MIRROR: A State-Conscious Concurrency Control Protocol for Replicated Real-Time Databases

Ming Xiong[*], Krithi Ramamritham[†], Jayant Haritsa[‡], John A. Stankovic[§]

Abstract

Data replication *is one of the main techniques by which database systems can hope to meet the stringent temporal constraints of current time-critical applications, especially Web-based directory and electronic commerce services. A pre-requisite for realizing the benefits of replication, however, is the development of high-performance* concurrency control *mechanisms. We present in this paper* **MIRROR** *(Managing Isolation in Replicated Real-time Object Repositories), a concurrency control protocol specifically designed for firm-deadline applications operating on replicated real-time databases. MIRROR augments the optimistic two-phase locking (O2PL) algorithm developed for non real-time databases with a novel and simple to implement state-based conflict resolution mechanism to fine-tune real-time performance.*

Using a detailed simulation model, we compare MIRROR's performance against the real-time versions of a representative set of classical protocols for a range of transaction workloads and system configurations. Our performance studies show that (a) the relative performance characteristics of replica concurrency control algorithms in the real-time environment could be significantly different from their performance in a traditional (non-real-time) database system, (b) MIRROR provides the best performance in both fully and partially replicated environments for real-time applications with low to moderate update frequencies, and (c) MIRROR's conflict resolution mechanism works almost as well as more sophisticated (and difficult to implement) strategies.

[*] Dept. of Computer Science, University of Massachusetts, Amherst, MA 01003. Email: xiong@cs.umass.edu

[†] Dept. of Computer Science, University of Massachusetts, Amherst, MA 01003, also affiliated with India Institute of Technology, Bombay. Email: krithi@cs.umass.edu. Research supported in part by the National Science Foundation Grant IRI-9619588.

[‡] SERC, Indian Institute of Science, Bangalore 560012, India . Email: haritsa@dsl.serc.iisc.ernet.in. Research supported in part by a grant from the Dept. of Science and Technology, Govt. of India.

[§] Dept. of Computer Science, University of Virginia, Charlottesville, VA 22903. Email: stankovic@cs.virginia.edu

1 Introduction

Many *time-critical* database applications are inherently *distributed* in nature. These include the intelligent network services database described in [14], the mobile telecommunication system discussed in [23], and the 1-800 telephone service in the United States. More recent applications include the directory, data-feed and electronic commerce services that have become available on the World Wide Web. The performance, reliability, and availability of such applications can be significantly enhanced through the *replication* of data on multiple sites of the distributed network.

A prerequisite for realizing the benefits of replication, however, is the development of efficient replica management mechanisms. In particular, for many of these applications, especially those related to on-line information provision and electronic commerce, stringent consistency requirements need to be supported while achieving high performance. Therefore, a major issue is the development of efficient *replica concurrency control* protocols. While a few isolated efforts in this direction have been made earlier, they have resulted in schemes wherein either the standard notions of database correctness are not fully supported [17, 18], or the maintenance of multiple historical *versions* of the data is required [16], or the real-time transaction semantics and performance metrics pose practical problems [19]. Further, none of these studies have considered the optimistic two-phase locking (O2PL) protocol [3] although it is the best-performing algorithm in conventional (non-real-time) replicated database systems [3].

In contrast to the above studies, we focus in this paper on the design of *one-copy serializable* concurrency control protocols for replicated real-time databases. Our study is targeted towards real-time applications with "firm deadlines".[1] For such applications, completing a transaction after its deadline has expired is of no utility and may even be harmful. Therefore, transactions that miss their deadlines are "killed", that is, immediately aborted and discarded from the system without being executed to completion. Accordingly, the performance metric is the *percentage of transactions that miss their deadlines*.

[1] In the rest of this paper, when we refer to real-time databases, we mean *firm* real-time databases unless specified otherwise.

Our choice of firm-deadline applications is based on the observation that many of the current distributed real-time applications belong to this category. For example, in the 1-800 service, a customer may hang up the phone if the answer to his query is not provided in a timely manner. Similarly, most Web-based services employ "stateless" communication protocols with timeout features.

The MIRROR Protocol

For the above application and system framework, we present in this paper a replica concurrency control protocol called **MIRROR** (Managing Isolation in Replicated Real-time Object Repositories). MIRROR augments the optimistic two-phase locking (O2PL) algorithm with a novel, simple to implement, state-based conflict resolution mechanism called *state-conscious priority blocking*. In this scheme, the choice of conflict resolution method is a dynamic function of the states of the distributed transactions involved in the conflict. A feature of the design is that acquiring the state knowledge does not require inter-site communication or synchronization, nor does it require modifications to the two-phase commit protocol [7] (the standard mechanism for ensuring distributed transaction atomicity).

Using a detailed simulation model of a *replicated* Real-Time Database System (RTDBS), we compare MIRROR's performance against the real-time versions of a representative set of classical replica concurrency control protocols for a range of transaction workloads and system configurations. These protocols include two phase locking (2PL), optimistic concurrency control (OCC) and optimistic two phase locking (O2PL).

Organization

The remainder of this paper is organized as follows: In Section 2, we present the distributed concurrency control algorithms evaluated in our study. We also develop a practical implementation of the OCC algorithm for replicated data. Section 3 presents existing real-time conflict resolution mechanisms. Section 4 presents MIRROR, the state-conscious conflict resolution mechanism. In Section 5, we describe our distributed real-time database simulation model. Experimental results are presented and discussed in detail in Section 6. Section 7 discusses the related work. Finally, we summarize our conclusions and suggest future research directions in Section 8.

2 Distributed Concurrency Control Protocols

In this section, we review the three classical families of distributed concurrency control (CC) protocols, 2PL [5], OCC and O2PL [3]. All three protocol classes belong to the ROWA ("read one copy, write all copies") category with respect to their treatment of replicated data. While the 2PL and O2PL implementations are taken from the literature, our OCC implementation is new. The discussion of the integration of real-time features into these protocols is deferred to the next section. In the following description, we assume that the reader is familiar with the standard concepts of distributed transaction execution [3, 7, 8].

2.1 Distributed Two-Phase Locking (2PL)

In the distributed two-phase locking algorithm described in [5], a transaction that intends to read a data item has to only set a read lock on *any* copy of the item; to update an item, however, write locks are required on *all* copies. Write locks are obtained as the transaction executes, with the transaction blocking on a write request until all of the copies of the item to be updated have been successfully locked by a local cohort and its remote updaters. Only the data locked by a cohort is updated in the data processing phase of a transaction. Remote copies locked by updaters are updated after those updaters have received copies of the relevant updates with the PREPARE message during the first phase of the commit protocol. Read locks are held until the transaction has entered the prepared state while write locks are held until they are committed or aborted.

