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

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

Abstract

Data replication can help database systems meet the stringent temporal constraints of current time-critical applications, especially Internet-based services. A pre-requisite, however, is the development of high-performance concurrency control mechanisms. We present here a new CC protocol called **MIRROR** *(Managing Isolation in Replicated Real-time Object Repositories), 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 easily implementable state-based conflict resolution mechanism to fine-tune the real-time performance.*

A simulation-based study of MIRROR's performance against the real-time versions of a representative set of classical protocols shows 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) compared to locking based protocols, MIRROR provides the best performance in replicated environments for real-time applications, and (c) MIRROR's conflict resolution mechanism works almost as well as more sophisticated (and difficult to implement) strategies.

1 Introduction

Many *time-critical* database applications are inherently *distributed* in nature. These include the intelligent network services database described in [18], the mobile telecommunication system discussed in [30], and the 1-800 telephone service in the United States. More recent applications include the multitude of directory, data-feed and electronic commerce services that have become available on the World Wide Web – for example, organizations the world over are starting to deploy Lightweight Directory Access Protocol (LDAP) accessible directories as part of their information infrastructure [13, 14].

The performance, reliability and availability of such applications can be significantly enhanced through the *replication* of data on multiple sites of the distributed network. This has been the main motivation, for example, for the now commonplace "mirror sites" on the Web for heavily accessed services. In fact, a large percentage of the data that directories are expected to provide reside in replicated relational databases.

A pre-requisite 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 [23, 24], or the maintenance of multiple historical *versions* of the data is required [22], or the real-time transaction semantics and performance metrics pose practical problems [26]. Further, none of these studies have considered the optimistic two-phase locking (O2PL) protocol [4] although it is the best-performing algorithm in conventional (non-real-time) replicated database systems [4].

In contrast to the above studies, we focus in this paper on the design of *one-copy serializable* [2] 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

[*] 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, and India Institute of Technology, Bombay, India. Email: krithi@cs.umass.edu

[‡] SERC, Indian Institute of Science, Bangalore 560012, India. Email: haritsa@dsl.serc.iisc.ernet.in

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

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

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*.

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 system responds with a "circuits are busy" message if a connection can not be made before the deadline. Similarly, most Web-based services employ "stateless" communication protocols with timeout features.

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 [8] (the standard mechanism for ensuring distributed transaction atomicity).

Using a detailed simulation model of a *replicated* 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) and optimistic two phase locking (O2PL).

The remainder of this paper is organized as follows: In Section 2, we present the locking-based distributed concurrency control algorithms evaluated in our study. In particular, we describe MIRROR, our new state-conscious conflict resolution protocol. The distributed real-time database simulation model is described in Section 3. Experimental results are presented and summarized in Section 4. Section 5 discusses the related work. Finally, we summarize our conclusions and suggest future research directions in Section 6.

2 Locking-based Distributed Concurrency Control Protocols

In this section, we review the 2PL [6] and O2PL [4] distributed CC protocols, and present our new MIRROR protocol. All three protocols belong to the ROWA ("read one copy, write all copies") category with respect to their treatment of replicated data. In the following description, we assume that the reader is familiar with the standard concepts of distributed transaction execution [4, 8, 9].

2.1 Distributed Two-Phase Locking (2PL)

In the distributed two-phase locking algorithm described in [6], 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 Two-Phase Locking (O2PL)

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.

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] 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.

2.3 The MIRROR Protocol

We now present our new replica concurrency control protocol called **MIRROR** (Managing Isolation in Replicated Real-Time Object Repositories). MIRROR augments the O2PL protocol 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.

Two real-time conflict resolution mechanisms are used in MIRROR. They are *Priority Blocking* (PB) and *Priority Abort* (PA). PB 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.

PA, on the other hand, 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.

The key idea of the MIRROR protocol is to resolve data conflicts based on distributed transaction *states*. As observed in earlier work on 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[12]. Therefore, in the MIRROR protocol, the state of a cohort (or 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 (or 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 (or updater) cannot reach its demarcation point unless it has acquired all the locks. Note also that a cohort (or updater) that reaches its demarcation point may still be aborted due to write lock conflict, as discussed earlier in Section 2.2.

2.3.1 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.

2.3.2 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 (or 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 because it already has all the locks.

