguish between two cases:

- a. T_u has no speculative shadow accounting for a conflict with T_i (e.g. T_u has not read anything that is written by 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 and that $later = t + E_{C_i} - \tau_i^o$, we get

$$V_{later} = V_i(later) + V_u(later),$$

- b. There exist a speculative shadow T_u^i of T_u which accounts for a conflict with transaction T_i on some database object X . In this case, the commitment of T_i^o at time $t + E_{C_i} - \tau_i^o$ will result in the abortion of the (finished) optimistic shadow T_u^o of T_u , and its replacement by the speculative shadow T_u^i . Assuming that by that time T_u^i will have executed for τ_u^i time units, we get that

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

Termination Rule:

Whenever an optimistic shadow T_u^o , executing on behalf of a transaction T_u , finishes its execution, evaluate whether it is advantageous, to delay T_u^o 's commitment.

- ◇ If T_u does not conflict with other uncommitted transactions, then invoke the *Commit Rule* to commit T_u^o .
- ◇ Otherwise, if T_u^o conflicts with the set of transactions \mathcal{T}^u , then

1. For every transaction $T_i \in \mathcal{T}^u$ compute:

- The expected value-added, V_{now} , should T_i vote to commit T_u^o at the current time t :

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

- The expected value-added, V_{later} , should T_i vote not to commit T_u^o at the current time t .

a. if T_u has no speculative shadow accounting for a conflict with T_i , then

$$V_{later} = V_i(later) + V_u(later)$$

b. otherwise, if there exists a speculative shadow T_u^i on behalf of T_u , accounting for a conflict with transaction T_i on some database object X , then

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

- Determine the *commit vote*, cv_u^i , of T_i as follows:

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

2. Compute the *commit indicator*, CI_u , for the transaction shadow T_u^o ,

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

3. If $CI_u(t) \leq 50\%$, then delay T_u^o 's commitment, otherwise invoke the *Commit Rule* and commit T_u^o .

In the above description, we assumed that the shadows T_i^u , T_i^o , and T_u^i have already executed for τ_i^u , τ_i^o and τ_u^i time units, respectively. Also, we assumed that E_{C_i} and E_{C_u} are the average execution times of transactions T_i and T_u , respectively. Also, we assumed that, at time t , the shadows T_i^u , T_i^o , and T_u^i have already executed for τ_i^u , τ_i^o and τ_u^i time units, respectively.

4 Performance Evaluation

In this section, we present a comparative evaluation of the performance of locking, optimistic, and speculative concurrency control protocols. In particular, we evaluate the performance of the following protocols: 2PL with Priority Abort (2PL-PA) [Abbo88] as a representative of PCC-based protocols, OCC-BC [Hari90b] and WAIT-50 [Hari90a] 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 infinite resources. We consider that the time spent on

performing concurrency control tasks is negligible because under infinite resources we can assume that dedicated processors are assigned for these tasks. The system model consists of five main modules as depicted 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 to a *Transaction Sink*.

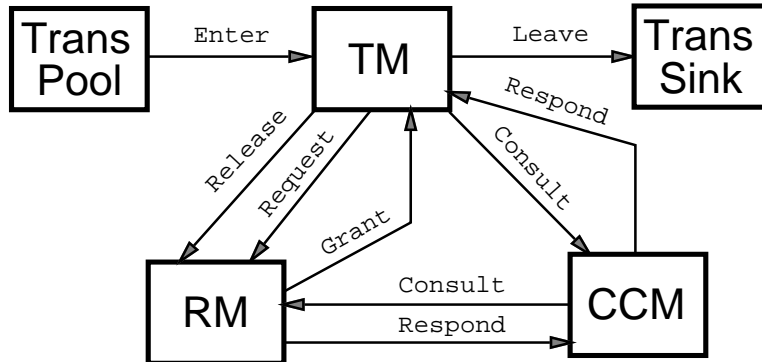


Figure 12: The Logical System Model

The primary performance measures that we employ here 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 within its deadline has a tardiness of zero. A transaction that completes after its deadline has a tardiness of $T - \text{Deadline}$, 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

In this section, we present a brief summary of the results of our simulations of the various concurrency control protocols. The 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 [Brao94].

The database consists of 1,000 pages 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 [Hari92]. 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 4000 transactions had completed their operations (committed or aborted). Enough runs to guarantee a 90% confidence interval were performed. Unless otherwise stated, our figures depict the average over all experiments.