2.2 Distributed Optimistic Concurrency Control (OCC)

Our distributed optimistic concurrency control algorithm, OCC, extends the implementation strategy for centralized OCC algorithms proposed in [10] to handle data distribution and replication.

In OCC, transactions execute in three phases: *read*, *validation*, and *write*. In the read phase, cohorts only access data items in their local sites and all updating of replicas is deferred to the end of transaction, that is, to the commit processing phase. More specifically, the two-phase commit (2PC) protocol is "overloaded" to perform validation in its first phase, and then installation of the private updates of successfully validated transactions in its second phase.

The validation process works as follows: After receiving a PREPARE message from its master, a cohort initiates *local* validation. If a cohort fails during validation, it sends an ABORT message to its master. Otherwise, it sends PREPARE messages as well as copies of the relevant updates to all the sites that store copies of its updated data items. Each site which receives a PREPARE message from the cohort initiates an updater to update the data in its <u>local work area</u> used by OCC. When the updates are done, the updater performs local validation and sends a PREPARED message to its cohort. After the cohort collects PREPARED messages from all its updaters, it sends a PREPARED message to the master. If the master receives PREPARED messages from all its cohorts, the transaction is successfully *globally* validated and the master then issues COMMIT messages to all the cohorts.

A cohort that receives a COMMIT message enters the write phase (the third phase) of the OCC algorithm. After it finishes the write phase, it sends a COMMIT message to all its updaters which then complete their write phase in the same manner as the cohort.

For the implementation of the validation test itself, we employ an efficient strategy called *Lock-based Distributed*

Validation, which is described in [22].

An important point to note here is that in contrast to centralized databases where transactions that validate successfully always commit, a distributed transaction that gets locally validated might be aborted later because it fails during global validation. This can lead to wasteful aborts of transactions – other transactions could be aborted when a transaction gets locally validated, but the locally validated transaction itself is aborted later. This is a potential performance drawback for OCC in distributed systems.

2.3 Distributed Optimistic Two-Phase Locking (O2PL)

The O2PL algorithm [3] can be thought of as a hybrid occupying the middle ground between 2PL and OCC. Specifically, O2PL handles read requests in the same way that 2PL does; in fact, 2PL and O2PL are *identical* in the absence of replication. However, O2PL handles replicated data optimistically. When a cohort updates a replicated data item, it requests a write lock immediately on the local copy of the item. But it defers requesting write locks on any of the remote copies until the beginning of the commit phase is reached.

As in the OCC algorithm, replica updaters are initiated by cohorts in the commit phase. Thus, communication with the remote copy site is accomplished by simply passing update information in the PREPARE message of the commit protocol. In particular, the PREPARE message sent by a cohort to its remote updaters includes a list of items to be updated, and each remote updater must obtain write locks on these copies before it can act on the PREPARE request.[2]

Since O2PL waits until the end of a transaction to obtain write locks on copies, both blocking and abort are possible rather late in the execution of a transaction. In particular, if two transactions at different sites have updated different copies of a common data item, one of the transactions has to be aborted eventually after the conflict is detected. In this case, the lower priority transaction is usually chosen for abort in RTDBS.[3]

2.4 Time of Updates to Replicas

It is important to note that the *time* at which the remote update processes are invoked is a function of the choice of CC protocol: In 2PL, a cohort invokes its remote replica update processes to obtain locks *before* the cohort updates a local data item in the transaction execution phase. Replicas are updated during the commitment of the transaction. However, in the O2PL and OCC protocols, a cohort invokes the remote replica update processes only in the *first phase* of the two-phase commit protocol.

[2]To speed up conflict detection, special "copy locks" rather than normal write locks are used for updaters. Copy locks are identical to write locks in terms of their compatibility matrix, but they enable the lock manager to know when a lock is being requested by a replica updater.

[3]The exception occurs when the lower priority transaction is *prepared* in which case the other transaction has to be aborted.

3 Data Conflict Resolution Mechanisms

In this section, we discuss the integration of real-time cognizant data conflict resolution mechanism into the replica concurrency control protocols described in the previous section.

We discuss three different ways to introduce real-time associated priorities into locking protocols:

Priority Blocking (PB): This mechanism is similar to the conventional locking protocol in that a transaction is always blocked when it encounters a lock conflict and can only get the lock after the lock is released. The *lock request queue*, however, is ordered by transaction priority.

Priority Abort (PA): This scheme attempts to resolve all data conflicts in favor of high-priority transactions. Specifically, at the time of a data lock conflict, if the lock holding cohort (updater) has higher priority than the priority of the cohort (updater) that is requesting the lock, the requester is blocked. Otherwise, the lock holding cohort (updater) is aborted and the lock is granted to the requester. Upon the abort of a cohort (updater), a message is sent to the master (cohort) of the cohort (updater) to abort and then restart the whole transaction (if its deadline has not expired by this time).

The only exception to the above policy is when the low priority cohort (updater) has already reached the PREPARED state at the time of the data conflict. In this case, it cannot be aborted unilaterally since its destiny can only be decided by its master and therefore the high priority transaction is forced to wait for the commit processing to be completed.

Priority Inheritance (PI): In this scheme, whenever data conflict occurs the requester is inserted into the lock request queue which is ordered by priority. If the requester's priority is higher than that of any of the current lock holders, then these low priority cohort(s) holding the lock subsequently execute at the priority of the requester, that is, they "inherit" this priority. This means that lock holders always execute either at their own priority or at the priority of the highest priority cohort waiting for the lock, whichever is greater.

The implementation of priority inheritance in distributed databases is not trivial. For example, whenever a cohort inherits a priority, it has to notify its master about the inherited priority. The master propagates this information to all the sibling cohorts of the transaction. This means that the dissemination of inheritance information to cohorts takes time and effort and significantly adds to the complexity of the system implementation.

For the optimistic protocol, OCC, we use the *OPT-WAIT* conflict resolution mechanism [9], [4] described below:

OPT-WAIT: In this mechanism, a transaction that reaches

[4]We have used OPT-WAIT although its variant called *WAIT-50* was found to provide better performance in centralized RTDBS[9], because as explained in [22], several problems arise in extending the *WAIT-50* scheme to the distributed environment.

validation and finds higher priority transactions in its conflict set is "put on the shelf", that is, it is made to wait and not allowed to commit immediately. This gives the higher priority transactions a chance to make their deadlines first. After all conflicting higher priority transactions leave the conflict set, either due to committing or due to aborting, the on-the-shelf waiter is allowed to commit. Note that a waiting transaction might be restarted due to the commit of one of the conflicting higher priority transactions.