2.3.3 Choice of Post-Demarcation Conflict Resolution Mechanism

In the above description of PA_PB, we have used Priority Blocking (PB) for the post-demarcation conflict resolution mechanism. Alternatively, we could have used *Priority Inheritance* (PI) 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, this approach may appear to be significantly *better* than PA_PB since not only are we preventing close-to-completion transactions from being aborted, but are also 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 4, 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.

3 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 [4], which has also been used in several other studies (for example, [9, 15]) of distributed database system behavior, and the real-time processing model of [28]. 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. Each page is fully replicated in our experiments. 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

Parameter	Meaning	Setting
$NumSites$	Number of sites	4
$DBSize$	Number of Pages in the databases	1000 pages
$NumCPUs$	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
$ArrivalRate$	Transactions arrived per second	Varied
$SlackFactor$	Slack factor in deadline assignment	6.0
$TransSize$	Average No. of pages accessed per trans.	16 pages
$UpdateProb$	Update frequency per page access	0.25
$PageCPU$	CPU processing time per page	10 ms
$InitWriteCPU$	Time to initiate a disk write	2 ms
$PageDisk$	Disk access time per page	20 ms
$LogDisk$	Log force time	5 ms
$MsgCPU$	CPU message send/receive time	1 ms

Table 1. Simulation Model Parameters and Default Settings.

$PageCPU$ 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.

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 probability of accessed pages that are also updated is determined by $UpdateProb$.

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 [21]. 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 First* policy [17], 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 [3, 7] 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.

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[4], 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.

All the $MissPercent$ 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.

4 Experiments and Results

4.1 Expt. 1: 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. The parameter values for the CPU, disk and message processing times are similar to those in [4]. [5] As in several other studies for replicated databases (for example, [1, 26]), 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, PB, 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 $MissPercent$ behavior of the O2PL-PB, O2PL-PA, O2PL-PI, and MIRROR (O2PL-PA_PB) 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.

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. This is clearly brought out in Figures 1(c), 1(d) and 1(e) which present the priority inversion ratio, the wait ratio

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

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

and the wait time statistics, respectively. These figures show that O2PL-PA greatly reduces these factors as compared to O2PL-PB. In contrast to centralized RTDBS, a *non-zero* priority inversion ratio for O2PL-PA is seen in Figure 1(c) – this is due to the inherent non-preemptability of *prepared* data of a low priority cohort (updater) which 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 conflicting high priority transaction is forced to wait for the commit processing to be completed.

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 deadlines. 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 in Figures 1(c), 1(d) and 1(e) where we observe that O2PL-PI and O2PL-PB have similar priority inversion ratio (PIR), wait ratio and wait time statistics. The similar performance results of PI and PB was also observed in our other experiments. Hence, we conclude that *priority inheritance does not help in improving performance in the distributed real-time 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 (or 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 (or updater) result in transaction abort. On the other hand, under MIRROR, such priority inversions do not result in transaction abort. The importance of this is quantified in Figure 1(f), where it is seen that a significant number of priority inversions due to unprepared data take place *after* the demarcation point. 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.

4.2 Expt. 2: Concurrency Control Algorithms

The goal of this experiment was to investigate the performance of CC protocols based on the two different techniques: 2PL and O2PL. The $MissPercent$ behavior of 2PL-PA_PB and MIRROR is presented in Figures 2(a) and 2(b) for the normal load and heavy load regions, respectively.

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 is clearly brought out in Figure 2(c), which profiles the message ratio statistic. 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. In Figures 2(d) and 2(e), it is clear that 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 fewer transaction blocks. In other words, unlike in 2PL-PA_PB, a cohort with O2PL cannot be blocked by data conflicts with cohorts or updaters on other sites before it reaches the commit phase. Thus, with MIRROR, transactions can proceed faster.

Figure 2(f) presents the *overall* and *useful* concurrency control (CC) abort ratios for the various protocols. The *useful* CC abort ratio only includes aborts caused by eventually committed transactions. Focusing our attention on 2PL-PA_PB and MIRROR, under normal workload range, MIRROR has significantly lower CC abort ratio than 2PL-PA_PB. Thus MIRROR utilizes resources better under normal workloads. Under heavy workload range, MIRROR has higher CC abort ratios than 2PL-PA_PB. This can be explained as follows : In case of O2PL, it is possible for each copy site to have an update transaction obtain locks locally, but discover the existence of competing update transaction(s) at other sites only after successfully executing locally. Discovering the conflict at this late stage leaves no option for resolving the conflicts except to abort the lower priority transaction. However, we also observe that MIRROR has much higher *useful* CC abort ratio than 2PL-PA_PB even though its *total* abort ratio is slightly higher than 2PL-PA_PB. For example, at an arrival rate of 22 trans-