Figures 13-a and 13-b 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.

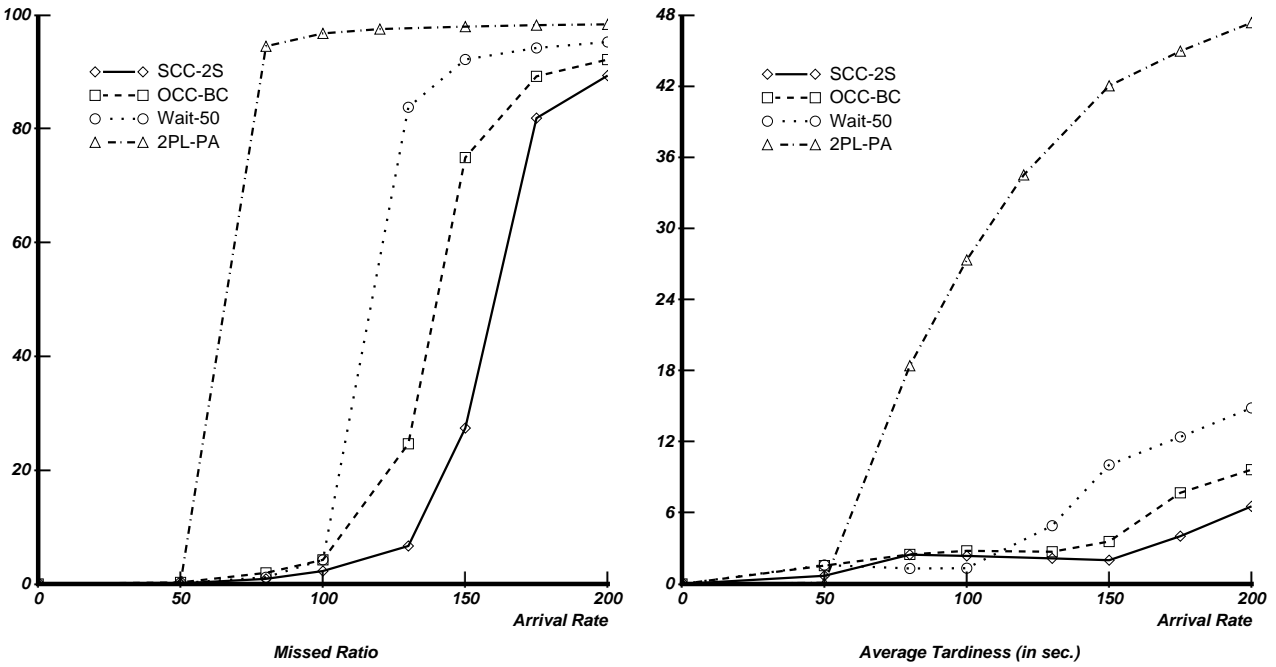


Figure 13: Baseline Model (a) Missed Ratio (b) Average Tardiness

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 OCC-BC with respect to *Average Tardiness* under all system loads (as figure 13-b 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.