4 The MIRROR Protocol

Our new replica concurrency control protocol, **MIRROR**(Managing Isolation in Replicated Real-Time Object Repositories), augments the O2PL protocol described in Section 2 with a novel, simple to implement, state-based conflict resolution mechanism called *state-conscious priority blocking*. In this scheme, the choice of conflict resolution method is a dynamic function of the states of the distributed transactions involved in the conflict. A feature of the design is that acquiring the state knowledge does not require inter-site communication or synchronization, nor does it require modifications of the two-phase commit protocol.

The key idea of the *MIRROR* protocol is to resolve data conflicts based on distributed transaction *states*. As observed in earlier work in centralized RTDBS, it is very expensive to abort a transaction when it is near completion because all the resources consumed by the transactions are wasted [11]. Therefore, in the MIRROR protocol, the state of a cohort/updater is used to determine which data conflict resolution mechanism should be employed. The basic idea is that Priority Abort (PA) should be used in the early stages of transaction execution, whereas Priority Blocking (PB) should be used in the later stages since in such cases a blocked higher priority transaction may not wait too long before the blocking transaction completes. More specifically, it follows the mechanism given below:

State-Conscious Priority Blocking (PA_PB): To resolve a conflict, the CC manager uses PA if the lock holder has not passed a point called the *demarcation point*, otherwise it uses PB.

We assign the demarcation points of a cohort/updater T_i as follows:

- T_i **is a cohort: when** T_i receives a PREPARE message from its master
- T_i **is a replica updater: when** T_i has acquired all the local write locks

Essentially, we want to set the demarcation point in such a way that, beyond that point, the cohort or the updater does not incur any locally induced waits. So, in the case of O2PL, a cohort reaches its demarcation point when it receives a PREPARE message from its master. This happens before the cohort sends PREPARE messages to its remote updaters. It is worth noting that, to a cohort, the difference between PA and PA_PB is with regard to when the cohort reaches the point after which it cannot be aborted by lock conflict. In case of the classical priority abort (PA) mechanism, a cohort enters the PREPARED state after it votes for COMMIT, and a PREPARED cohort cannot be aborted unilaterally. This happens *after* all the remote updaters of the cohort vote to COMMIT. On the other hand, in the PA_PB mechanism, a cohort reaches its demarcation point *before* it sends PREPARE messages to its remote updaters. PA and PA_PB become identical if databases are not replicated. Thus, in state-conscious protocols, cohorts or updaters reach demarcation points only after the two phase commit protocol starts. This means that a cohort/updater cannot reach its demarcation point unless it has acquired all the locks. Note also that a cohort/updater that reaches its demarcation point may still be aborted due to write lock conflict, as discussed earlier in Section 2.3.

4.1 Implementation Complexity

We now comment on the overheads involved in implementing MIRROR in a practical system. First, note that MIRROR does not require any inter-site communication or synchronization to determine when its demarcation points have been reached. This information is known at each local cohort or updater by virtue of its own local state. Second, it does not require any modifications to the messages, logs, or handshaking sequences that are associated with the two-phase commit protocol. Third, the changes to be made to the local lock manager at each site to implement the protocol are quite simple.

4.2 Incorporating PA_PB into the 2PL Protocol

Note that the PA_PB conflict resolution mechanism, which we discussed above in the context of the O2PL-based MIRROR protocol, can be also added to the distributed 2PL protocol.

For 2PL, we assign the demarcation points of a cohort/updater T_i as follows:

- T_i **is a cohort: when** T_i receives a PREPARE message from its master
- T_i **is a replica updater: when** T_i receives a PREPARE message from its cohort

One special effect in combining with 2PL, unlike the combination with O2PL, is that a low priority transaction which has reached its demarcation point and has blocked a high priority transaction will not suffer any lock based waits.

4.3 Choice of Post-Demarcation Conflict Resolution Mechanism

In the above description, we have used Priority Blocking (PB) for the post-demarcation conflict resolution mechanism. Alternatively, we could have used *Priority Inheritance* instead, as given below:

State-Conscious Priority Inheritance (PA_PI):

> To resolve a conflict, the CC manager uses PA if the lock holder has not passed the demarcation point, otherwise it uses PI.

At first glance, the above approach may appear to be significantly *better* than PA_PB since not only are we preventing close-to-completion transactions from being aborted, but also are helping them complete quicker, thereby reducing the waiting time of the high-priority transactions blocked by such transactions. However, as we will show later in Section 6.5, this does not turn out to be the case, and it is therefore the simpler and easier to implement PA_PB that we finally recommend for the MIRROR implementation.

5 Simulation Model

To evaluate the performance of the concurrency control protocols described in Section 2, we developed a detailed simulation model of a distributed real-time database system (DRTDBS). Our model is based on the distributed database model presented in [3], which has also been used in several other studies (for example, [8, 12]) of distributed database system behavior, and the real-time processing model of [21]. A summary of the parameters used in the simulation model are presented in Table 1.

The database is modeled as a collection of $DBSize$ pages that are distributed over $NumSites$ sites. The number of replicas of each page, that is, the "replication degree", is determined by the $ReplDegree$ parameter. The physical resources at each site consist of $NumCPUs$ CPUs, $NumDataDisks$ data disks and $NumLogDisks$ log disks. At each site, there is a single common queue for the CPUs and the scheduling policy is preemptive Highest-Priority-First. Each of the disks has its own queue and is scheduled according to a Head-Of-Line policy, with the request queue being ordered by transaction priority. The $PageCPU$ and $PageDisk$ parameters capture the CPU and disk processing times per data page, respectively. The parameter $InitWriteCPU$ models the CPU overhead associated with initiating a disk write for an updated page. When a transaction makes a request for accessing a data page, the data page may be found in the buffer pool, or it may have to be accessed from the disk. The *BufHitRatio* parameter gives the probability of finding a requested page already resident in the buffer pool.

The communication network is simply modeled as a switch that routes messages and the CPU overhead of message transfer is taken into account at both the sending and receiving sites and its value is determined by the $MsgCPU$ parameter – the network delays are subsumed in this parameter. This means that there are two classes of CPU requests – local data processing requests and message processing requests. We do not make any distinction, however, between these different types of requests and only ensure that all requests are served in priority order.

