MIRROR: a state-conscious concurrency control protocol for replicated real-time databases☆

Ming Xiong[a,*], Krithi Ramamritham[a,1], Jayant R. Haritsa[b,2], John A. Stankovic[c,3]

[a] *Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA*
[b] *SERC, Indian Institute of Science, Bangalore 560012, India*
[c] *Department of Computer Science, University of Virginia, Charlottesville, VA 22903, USA*

Abstract

Data replication can help database systems meet the stringent temporal constraints of current real-time applications, especially Web-based directory and electronic commerce services. A prerequisite for realizing the benefits of replication, however, is the development of high-performance *concurrency control* mechanisms. In this paper, we present managing isolation in replicated real-time object repositories (MIRROR), a concurrency control protocol specifically designed for firm-deadline applications operating on replicated real-time databases. MIRROR augments the classical O2PL concurrency control protocol with a novel state-based real-time conflict resolution mechanism. 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.

Using a detailed simulation model, 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. Our performance studies show that (a) the relative performance characteristics of these protocols in the real-time environment can 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 simple to implement conflict resolution mechanism works almost as well as more sophisticated strategies.

Keywords: Concurrency control; Data replication; Real-time transaction processing; Two-phase commit; Two-phase locking (2PL); Optimistic concurrency control (OCC); Optimistic two-phase locking (O2PL)

☆Recommended by Patrick O'Neil, Area Editor. A short version of this paper [1] appeared in the 5th IEEE Real-Time Technology and Applications Symposium, Vancouver, Canada, June, 1999.

*Corresponding author. Present address: Bell Labs, Lucent Tech. 600 Mountain Ave., Room 2A-404, Murray Hill, NJ 07974, USA.

E-mail addresses: mxiong@lucent.com (M. Xiong), krithi@cs.umass.edu (K. Ramamritham), haritsa@dsl.serc.iisc.ernet.in (J.R. Haritsa), stankovic@cs.virginia.edu (J.A. Stankovic).

[1] Also affiliated with Indian Institute of Technology, Bombay. Research supported in part by the National Science Foundation Grant IRI-9619588.

[2] Research supported in part by grants from the Department of Science and Technology and the Department of Bio-technology, Government of India.

[3] Research supported in part by the National Science Foundation Grant EIA-9900895.

.

1. Introduction

Many *real-time* database applications are inherently *distributed* in nature. These include the intelligent network services database described in [2], telecom databases [3,4], the mobile telecommunication system discussed in [5], 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* (CC) protocols. While a few isolated efforts in this direction have been made, they have resulted in schemes wherein either the standard notions of database correctness are not fully supported [6,7], the maintenance of multiple historical *versions* of the data is required [8], or the real-time transaction semantics and performance metrics pose practical problems [9]. Further, none of these studies have considered the optimistic two-phase locking (O2PL) protocol [10] although it is the best-performing algorithm in conventional (non-real-time) replicated database systems [10].

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 database systems (RTDBS). One-copy serializability is also adopted by a commercial real-time DBMS, ClustRa [3,4], which has been used in telecom applications. Our study is targeted towards real-time applications with "firm deadlines".[4] 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 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 customer may hang up the phone if the answer to his query is not provided in a timely manner—obviously, there is no value in continuing to process his request after this event. 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 CC protocol called managing isolation in replicated real-time object repositories (MIRROR). MIRROR augments the optimistic two-phase locking (O2PL) algorithm with a novel, simple to implement, state-based data conflict resolution mechanism called *state-conscious priority blocking* (PB). In this scheme, the choice of conflict resolution method is a 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 [11] (the standard mechanism for ensuring distributed transaction atomicity).

Using a detailed simulation model of a distributed real-time database system (DRTDBS), we compare MIRROR's performance against the real-time versions of a representative set of classical replica concurrency control protocols over a range of transaction workloads and system configurations. These protocols include two-phase locking (2PL), optimistic concurrency control (OCC) and O2PL.