actions/second, the total and useful abort ratio of MIRROR are 0.78 and 0.51, respectively, while the total and useful abort ratio of 2PL-PA_PB at the same arrival rate are 0.69 and 0.26, respectively. Thus the wasteful CC abort ratio of MIRROR and 2PL-PA_PB are 0.27 and 0.43, respectively. This clearly demonstrates that MIRROR improves performance by detecting global CC conflicts late in the transaction execution, thereby reducing wasted transaction aborts.

4.3 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 2.3, wherein priority inheritance is used for conflict resolution after the demarcation point, result in *even better* performance? This expectation is because 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.

5 Related Work

Concurrency control algorithms and real-time conflict resolution mechanisms for RTDBS have been studied extensively (e.g. [10, 11, 12, 26]). However, concurrency control for replicated DRTDBS has only been studied in [22, 23, 24, 26]. An algorithm for maintaining consistency and improving the performance of replicated DRTDBS is proposed in [22]. 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 [23, 24]. These algorithms employ Epsilon-serializability (ESR) [27] 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 based concurrency control protocols. We also include an investigation of the O2PL algorithm which has not been studied before in the real-time context.

The performance of the classical distributed 2PL locking protocol (augmented with the priority abort (PA) and priority inheritance(PI) conflict resolution mechanisms) and of validation-based algorithms was studied in [26] for real-time applications with "soft" deadlines operating on replicated DRTDBS.[6] 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.

In [12], 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.

6 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 to the performance of the 2PL and O2PL 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 non real-time O2PL algorithm, which is reputed to provide the best overall performance in traditional databases, performs poorly in real-time databases.

2. Generally, O2PL based algorithms perform better than their 2PL counterparts. In particular, the MIRROR protocol provides the best performance in replicated environments for real-time applications.

3. As mentioned above, MIRROR implements a state-conscious priority blocking-based conflict resolution

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

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 [9] to investigate the performance improvements that may arise out of this combination.

Acknowledgements

The work of J. Haritsa was partially supported by a grant from the Dept. of Science and Technology, Govt. of India.

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] Bernstein, P. A., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," *Addison-Wesley Publishing Company*, 1987.

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

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

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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).

[11] 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.

[12] 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.

[13] http://ftp.sunet.se/ftp/pub/x500/ldap/papers/ldap.ps

[14] http://www.umich.edu/cgi-bin/ldapman

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

[16] Lin, K. and Lin, M., "Enhancing Availability in Distributed Real-time Databases," *ACM Sigmod Record*, Vol. 17, No. 1, 1988.

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

[18] 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.

[19] Sha, L., Rajkumar, L., and Lehoczky, J.P., "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record,* 17(1), March, 1988.

[20] 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.

[21] Stankovic, J.A., Zhao, W., "On Real-Time Transactions," *ACM Sigmod Record*, March 1988.

[22] 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.

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

[24] 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.

[25] Thomasian, A., and Rahm, E., "A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking," *Proceedings of the 10th International Conference on Distributed Computing Systems,* Paris, France, May 1990.

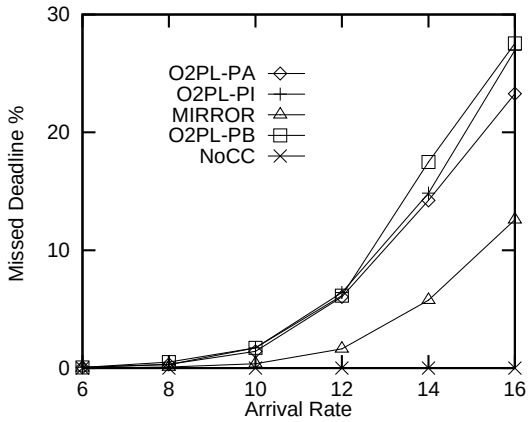
[26] Ulusoy, O., "Processing Real-Time Transactions in a Replicated Database System," *Distributed and Parallel Databases,* 2, pp. 405-436, 1994.