Parameter	Meaning	Setting
$NumSites$	Number of sites	4
$DBSize$	Number of Pages in the databases	1000 pages
$ReplDegree$	Degree of Replication	4
$NumCPUs$	Number of CPUs per site	2
$NumDataDisks$	Number of data disks per site	4
$NumLogDisks$	Number of log disks per site	1
$BufHitRatio$	Buffer hit ratio on a site	0.1
$ArrivalRate$	Transaction arrival rate (Trans./Second)	Varied
$SlackFactor$	Slack factor in deadline assignment	6.0
$TransSize$	No. of pages accessed per trans.	16 pages
$UpdateFreq$	Update frequency	0.25
$PageCPU$	CPU page processing time	10 ms
$InitWriteCPU$	Time to initiate a disk write	2 ms
$PageDisk$	Disk page access time	20 ms
$LogDisk$	Log force time	5 ms
$MsgCPU$	CPU message send/receive time	1 ms

Table 1. Simulation Model Parameters and Default Settings.

With regard to logging costs, we explicitly model only *forced* log writes since they are done synchronously, i.e., operations of the transaction are suspended during the associated disk writing period. This logging cost is captured by the $LogDisk$ parameter.

Transactions arrive in a Poisson stream with rate $ArrivalRate$, and each transaction has an associated firm deadline, assigned as described below. Each transaction randomly chooses a site in the system to be the site where the transaction originates and then forks off cohorts at all the sites where it has to access data. Transactions in a distributed system can execute in either *sequential* or *parallel* fashion. The distinction is that cohorts in a sequential transaction execute one after another, whereas cohorts in a parallel transaction are started together and execute independently until commit processing is initiated. We consider only sequential transactions in this study. Note, however, that the execution of replica updaters belonging to the same cohort is *always in parallel*.

The total number of pages accessed by a transaction, ignoring replicas, varies uniformly between 0.5 and 1.5 times $TransSize$. These pages are chosen uniformly (without replacement) from the entire database. The proportion of accessed pages that are also updated is determined by $UpdateFreq$.

Upon arrival, each transaction T is assigned a firm completion deadline using the formula

$$Deadline_T = ArrivalTime_T + SlackFactor * R_T$$

where $Deadline_T$, $ArrivalTime_T$, and R_T are the deadline, arrival time, and resource time, respectively, of transaction T, while $SlackFactor$ is a slack factor that provides control of the tightness/slackness of transaction deadlines. The resource time is the total service time at the resources at all sites that the transaction requires for its execution *in the absence of data replication*. This is done because the replica-related cost differs from one CC protocol to another. It is important to note that while transaction resource requirements are used in assigning transaction deadlines, *the* *system itself lacks any knowledge of these requirements* in our model since for many applications it is unrealistic to expect such knowledge. This also implies that a transaction is detected as being late only when it *actually* misses its deadline.

As discussed earlier, transactions in an RTDBS are typically assigned priorities so as to minimize the number of killed transactions. In our model, all cohorts inherit their parent transaction's priority. Messages also retain their sending transaction's priority. The transaction priority assignment used in all of the experiments described here is the widely-used *Earliest Deadline* policy [13], wherein transactions with earlier deadlines have higher priority than transactions with later deadlines.

Deadlock is possible with some of the CC protocols that we evaluate – in our experiments, deadlocks are detected using a time out mechanism. Both our own simulations as well as the results reported in previous studies [2, 6] show that the frequency of deadlocks is extremely small – therefore a low-overhead solution like timeout is preferable compared to more expensive graph-based techniques.

6 Experiments and Results

The performance metric employed is *MissPercent*, the *percentage of transactions that miss their deadlines*. MissPercent values in the range of 0 to 30 percent are taken to represent system performance under "normal" loads, while MissPercent values in the range of 30 to 100 percent represent system performance under "heavy" loads. Several additional statistics are used to aid in the analysis of the experimental results, including the *abort ratio*, which is the average number of aborts per transaction[5], the *message ratio*, which is the average number of messages sent per transaction, the *priority inversion ratio (PIR)*, which is the average number of priority inversions per transaction, and the *wait ratio*, which is the average number of waits per transaction. Further, we also measure the *useful resource utilization* as the resource utilization made by those transactions that are successfully completed before their deadlines. Due to space constraints, not all results are discussed here. Interested readers can refer to [22] for additional details.

All the missed deadline percentage graphs in this paper show mean values that have relative half widths about the mean of less than 10% at the 90% confidence interval, with each experiment having been run until at least 10000 transactions are processed by the system. Only statistically significant differences are discussed here.

6.1 Expt. 1: Baseline – Real-Time Conflict Resolution

Table 1 presents the setting of the simulation model parameters for our first experiment. With these settings, the database is *fully replicated* and each transaction executes in a *sequential* fashion (note, however, that the execution of replica updaters belonging to the same cohort is always in parallel). The parameter values for CPU, disk and message processing times are similar to those in [3]. [6] While these times have certainly reduced due to technology advances in the interim period, we continue to use them here for the following reasons: 1) To enable easy comparison and continuity with the several previous studies that have used similar models and parameter values; 2) The *ratios* of the settings, which is what really matters in determining performance behavior, have changed a lot less as compared to the decrease in absolute values; 3) Our objective is to evaluate the *relative* performance characteristics of the protocols, not their absolute levels. As in several other studies for replicated databases (for example, [1, 19]), here the database size represents only the "hot spots", that is, the heavily accessed data of practical applications, and not the entire database.

Our goal in this experiment was to investigate the performance of the various conflict resolution mechanisms (PA, PI and PA_PB) when integrated with the 2PL and O2PL concurrency control protocols. Since the qualitative performance of the conflict resolution mechanisms was found to be similar for 2PL and O2PL, for ease of exposition and graph clarity we only present the O2PL-based performance results here.

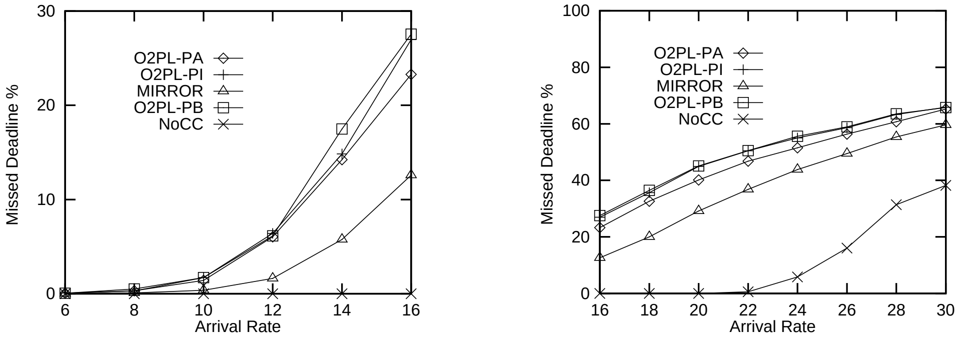
For this experiment, Figures 1(a) and 1(b) present the missed deadline percentages of transactions for the O2PL-PB, O2PL-PA, O2PL-PI, and MIRROR protocols under normal loads and heavy loads, respectively. To help isolate the performance degradation arising out of concurrency control, we also show the performance of NoCC - that is, a protocol which processes read and write requests like O2PL, but ignores any data conflicts that arise in this process and instead grants all data requests immediately. It is important to note that NoCC is only used as an artificial baseline in our experiments.