The remainder of this paper is organized as follows: in Section 2, we present the transaction execution model and commitment protocol. In Section 3, we present the distributed CC algorithms evaluated in our study. We also develop a

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

practical implementation of the OCC algorithm for replicated data. The options available for real-time conflict resolution are discussed in Section 4. Then, our new MIRROR protocol is presented in Section 5. We describe the replicated RTDBS simulation model in Section 6, and highlight the experimental results in Section 7. Related work is reviewed in Section 8. Finally, we summarize the conclusions of our study in Section 9.

2. Transaction execution and commitment

Before we go on to describe the CC protocols themselves, we first present the distributed transaction execution model and commit protocol adopted in our study.

2.1. Transaction execution model

We follow the common "subtransaction model" [10] in which there is one process, called the *master*, which is executed at the site where the transaction is submitted, and a set of other processes, called *cohorts*, which execute on behalf of the transaction at the various sites that are accessed by the transaction. Cohorts are created by the master sending a STARTWORK message to the local transaction manager at that site. This message includes the work to be done at that site and is passed on to the cohort. Each cohort sends a WORKDONE message to the master after it has completed its assigned data processing work. The master initiates the commit protocol (only) after it has received this message from all its cohorts.

Within the above framework, a transaction may 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 execute concurrently. Both modes of transaction execution are supported in commercial database systems, and we evaluate them in our study as well.

Each cohort that updates a replicated data item has one or more *remote replica update* processes (called *updaters*) associated with it at other sites to update the remote site copies. In particular, a cohort has a remote update process at every site that stores a copy of a data item that it updates. The cohort communicates with its remote update processes for concurrency control purposes, and also sends them copies of the relevant updates. The *time* at which these remote update processes are invoked is a function of the CC protocol, as described later in Section 3.4.

Note that independent of whether the parent transaction's execution is sequential or parallel, the replica updaters associated with a cohort *always execute in parallel*.

2.2. Two-phase commit

We assume that the master implement the classical two-phase commit protocol [11] to maintain transaction atomicity. In this protocol, the master, after receiving the WORKDONE message from all its cohorts, initiates the first phase of the commit protocol by sending PREPARE (to commit) messages in parallel to all its cohorts. Each cohort that is ready to commit first force-writes a `prepare` log record to its local stable storage and then sends a YES vote to the master. At this stage, the cohort has entered a prepared state wherein it cannot unilaterally commit or abort the transaction, but has to wait for the final decision from the master. On the other hand, each cohort that decides to abort force-writes an `abort` log record and sends a NO vote to the master. Since a NO vote acts like a veto, the cohort is permitted to unilaterally abort the transaction without waiting for the decision from the master.

After the master receives votes from all its cohorts, the second phase of the protocol is initiated. If all the votes are YES, the master moves to a committing state by force-writing a `commit` log record and sending COMMIT messages to all its cohorts. Each cohort, upon receiving the COMMIT message, moves to the committing state, force-writes a `commit` log record, and sends an ACK message to the master.

On the other hand, if the master receives one or more NO votes, it moves to the aborting state by force-writing an `abort` log record and sends ABORT messages to those cohorts that are in the prepared state. These cohorts, after receiving the ABORT message, move to the aborting state, force-write an abort log record and send an ACK message to the master.

Finally, the master, after receiving ACKs from all the prepared cohorts, writes an end log record and then "forgets" the transaction (by removing from virtual memory all information associated with the transaction).

3. Distributed CC protocols

In this section, we review three families of distributed CC protocols, namely, 2PL [12], OCC and O2PL [10]. All three protocol classes belong to the "read one copy, write all copies" (ROWA) 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.

3.1. Distributed 2PL

In the distributed 2PL algorithm described in [12], a transaction that intends to read a data item has to only set a read lock on *any* one 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.

3.2. Distributed OCC

Since, to the best of our knowledge, there have been no replicated OCC algorithms presented in the literature, we have devised a new algorithm for this purpose. Our algorithm extends the implementation strategy for centralized OCC algorithms proposed in [13] to handle data distribution and replication.