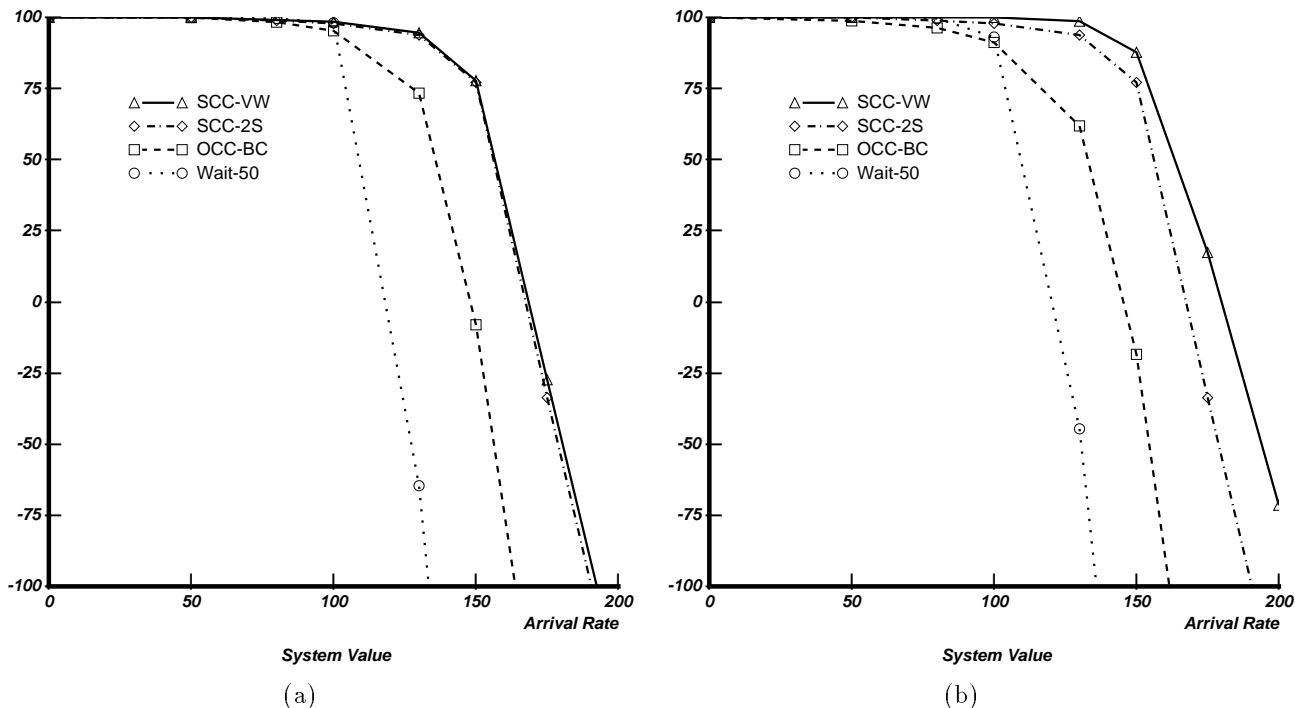


Figure 14: System Value (baseline model): (a) one class (b) two classes

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 14(a) depicts the *System Value* for the protocols in question, where all transactions are assigned the same *value function*.⁷ 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 15-a 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 15-b. 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.

⁷There is constant value if a transaction completes before its deadline, which declines (with a 45 degree slope) after it misses its deadline. All other parameters are set to that of the baseline model.