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

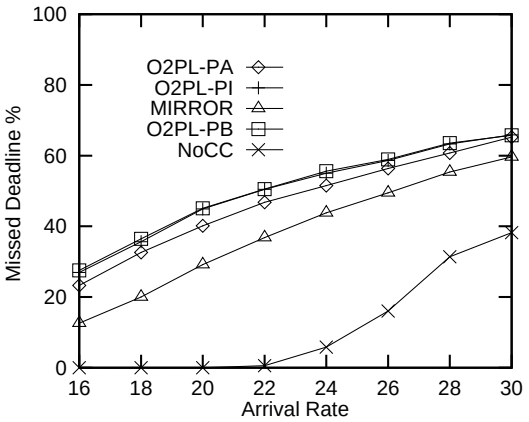
[28] Xiong, M., Sivasankaran, R., Stankovic, J.A., Ramamritham, K. and Towsley, D., "Scheduling Transactions with Temporal Constraints: Exploiting Data Semantics," *Proceedings of the 17th IEEE Real-Time Systems Symposium,* pp. 240-251, Washington, DC, December 1996.

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

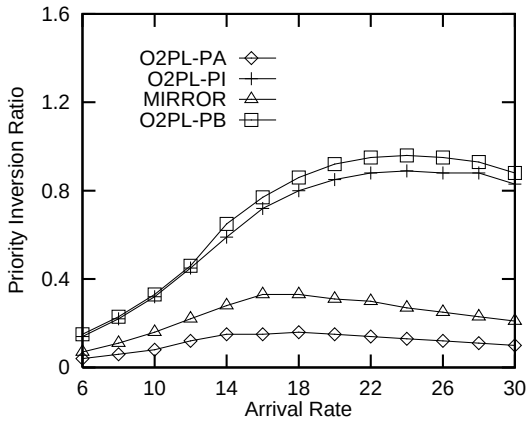
[30] Yoon, Y., "Transaction Scheduling and Commit Processing for Real-Time Distributed Database Systems", *Ph.D. Thesis*, Korea Adv. Inst. of Science and Technology, May 1994.