Focusing our attention first on O2PL-PA, we observe that O2PL-PA and O2PL-PB have similar performance at arrival rates lower than 14 transactions per second, but O2PL-PA outperforms O2PL-PB under heavier loads. This is because O2PL-PA ensures that urgent transactions with tight deadlines can proceed quickly since they are not made to wait for transactions with later deadlines in the event of data conflicts. From collected statistics, we have found out that O2PL-PA greatly reduces the priority inversion ratio, the wait ratio and the wait time as compared to O2PL-PB.

Note, however, that the performance of O2PL-PI and O2PL-PB is *virtually identical*. This is because (1) a low priority transaction whose priority is increased holds the new priority until it commits, i.e., the priority inversion persists for a long time. Thus, higher priority transactions which are blocked by that transaction may miss their dead-

[5] In RTDBS, transaction aborts can arise out of deadline expiration or data conflicts. Only aborts due to data conflicts are included in this statistic.

[6] The log force time is much smaller than that of a disk read/write operation because logging activities are sequential disk operations.



(a) Normal Load (b) Heavy Load

Figure 1. **O2PL-based Algorithms**

lines. In contrast, normal priority inheritance in real-time systems only involves critical sections which are usually short so that priority increase of a task only persists for a short time, i.e., until the low priority task gets out of the critical section. This is the primary reason that priority inheritance works well for real-time tasks accessing critical sections, but it fails to improve performance in real-time transaction processing; (2) it takes considerable time for priority inheritance messages to be propagated to the sibling cohorts (or updaters) on different sites, and (3) under high loads, high priority transactions are repeatedly data-blocked by lower priority transactions. As a result, many transactions are assigned the same priority by "transitive inheritance" and priority inheritance essentially degenerates to "no priority", i.e., to basic O2PL, defeating the original intention. This is confirmed by the similar priority inversion ratio (PIR), wait ratio and wait time statistics of O2PL-PI and O2PL-PB collected in our experiments. Hence, we conclude that *priority inheritance does not help to improve performance in distributed environment.*

Turning our attention to the MIRROR protocol, we observe that MIRROR has the best performance among all the protocols. The improved behavior here is due to MIRROR's feature of avoidance of transaction abort after a cohort/updater has reached its demarcation point. The performance improvement obtained in MIRROR can be explained as follows: Under O2PL-PA, priority inversions that occur beyond the demarcation point involving a lower priority (unprepared) cohort/updater result in transaction abort. On the other hand, under MIRROR, such priority inversions do not result in transaction abort. In such situations, a high priority transaction may afford to wait for a lower priority transaction to commit since it is near completion, and wasted resources due to transaction abort can be reduced, as is done by the MIRROR protocol.

In fact, MIRROR does better than all the other O2PL-based algorithms under all the workload ranges that we tested.

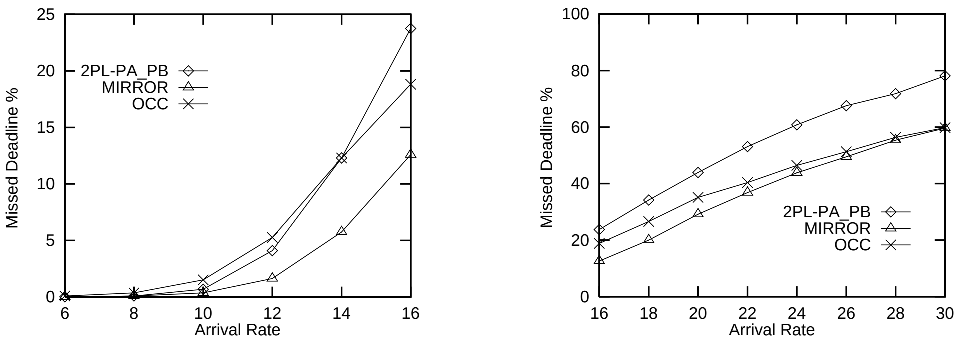
6.2 Expt. 2: Baseline - Concurrency Control Algorithms

The goal of our next experiment was to investigate the performance of CC protocols based on the three different techniques: 2PL, O2PL and OCC. For this experiment, the parameter settings are the same as those used for Experiment 1. The missed deadline percentage of transactions is presented in Figures 2(a) and 2(b) for the normal load and heavy load regions, respectively.

Focusing our attention on the locking-based schemes, we observe that MIRROR outperforms 2PL-PA_PB in both normal and heavy workload ranges. For example, MIRROR outperforms 2PL-PA_PB by about 12% (absolute) at an arrival rate of 14 transactions/second. This can be explained as follows: First, 2PL results in much higher message overhead for each transaction, as was indicated by the message ratio statistic collected in the experiments. The higher message overhead results in higher CPU utilization, thus aggravating CPU contention. Second, 2PL-PA_PB detects data conflicts earlier than MIRROR. However, data conflicts cause transaction blocks or aborts. 2PL-PA_PB results in more number of waits per transaction and longer wait time per wait instance. Thus 2PL-PA_PB results in more transaction blocks and longer blocking times than MIRROR. On the other hand, MIRROR has less transaction blocks. In other words, unlike in 2PL-PA_PB, a cohort with O2PL cannot be blocked or aborted by data conflicts with cohorts on other sites before one of them reaches the commit phase. Thus, with MIRROR, transactions can proceed faster. In addition, MIRROR improves performance by detecting global CC conflicts late in the transaction execution thereby reducing wasted transaction aborts.

Turning our attention to the OCC protocol, we observe that OCC is slightly worse than 2PL-PA_PB and MIRROR under arrival rates less than 14 transactions/second. This is due to the fact that OCC has a higher CC abort ratio than 2PL-PA_PB and MIRROR under those loads. With higher loads, OCC outperforms 2PL-PA_PB because OCC has less number of wasteful aborts, less number of waits and shorter blocking time of a transaction than 2PL-PA_PB.