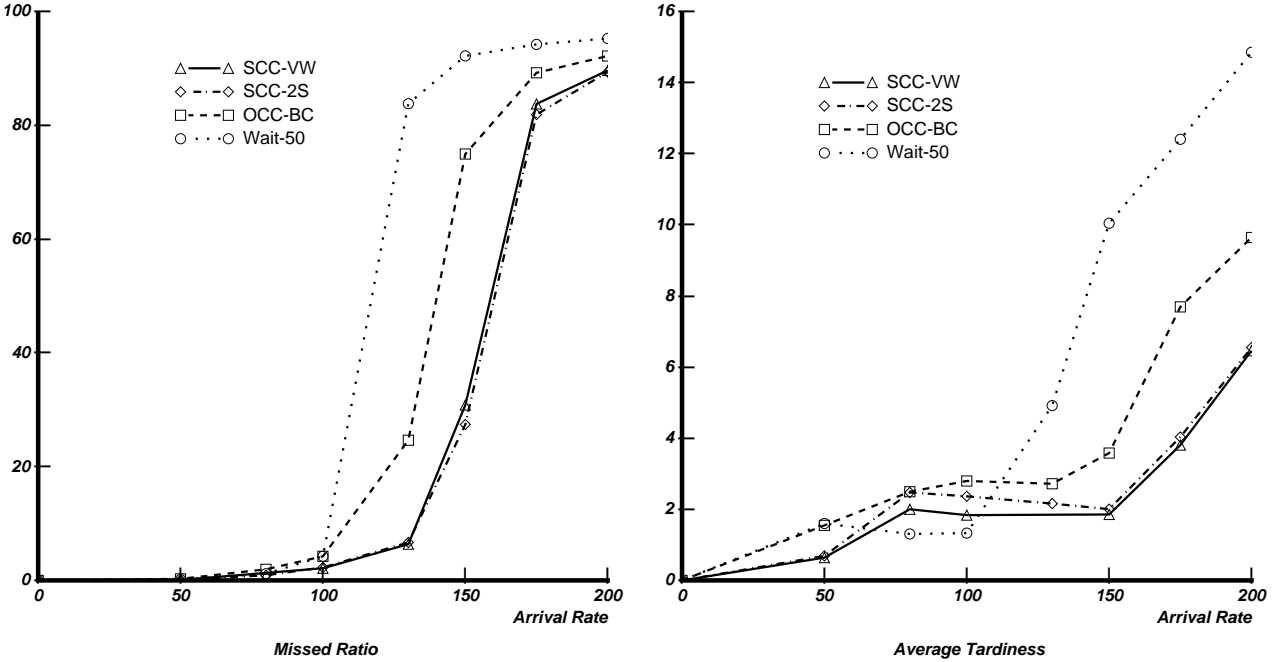


Figure 15: SCC-VW (a) Missed Ratio (b) Average Tardiness

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 14(b) 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, 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 14(a)). The results in figure 14(b) 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 more (redundant) 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.

References

- [Abbo88] Robert Abbott and Hector Garcia-Molina. "Scheduling real-time transactions: A performance evaluation." In *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, Ca, 1988.
- [Agra87] R. Agrawal, M. Carey, and M. Linvy. "Concurrency control performance modeling: Alternatives and implications." *ACM Transaction on Database Systems*, 12(4), December 1987.
- [Agra92] 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.
- [Best92] Azer Bestavros. "Speculative Concurrency Control: A position statement." Technical Report TR-92-016, Computer Science Department, Boston University, Boston, MA, July 1992.
- [Best94] Azer Bestavros and Spyridon Braoudakis. "Timeliness via speculation for real-time databases." In *Proceedings of RTSS'94: The 14th IEEE Real-Time System Symposium*, San Juan, Puerto Rico, December 1994.
- [Biy88] Sara Biyabani, John Stankovic, and Krithi Ramamritham. "The integration of deadline and criticalness in hard real-time scheduling." In *Proceedings of the 9th Real-Time Systems Symposium*, December 1988.
- [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.
- [Fran85] Peter Franaszek and John Robinson. "Limitations of concurrency in transaction processing." *ACM Transactions on Database Systems*, 10(1), March 1985.
- [Gray76] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. "Granularity of locks and degrees of consistency 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.
- [Hari90a] Jayant R. Haritsa, Michael J. Carey, and Miron Linvy. "Dynamic real-time optimistic concurrency control." In *Proceedings of the 11th Real-Time Systems Symposium*, December 1990.
- [Hari90b] Jayant R. Haritsa, Michael J. Carey, and Miron Linvy. "On being optimistic about real-time constraints." In *Proceedings of the 1990 ACM PODS Symposium*, April 1990.
- [Hari91] Jayant R. Haritsa, Michael J. Carey, and Miron Linvy. "Value-based scheduling in real-time database systems." Technical Report Computer Sciences Technical Report #1024, University of Wisconsin-Madison, May 1991.
- [Hari92] Jayant R. Haritsa, Michael J. Carey, and Miron Linvy. "Data access scheduling 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 *Proceedings of the 17th International Conference on Very Large Data Bases*, Barcelona, Spain, September 1991.
- [Jens85] 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 Symposium*, December 1985.
- [Kim91] Woosaeng Kim and Jaideep Srivastava. “Enhancing real-time dbms performance with multiversion data and priority based disk scheduling.” In *Proceedings of the 12th Real-Time Systems Symposium*, December 1991.
- [Kort90] Henry Korth. “Triggered real-time databases with consistency constraints.” In *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, 1990.
- [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.
- [Lock86] C. Locke. *Best Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie-Mellon University, Department of Computer Science, May 1986.
- [Mena82] D. Menasce and T. Nakanishi. “Optimistic versus pessimistic concurrency control mechanisms in database management systems.” *Information Systems*, 7(1), 1982.
- [Robi82] John Robinson. *Design of Concurrency Controls for Transaction Processing Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1982.
- [Sha88] Lui Sha, R. Rajkumar, and J. Lehoczky. “Concurrency control for distributed real-time databases.” *ACM, SIGMOD Record*, 17(1):82–98, 1988.
- [Sha90] L. Sha, R. Rajkumar, and J. Lojoczky. “Priority inheritance protocols: An approach to real-time synchronization.” *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.
- [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] S. Son, S. Park, and Y. Lin. “An integrated real-time locking protocol.” In *Proceedings of the IEEE International Conference on Data Engineering*, Tempe, AZ, February 1992.
- [Stan88] John Stankovic and Wei Zhao. “On real-time transactions.” *ACM, SIGMOD Record*, 17(1):4–18, 1988.