In OCC, transactions execute in three stages: *read*, *validation*, and *write*. In the read stage, cohorts read and update data items freely, storing their updates into *private workspaces*. Only local data items are accessed and all updating of replicas is deferred to the end of transaction, that is, to the commit processing phase. More specifically, the 2PC protocol is "overloaded" to perform validation in its first phase, and then public installation of the private updates of successfully validated transactions in its second phase.

In the validation stage, the following procedure is followed: 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 the *private workspace* 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 stage (the third stage) of the OCC algorithm. In this write stage, the cohort first installs all the private updates in the public database and then it sends a COMMIT message to all its updaters which then complete their write phase in the same manner as the cohort.

A validation test is performed for each local validation initiated by a cohort or updater. For the implementation of the validation test itself, we employ an efficient strategy called *lock-based distributed validation*, which is described in Appendix A.

An important point to note here is that in contrast to centralized databases where transactions that validate successfully *always* commit, a distributed transaction that is successfully locally validated might be aborted later because it fails during global validation. This can lead to "wasteful" aborts of transactions—a transaction that is locally validated may abort other transactions in this process, as discussed above. If this transaction is itself later aborted during global validation, it means that all the aborts it caused during local validation were unnecessary.

3.3. Distributed O2PL

The O2PL algorithm [10] is 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.[5]

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.[6]

3.4. Scheduling 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 during the *commit processing* (more precisely, during the *first phase* of *commit processing*).

4. Real-time conflict resolution mechanisms

We now move on to discussing the integration of real-time data conflict resolution mechanisms with the replica concurrency control protocols of the previous section. In particular, we consider three different ways—*PB*, *Priority Abort* (PA), and *Priority Inheritance* (PI)—to introduce real-time priorities into the locking protocols, while for the optimistic protocol, we utilize *Priority Wait* (PW). These various mechanisms are described in the remainder of this section.

4.1. Priority blocking (PB)

The PB mechanism is similar to the conventional locking protocols in that a transaction is always blocked when it encounters a lock conflict and can obtain the lock only after the lock is released. The *lock request queue*, however, is ordered by transaction priority.

[5] 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.

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

4.2. Priority abort (PA)

The PA scheme [14] attempts to resolve all data conflicts in favor of high-priority transactions. Specifically, at the time of a data lock conflict, if a lock holding cohort or updater has higher priority than the priority of a cohort or updater that is requesting the lock, the requester is blocked. Otherwise, the lock holder is aborted and the lock is granted to the requester. Upon the abort of a cohort (updater), a message is sent to its master (cohort) to abort and then restart the whole transaction (if its deadline has not already 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.

4.3. Priority inheritance (PI)

In the PI scheme [15], 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.

4.4. Priority wait (PW)

In the PW mechanism [16], which is used in conjunction with the OCC protocol, a transaction that reaches 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.

5. The MIRROR protocol

Our new replica concurrency control protocol, MIRROR, augments the O2PL protocol described in Section 3 with a novel, and simple to implement, state-based conflict resolution mechanism called *state-conscious PB*. 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 2PC protocol.

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 transaction are wasted [17]. 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 PA should be used in the early stages of transaction execution, whereas 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 PB (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.

The demarcation points of a cohort/updater T_i are assigned 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.

5.1. Choice of demarcation points

We now explain the rationale behind the above assignment of demarcation points. Essentially, we wish 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. 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 the case of the classical 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. Thus, in state-conscious protocols, cohorts or updaters reach demarcation points only after the start of the 2PC protocol. 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 3.3.

5.2. 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 because PB is applied based on its local cohort or updater state.

5.3. Incorporating PA_PB with 2PL

The PA_PB conflict resolution mechanism, which we discussed above in the context of the O2PL-based MIRROR protocol, can be also integrated with the distributed 2PL protocol, as described below.

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.

A special effect in combining PA_PB 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 the low-priority transaction has already acquired all the locks.