It may be considered surprising that MIRROR has the



(a) Normal Load (b) Heavy Load

Figure 2. **2PL, O2PL and OCC Algorithms**

best performance over a wide workload range, improving slightly even over OCC. We observe that MIRROR has higher *useful* CPU and disk utilization, even though its *overall* CPU and disk utilization is lower than OCC. This clearly indicates that OCC wastes more resources than MIRROR does. It implies that the average progress made by transactions before they were aborted due to CC conflicts is larger in OCC than that in MIRROR.[7] As observed in the previous studies of centralized RTDB settings[9], the wait control in OCC can actually cause all the conflicting transactions of a validating transaction to be aborted at a later point in time, thereby wasting more resources even if OCC has slightly less CC abort ratio than MIRROR. In contrast, MIRROR reduces wasted resources by avoiding transaction aborts after cohorts/updaters reach demarcation points.

In summary, although OCC outperforms 2PL-PA_PB, MIRROR, the protocol of O2PL augmented with PA_PB, outperforms OCC in the tested workloads.

6.3 Expt. 3: Varying Update Frequency

The next experiment investigates the performance of these algorithms under different update frequencies. For this experiment, Figure 3(a) and (b) present the missed deadline percentage when the update frequencies are low and high for an arrival rate of 14 transactions/second. It should be noted that data is normally replicated in distributed database systems only when the update frequency is not very high. Therefore, the high update frequency results that we present here are only to aid in understanding the tradeoffs of different protocols.

When the update frequency is comparatively low (less than 0.5), we observe that the qualitative behavior of the various algorithms is similar to that of Experiment 1. A difference, however, occurs when the update frequency is high (more than 0.5). We observe in Figure 3(b) that the performance of MIRROR degrades more drastically with the increase of update frequency. For example, MIRROR performs slightly worse than both 2PL-PA_PB and OCC when the update frequency is 1.0. The reason for the degraded performance of MIRROR is that with high update frequency, MIRROR causes much more aborts due to both data contention in the local site and global update conflicts, as discussed earlier in Section 6.2, and more aborts are wasted under MIRROR.

In summary, for low to moderate update frequencies, MIRROR is the preferred protocol. For high update frequencies, on the other hand, OCC performs better than MIRROR.

6.4 Expt. 4: Partial Replication

For this experiment, the $NumSites$ and $DBSize$ are fixed at 8 and 800, respectively, while the $NumCPUs$ and $NumDataDisks$ per site are set at 1 and 2, respectively. These changes were made to provide a system operational region of interest without having to model very high transaction arrival rates. The other parameter settings are the same as those given in Table 1. For this environment, Figure 4(a) presents the missed deadline percentage of transactions when the number of replicas is varied from 1 to 8, i.e., from no replication to full replication, for an arrival rate of 14 transactions/second.

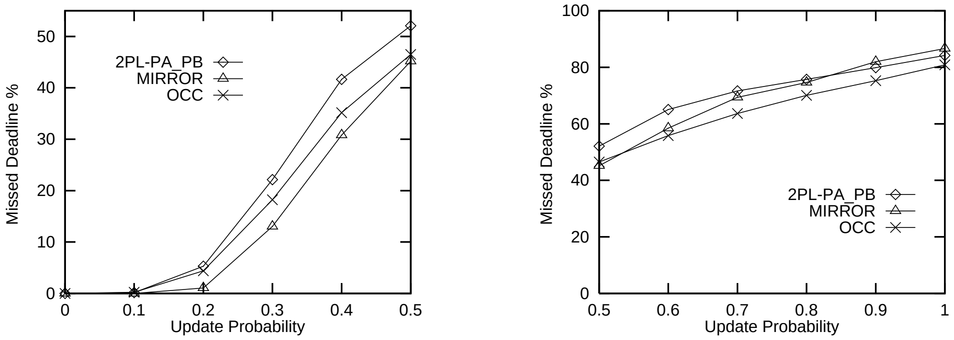
In the absence of replication, we observe first that 2PL-PA_PB and MIRROR perform identically as expected since O2PL reduces to 2PL in this situation. Further, OCC outperforms all the other algorithms.

As the number of replicas increases, the performance difference between MIRROR and 2PL-PA_PB increases. Because of its inherent mechanism for detecting data conflicts, 2PL-PA_PB suffers much more from data replication than MIRROR and OCC do. We observe a *performance crossover* between MIRROR and OCC. The reason for this change in their relative performance behavior is explained in the abort curves shown in Figure 4(b) (for graph clarity, we only show the abort ratio and useful abort ratio of MIRROR and OCC), where we see that the number of aborts of MIRROR is significantly reduced while data is replicated. This helps reduce the resource wastage in MIRROR.

We also observe a bowl curve for the missed deadline percentage of MIRROR.[8] In O2PL, read operations can

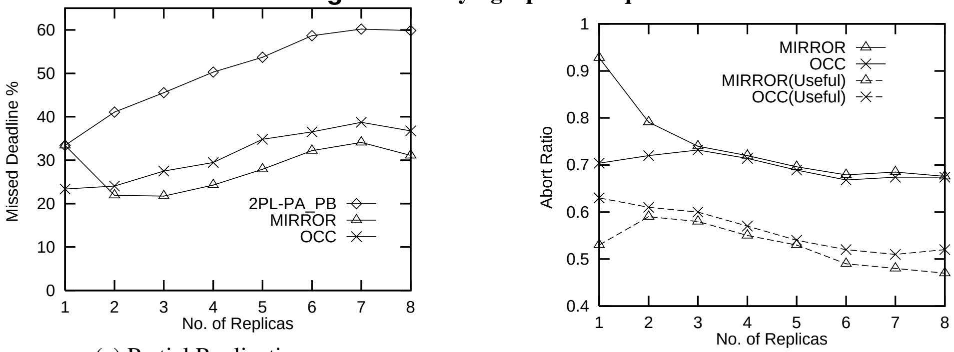
[7]A transaction's progress is measured in terms of the resources it has consumed.

[8]The same bowl curve is also observed for the performance of OCC when the update frequency is lower.



(a) Low $UpdateFreq$ (b) High $UpdateFreq$

Figure 3. Varying UpdateFreq



(a) Partial Replication ($DBSize = 800, NumSites = 8$) (b) Abort Ratio (Partial Replication)

Figure 4. Partial Replication

benefit from local data when data is replicated. However, as data replication level goes up, update operations suffer due to updates to remote data copies. Hence, the performance degrades after a certain replication level. On the other hand, we observe that the performance of 2PL-PA_PB always degrades as data replication level goes up. This is due to the pessimistic conflict detection mechanism in 2PL since the number of messages sent out for conflict detection increases drastically which in turn increases CPU contention. The similar behavior of OCC and 2PL is also observed in conventional replicated databases [4].