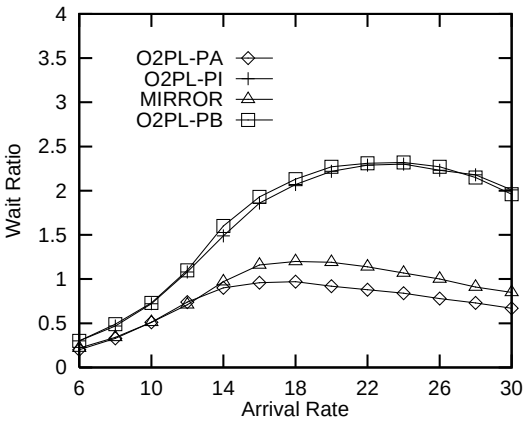
(a) Normal Load



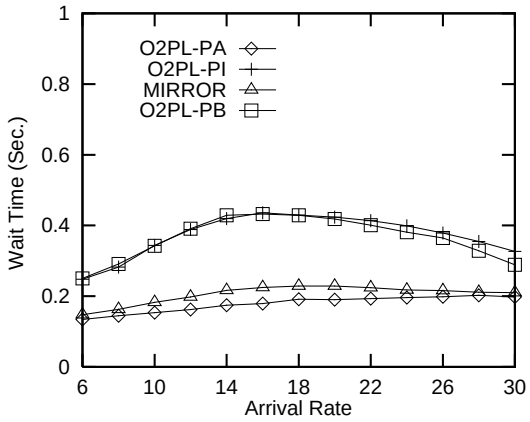
(b) Heavy Load



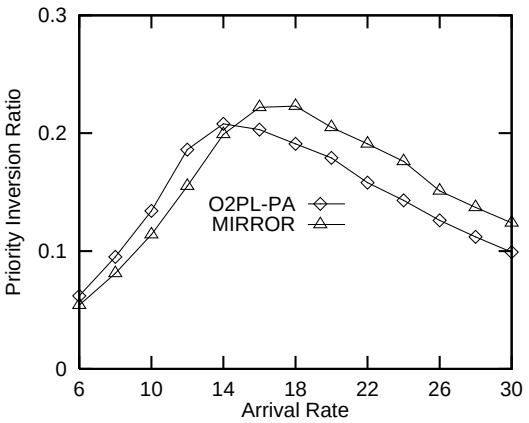
(c) PIR



(d) Wait Ratio

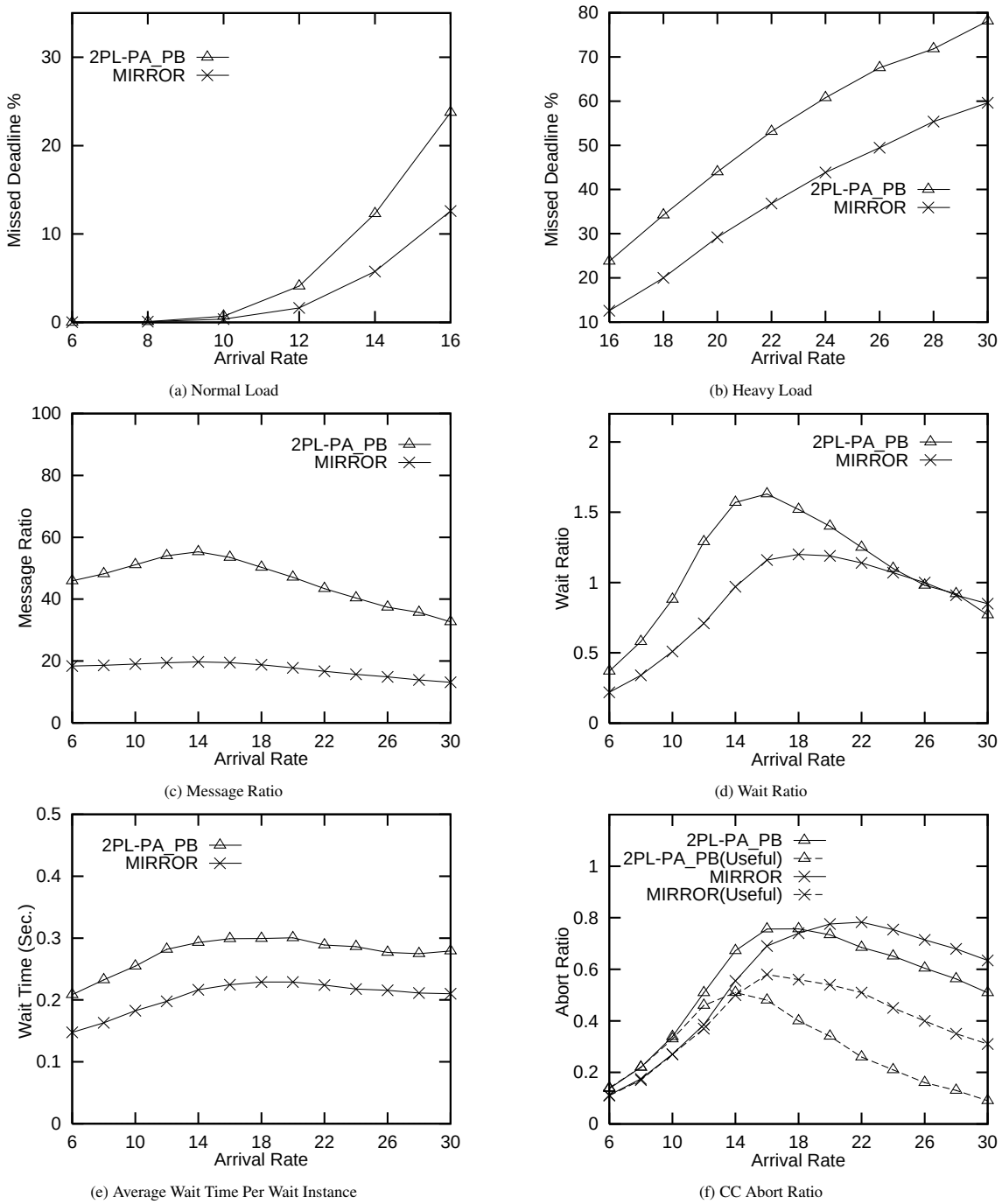


(e) Average Wait Time Per Wait Instance



(f) PIR due to Unprepared Data beyond Demarcation Point

Figure 1. O2PL-based Algorithms



(a) Normal Load

(b) Heavy Load

(c) Message Ratio

(d) Wait Ratio

(e) Average Wait Time Per Wait Instance

(f) CC Abort Ratio

Figure 2. 2PL and O2PL Algorithms