5.4. The PI alternative

In the above description, PB was used as the post-demarcation conflict resolution mechanism. Alternatively, we could use PI instead, as given below:

State-conscious PI (*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

Table 1
Simulation model parameters and default values

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
TransType	Trans. Type (Parallel/Sequential)	Sequential
ArrivalRate	Transaction arrival rate (Trans./s)	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 access time per page	20 ms
LogDisk	Log force time	5 ms
MsgCPU	CPU message send/receive time	1 ms

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 7.10, 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.

6. Simulation model, metrics, and settings

To evaluate the performance of the CC protocols described in Section 3, we developed a detailed simulation model of a distributed real-time database system. Our model is based on the distributed database model presented in [10], which has also been used in several other studies (for example, [18,19]) of distributed database system behavior, and the real-time processing model of [20]. A summary of the parameters used in the simulation model are presented in Table 1.

6.1. Simulation model

The database is modeled as a total collection of *DBSize* pages[7] 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*

[7] This is the basic set of database pages without replication.

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 according to a Poisson process 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. This is determined by parameter *TransType*. 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 both sequential and parallel transactions in this study. Note, however, that replica updaters belonging to the same cohort *always execute 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 (including CPUs and disks) at all sites that the transaction requires for its execution *in the absence of data replication*. This is computed in this manner 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 [21], 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 [22,23] show that the frequency of deadlocks is extremely small—therefore, a low-overhead solution like timeout is preferable to more expensive graph-based techniques.

6.2. Performance metric

The performance metric employed is *MissPercent*, the *percentage of transactions that miss their deadlines*. MissPercent values in the range of 0–30 percent are taken to represent system performance under "normal" loads, while MissPercent values in the range of 30–100 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;[8] 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.

6.3. Parameter settings

Table 1 presents the settings 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 replica updaters belonging to the same cohort always execute in parallel). The parameter values for CPU, disk and message processing times are similar to those in [10]. While these times have certainly decreased 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 less than 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, [9,24]), here the database size represents only the "hot spots", that is, the heavily accessed data of practical applications, and not the entire database.

7. Experiments and results

Using the firm-deadline DRTDBS model described in the previous section, we conducted an extensive set of simulation experiments comparing the real-time performance of the various replica CC protocols. In this section, we present the results of a representative set of experiments.

We first analyze the performance of the different conflict-resolution mechanisms for the locking protocols and, in particular, determine whether MIRROR can provide better performance as compared to the other O2PL variations. Then, we move on to comparing the relative performance of the locking and optimistic protocols. We explore the impacts of resource contention (I/O and CPU contention), data contention, and time contention (deadline distribution) in our experiments. In addition, we also evaluate locking and OCC-based algorithms for both *sequential* and *parallel* transactions. Finally, we compare the performance of MIRROR and its alternative—O2PL–PA_PI.

In the experiments, 90 percent confidence intervals have been obtained whose widths are less than ±10% of the point estimate for the Missed Deadline Percentage (MissPercent). Only statistically significant differences are discussed here.

7.1. Expt. 1: O2PL-based conflict resolution

Our goal in this experiment was to investigate the performance of the various conflict resolution mechanisms (PB, PA, PI and PA_PB) when integrated with the O2PL CC protocols.

For this experiment, Figs. 1 and 2 present the MissPercent 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 CC, 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.

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

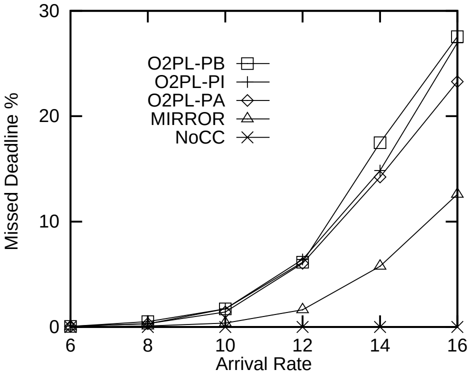

Fig. 1. O2PL algorithms MissPercent with normal load.