6.5 Efficiency of MIRROR

Our previous experiments demonstrated MIRROR's ability to provide good performance. But there still remains the question of how efficient is MIRROR's use of its state knowledge – in particular, wouldn't replacing the PA_PB mechanism with the PA_PI mechanism described in Section 4, wherein priority inheritance is used for conflict resolution after the demarcation point, result in *even better* performance? This expectation is because, as mentioned earlier in Section 3, PI seems capable of providing earlier termination of the (priority-inversion) blocking condition than PB.

We conducted experiments to evaluate the above possibility and found that the performance of O2PL-PA_PI was *virtually identical* to that of MIRROR in all cases. The reason for this perhaps counter-intuitive result is that priority-inheritance in the distributed environment involves excessive message costs and dissemination delay, thereby neutralizing its positive points.

In summary, MIRROR provides the same level of performance as O2PL-PA_PI without attracting its implementation difficulties – we therefore recommend it as the algorithm of choice for replicated RTDBS.

6.6 Summary of Experimental Results

Apart from the experiments described above, we have conducted a variety of experiments that cover a range of workloads and system configurations, including "infinite" resources[9] to isolate the impact of data contention, variations in *message cost*, *message propagation delay*, *slack factor* and *data access ratio*, etc. The results in these other experiments were qualitatively similar to those shown here and included in [22]. Table 2 summarizes these results under both tight and loose slack factor: In the table, system parameters, i.e., load, message cost, data access ratio (DAR) and update frequency have been coarsely categorized into low and high, and '*' refers to both low and high categories. The terms "poor", "fair", "good", and "best" are used to describe the relative performance in a given system state and for a given algorithm. Whereas in a particular row, "fair" is better than "poor", "good" is better than "fair", and "best" represents the best algorithm in a row, the terms in two different rows are not comparable. The following general observations pertain to Table 2.

[9]No queuing for the CPU and disk resources.

Parameter				Algorithms' Performance									
				2PL				O2PL				OCC
Load	MsgCost	DAR	UpdateFreq	PB	PI	PA	PA_PB	PB	PI	PA	MIRROR	Wait
Low	Low	High	Low	Poor	Poor	Fair	Good	Poor	Poor	Good	Best	Good
Low	High	High	Low	Poor	Poor	Poor	Poor	Poor	Poor	Good	Best	Good
High	Low	High	Low	Poor	Poor	Poor	Fair	Poor	Poor	Fair	Best	Good
High	High	High	Low	Poor	Poor	Poor	Poor	Fair	Fair	Good	Best	Good
*	*	High	High	Poor	Poor	Poor	Poor	Poor	Poor	Poor	Good	Best
*	*	Low	Low	Fair	Fair	Fair	Fair	Good	Good	Good	Best	Good
*	*	Low	High	Poor	Poor	Poor	Poor	Good	Good	Good	Good	Good

Table 2. Performance of Algorithms.

1. 2PL based algorithms perform poorly in most cases, especially when the message cost is high. Thus 2PL based algorithms are not the proper choices for high message cost environments.
2. O2PL-PA and MIRROR achieve good performance for low to moderate update frequencies but the O2PL approach does not work well at high update frequencies.
3. OCC achieves better performance than all the O2PL-based and 2PL-based algorithms, except for MIRROR, over most of the update frequency range.
4. Protocols integrated with only PB or PI (e.g., O2PL-PB, O2PL-PI) always perform poorly. Thus they are not suited to distributed real-time databases. A similar poor performance of these mechanisms has also been observed earlier for centralized real-time databases [9].
5. No single algorithm can *always* outperform all the others: MIRROR (O2PL-PA_PB) performs best for low to moderate update frequencies whereas OCC performs best at high update frequencies. However, since we expect that most replicated RTDBS applications will belong to the former category, MIRROR appears to be the best overall choice for implementation in these systems.

7 Related Work

Concurrency control algorithms and real-time conflict resolution mechanisms for RTDBS have been studied extensively (e.g. [9, 10, 11, 19]). However, concurrency control for replicated DRTDBS has only been studied in [16, 17, 18, 19]. An algorithm for maintaining consistency and improving the performance of replicated DRTDBS is proposed in [16]. In this algorithm, a *multiversion* technique is used to increase the degree of concurrency. Replication control algorithms that integrate real-time scheduling and replication control are proposed in [17, 18]. These algorithms employ Epsilon-serializability (ESR) [20] which is less stringent than conventional one-copy-serializability.

In contrast to the above studies, our work retains the standard one-copy-serializability as the correctness criterion and focuses on the locking and OCC based concurrency control protocols.

The performance of the classical distributed 2PL locking protocol (augmented with the priority abort (PA) and priority inheritance(PI) conflict resolution mechanisms) and of OCC algorithms in replicated DRTDBS was studied in [19] for real-time applications with "soft" deadlines. [10] The results indicate that 2PL-PA outperforms 2PL-PI only when the update transaction ratio and the level of data replication are both low. Similarly, the performance of OCC is good only under light transaction loads.

Making clear-cut recommendations on the performance of protocols in the soft deadline environment is rendered difficult, however, by the following: (1) There are *two* metrics – Missed Deadlines and Mean Tardiness, and protocols which improve one metric usually degrade the other. (2) The choice of the post-deadline value function has considerable impact on relative protocol performance; (3) There is no inherent load control, so the system could enter an unstable state. Due to such problems with the soft-deadline framework and, more importantly, because many of the replicated applications fall into the firm-deadline category, we have modeled firm real-time transactions in this paper. Finally, we also include an investigation of the O2PL algorithm which has not been studied before in the real-time context.

In [11], a *conditional priority inheritance* mechanism is proposed to handle priority inversion. This mechanism capitalizes on the advantages of both priority abort and priority inheritance in real-time data conflict resolution. It outperforms both priority abort and priority inheritance when integrated with two phase locking in centralized real-time databases. However, the protocol assumes that the *length* (in terms of the number of data accesses) of transactions is known in advance which may not be practical in general, especially for distributed applications. In contrast, our *state-conscious priority blocking* and *state-conscious priority inheritance* protocols resolve real-time data conflicts based on the *states* of transactions rather than their lengths.