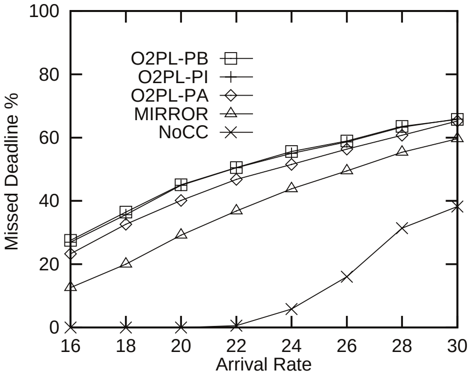

Fig. 2. O2PL algorithms MissPercent with heavy load.

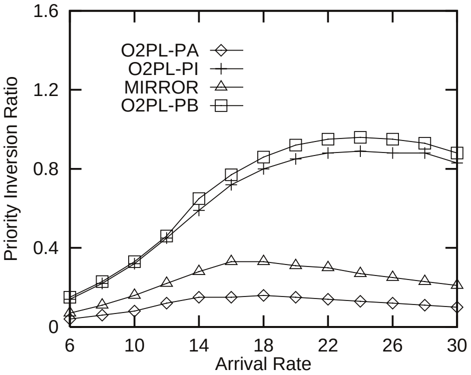

Fig. 3. Priority inversion ratio.

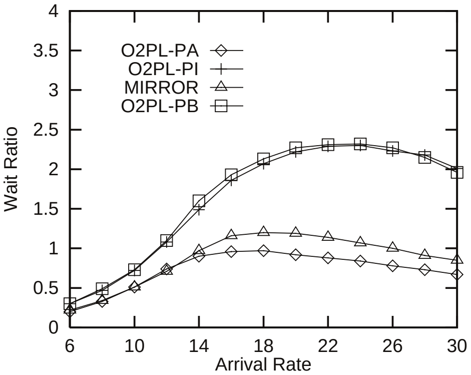

Fig. 4. Wait ratio.

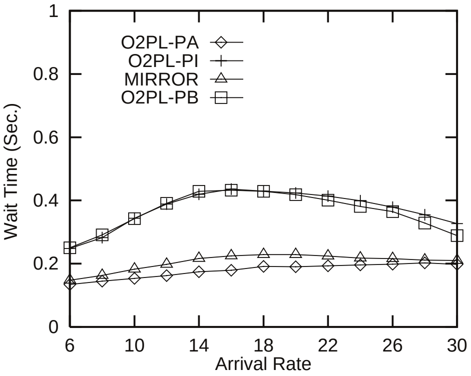

Fig. 5. Average wait time per wait instance.

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 s, 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 Figs. 3–5 which present the priority inversion ratio, the wait ratio 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 Fig. 3—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 O2PL–PI and O2PL–PB perform basically the same with O2PL–PI being slightly better than O2PL–PB under normal load. 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 Figs. 3–5, where we observe that O2PL–PI and O2PL–PB have similar PIR, wait ratio and wait time statistics.

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 Fig. 6, 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.

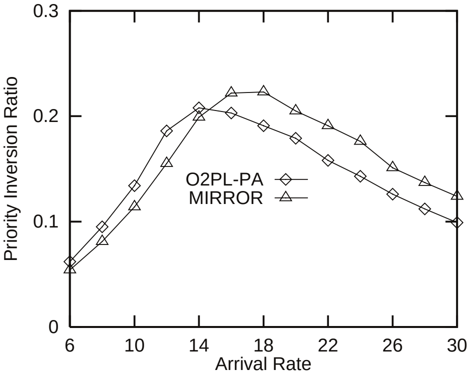

Fig. 6. PIR due to unprepared data beyond demarcation point.

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

7.2. Expt. 2: baseline–CC 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 Exp. 1. The MissPercent of transactions is presented in Figs. 7 and 8 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 per cent (absolute) at an arrival rate of 14 transactions/s. 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

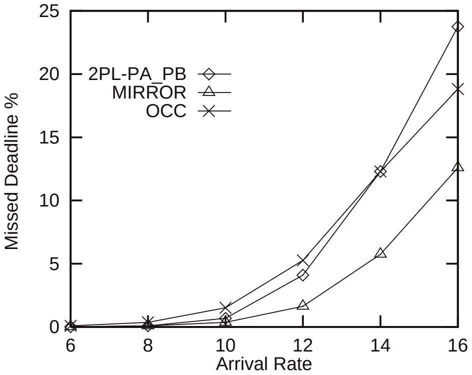

Fig. 7. CC algorithms MissPercent with normal load.