8 Conclusions

In this paper, we have addressed the problem of accessing replicated data in distributed real-time databases where transactions have firm deadlines, a framework under which many current time-critical applications, especially Web-based ones, operate. In particular, for this environment we proposed a novel state-conscious protocol called MIRROR which can be easily integrated and implemented in current systems, and investigated its performance relative

[10] With soft deadlines, a reduced value is obtained by the application from transactions that are completed after their deadlines have expired.

to the performance of the 2PL, O2PL and OCC based concurrency control algorithms. Our performance studies show the following:

1. The relative performance characteristics of replica concurrency control algorithms in the real-time environment could be significantly different from their performance in a traditional (non-real-time) database system. For example, the O2PL algorithm, which is reputed to provide the best overall performance in traditional databases, performs poorly in real-time databases.
2. OCC outperforms 2PL and O2PL based algorithms when these locking based algorithms are integrated with priority blocking, priority abort and priority inheritance protocols.
3. The MIRROR protocol provides the best performance in both fully and partially replicated environments for real-time applications with low or moderate update frequencies. For high update frequencies, however, OCC is better. But, given that most of the distributed real-time applications that we are aware of fall into the former category, MIRROR appears to be an attractive choice for designers of replicated RTDBS.
4. As mentioned above, MIRROR implements a state-conscious priority blocking-based conflict resolution mechanism. We also evaluated alternative implementations of MIRROR with more sophisticated, and difficult to implement, conflict resolution mechanisms such as *state-conscious priority inheritance*. Our experiments demonstrate, however, that little value is added with these enhancements – that is, the basic simple implementation of MIRROR itself is sufficient to deliver good performance.

We are currently working on combining the MIRROR protocol with the optimistic distributed real-time commit protocol of [8] to investigate the performance improvements that may arise out of this combination. We will also investigate mechanisms for deferred update propagation of replicas in real-time databases.

References

[1] Anderson, T., Breitbart, Y., Korth, H. and Wool, A., "Replication, Consistency, and Practicality: Are These Mutually Exclusive?" *Proceedings of the ACM-SIGMOD 1998 International Conference on Management of Data,* Seattle, WA., pages 484-495, 1998.

[2] Agrawal, R., Carey, M., and McVoy, L., "The Performance of Alternative Strategies for Dealing With Deadlocks in Database Management Systems", *IEEE TOSE*, Dec 1987.

[3] Carey, M., and Livny, M., "Conflict Detection Tradeoffs for Replicated Data," *ACM Transactions on Database Systems,* Vol. 16, pp. 703-746, 1991.

[4] Ciciani, B., Dias, D. M., Yu, P. S., "Analysis of Replication in Distributed Database Systems," *IEEE Transactions on Knowledge and Data Engineering,* Vol. 2, No. 2, June 1990.

[5] Gray, J., "Notes On Database Operating Systems," in *Operating Systems: An Advanced Course,* R. Bayer, R. Graham, and G. Seegmuller, eds., Springer-Verlag, 1979.

[6] Gray, J., Homan, P., Obermarck, R., and Korth, H., "A Strawman Analysis of Probability of Waiting and Deadlock in a Database System," IBM Res. Rep. RJ 3066, San Jose, CA.

[7] Gray, J. and Reuter A., "Transaction Processing: Concepts and Techniques," *Morgan Kaufmann*, 1992.

[8] Gupta, R., Haritsa, J., Ramamritham, K., and Seshadri S., "Commit Processing in Distributed Real-Time Database Systems," *Proceedings of 17th IEEE Real-Time Systems Symposium*, 1996.

[9] Haritsa, J. R., Carey, M., and Livny, M., "Data Access Scheduling in Firm Real-Time Database Systems," The Journal of Real-Time Systems, 4, 203-241 (1992).

[10] Huang, J., Stankovic, J.A., Ramamritham, K., Towsley, D., "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *Proc. of the 17th International Conference on Very Large Data Bases,* Barcelona, September, 1991.

[11] Huang, J., Stankovic, J.A., Ramamritham, K., Towsley, D., and Purimetla, B., "Priority Inheritance In Soft Real-Time Databases," *The Journal of Real-Time Systems,* 4, pp. 243-268, 1992.

[12] Lam, K.Y., "Concurrency Control in Distributed Real-Time Database Systems," *Ph.D. Dissertation*, City University of Hong Kong, Oct., 1994.

[13] Liu, C., and Layland, J., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, 20(1), 1973.

[14] Purimetla, B., Sivasankaran, R., Stankovic, J.A., and Ramamritham, K., "A Study of Distributed Real-Time Active Database Applications," *IEEE Workshop on Parallel and Distributed Real-Time Systems,* Newport Beach, California, 1993.

[15] Sha, L., Rajkumar, R., and Lehoczky, J.P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," In *IEEE Transactions on Computers,* vol. 39, pp. 1175-1185, Sep. 1990.

[16] Son, S., "Using Replication for High Performance Database Support in Distributed Real-Time Systems," *Proceedings of the 8th IEEE Real-Time Systems Symposium,* pp. 79-86, 1987.

[17] Son, S., and Kouloumbis, S., "A Real-Time Synchronization Scheme for Replicated Data in Distributed Database Systems," *Information Systems*, 18(6), 1993.

[18] Son, S., and Zhang, F., "Real-Time Replication Control for Distributed Database Systems: Algorithms and Their Performance," *4th International Conference on Database Systems for Advanced Applications,* Singapore, April, 1995.

[19] Ulusoy, O., "Processing Real-Time Transactions in a Replicated Database System," *Distributed and Parallel Databases,* 2, pp. 405-436, 1994.

[20] Wu, K.L., Yu, P.S. and Pu, C., "Divergence Control for Epsilon-Serializability," In *Proceedings of Eighth International Conference on Data Engineering,* Phoenix, February 1992.

[21] Xiong, M., Sivasankaran, R., Stankovic, J.A., Ramamritham, K. and Towsley, D., "Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics," *Proceedings of the 17th IEEE Real-Time Systems Symposium,* pp. 240-251, Washington, DC, December 1996.

[22] Xiong, M., Ramamritham, K., Haritsa, J., and Stankovic, J., "MIRROR: A State-Conscious Concurrency Control Protocol in Replicated Real-Time Databases," *Technical Report 98-36,* Department of Computer Science, University of Massachusetts at Amherst, 1998 (http://www-ccs.cs.umass.edu/rtdb/publications.html).

[23] Yoon, Y., "Transaction Scheduling and Commit Processing for Real-Time Distributed Database Systems", *Ph.D. Thesis*, Korea Adv. Inst. of Science and Technology, May 1994.