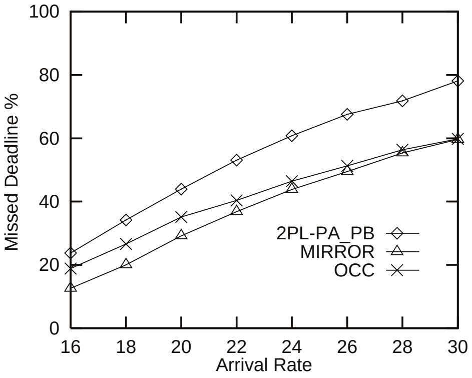

Fig. 8. CC algorithms MissPercent with heavy load.

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 a 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 exhibits fewer 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/s. 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 best performance over a wide workload range, even slightly outperforming 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.[9] As observed in the previous studies of centralized RTDB settings [16], 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, MIRROR outperforms OCC and 2PL–PA_PB in the tested workloads.

7.3. Expt. 3: varying update frequency

So far we observed that MIRROR outperforms OCC and 2PL–PA_PB under a certain update frequency. The next experiment investigates the performance of these algorithms under different update frequencies. For this experiment, Fig. 9 presents the MissPercent when the update frequencies are low and moderate for an arrival rate of 14 transactions/s. It should be noted that data is normally replicated in distributed database systems only when the update frequency is not

[9] A transaction's progress is measured in terms of the CPU and disk services it has received.

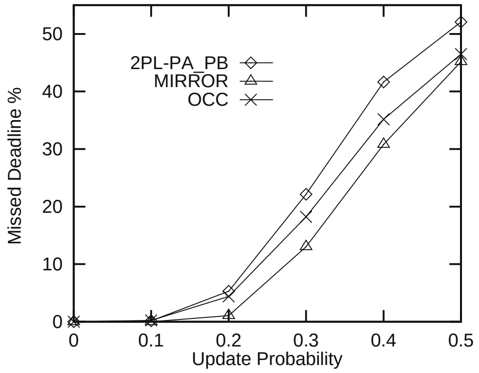

Fig. 9. MissPercent with low *UpdateFreq*.

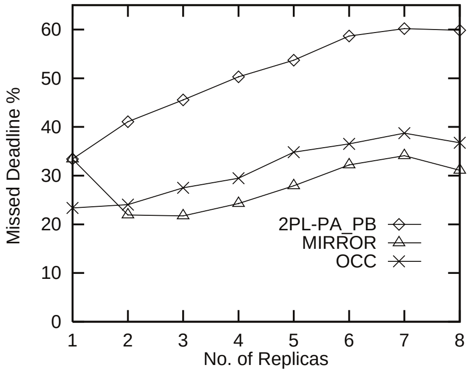

Fig. 10. MissPercent with partial replication ($DBSize = 800, NumSites = 8$).

very high. Therefore, the update frequency results that we present here can aid in understanding the tradeoffs of different protocols in replicated RTDBS.

When the update frequency is varied from low to moderate, we observe that the qualitative behavior of the various algorithms is similar to that of Expt. 1. However, we also observe in Fig. 9 that the performance of MIRROR degrades more drastically with the increase of update frequency. For example, MIRROR performs only slightly better than OCC when the update frequency is 0.5. The reason for the degraded performance of MIRROR is that with higher 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 7.2, and more aborts are wasted under MIRROR.

In summary, for low to moderate update frequencies, MIRROR is the preferred protocol.

7.4. Expt. 4: partial replication

Previous experiments were conducted when data are fully replicated. The purpose of this experiment is to examine the impact of replication level on the three algorithms: MIRROR, OCC and 2PL–PA_PB. For this experiment, *NumSites* and *DBSize* are fixed at 8 and 800, respectively, while *NumCPUs* and *NumDataDisks* per site are set to 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 values are the same as those given in Table 1. For this environment, Fig. 10 presents the MissPercent 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/s.

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 in the absence of replication under tested workloads.

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 by the abort curves shown in Fig. 11 (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 under MIRROR is significantly reduced as number of replicas increases. This helps reduce the resource wastage in MIRROR.

In Fig. 10, we also observe that the performance of MIRROR is initially improved, then it is degraded as the number of replicas is increased.[10]

[10] The same behavior is also observed for the performance of OCC when the update frequency is lower.

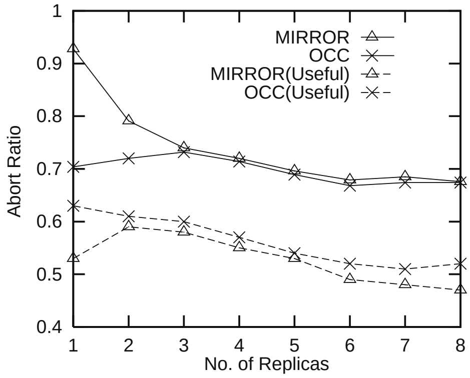

Fig. 11. Abort ratio with partial replication ($DBSize = 800$, $NumSites = 8$).

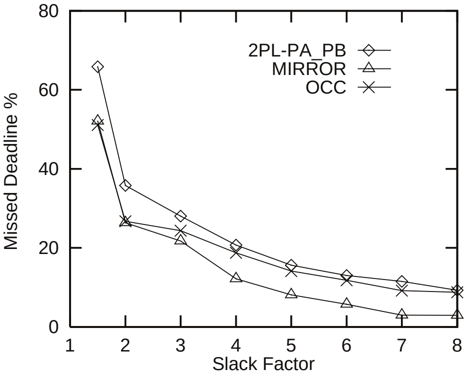

Fig. 12. MissPercent with slack factor.

In O2PL, read operations can benefit from local data when data is replicated. However, as the data replication level goes up, update operations suffer due to updates to remote data copies. Hence, the performance degrades after a certain replication level. For example, the performance of MIRROR is improved when the number of replicas is from 1 to 3, then it is degraded as the number of replicas is larger than 3. 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. Such performance degradation of OCC and 2PL is also observed in conventional replicated databases [25].

7.5. Expt. 5: varying slack factor

In this section, we study the behavior of MIRROR, OCC and 2PL–PA_PB as we vary the slack factor, a real-time parameter. Fig. 12 presents the MissPercent when the slack factor is varied from 1.5 to 8 while the arrival rate is fixed at 14 transactions/s. The other parameters take default values as in Table 1. We observe that MissPercent of all algorithms increases rapidly at low slack factors. As the slack factor decreases, transactions are given less time to complete, and the missed deadline percentage of all algorithms increases. With moderate to high slack factors, the relative performance of OCC, 2PL–PA_PB and MIRROR remains to be the same as in previous experiments. As the slack factor increases, transactions are given more time to complete, and the missed deadline percentage of all algorithms decreases. However, after the slack factor increases beyond a value, the MissPercent stays almost constant. For example, in the case of MIRROR, the missed deadline percentage stays almost constant beyond a slack factor of 6. Since increasing the slack factor provides transactions with more time to complete, it results in a higher restart ratio of transactions and more transactions running concurrently in the system. Eventually, resources in the system are saturated and MissPercent becomes stable for a fixed arrival rate.

7.6. Expt. 6: varying data access ratio

Our next experiment investigates the impact that data access ratio (DAR) has on the performance of 2PL, O2PL and OCC-based algorithms by varying the database size. DAR is defined to be the maximum number of pages that can be simultaneously accessed by all the transactions in the system relative to the size of the database [16]. It has been shown that DAR has an impact on the qualitative performance of CC algorithms [16]. The parameter values are identical to Expt. 4 except that the number of replicas is fixed at 3. For graph clarity, we focus on OCC, MIRROR and O2PL–PA. Here, we choose O2PL–PA because it is observed in a centralized database setting [16]

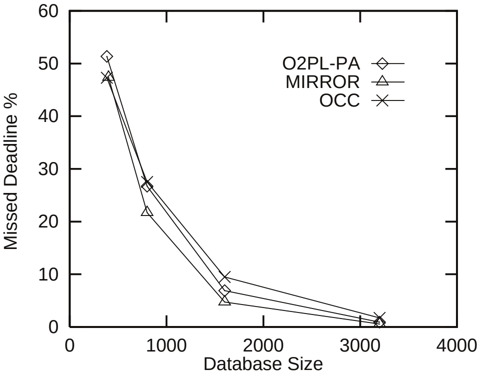

Fig. 13. MissPercent with *DBSize* (*ReplDegree* = 3).

that there is a performance crossover of 2PL–PA and OCC while the database size is varied. In Fig. 13, we also observe a performance crossover of O2PL–PA and OCC while the database size is increased. We actually observe that MIRROR is always the best choice under the tested database sizes. The same performance is also observed in our study by varying the level of data replication.

7.7. Expt. 7: varying message cost

Our next experiment investigates the impact of an increase of message cost on CC algorithms. So far we assumed a low message cost of $MsgCPU =$ 1 ms. Fig. 14 presents the missed deadline percentage when $MsgCPU$ is 5 ms. As we can observe from the figure, the performance of 2PL–PA_PB degrades drastically due to the increase of message cost. Both OCC and MIRROR constantly outperform 2PL–PA_PB throughout the workload range. This, however, is due to the much larger number of messages transmitted under 2PL–PA_PB. This is clearly brought out in Fig. 15. The increased message costs significantly aggravate the CPU contention of 2PL–PA_PB, thereby drastically degrading 2PL–PA_PB's performance. It is also observed that the increase of message cost does not change the qualitative performance of MIRROR and OCC.

We have also done experiments with different $MsgCPU$ values and the results in these other experiments were similar.

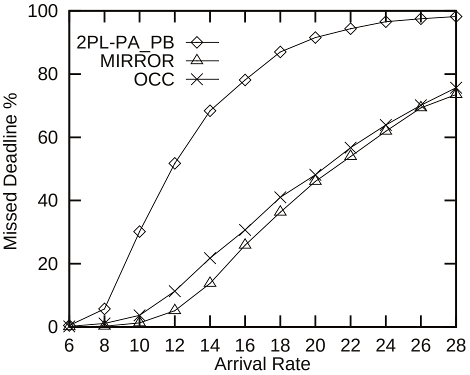

Fig. 14. MissPercent with MsgCPU = 5 ms.

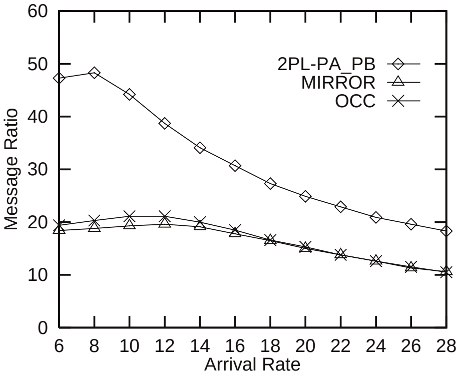

Fig. 15. Message ratio with MsgCPU = 5 ms.

7.8. Expt. 8: the impact of buffering

The purpose of this experiment is to examine the impact of buffering on the performance of 2PL–PA_PB, MIRROR and OCC. To this end, Fig. 16 presents the missed deadline percentage when *BufHitRatio* is set to 80 percent. Other parameters are identical to the previous experiment (Expt. 7). We found that the main impact of having an 80 percent *BufHitRatio* is to make the system heavily CPU-bound. The qualitative performance of 2PL–PA_PB, MIRROR and OCC is not changed as compared to the previous experiment. However, we also observe a difference. Compared to MIRROR and OCC, 2PL–PA_PB in Fig. 16 begins to miss deadlines at a lower arrival rate than in Fig. 14. For example, in Fig. 16, 2PL–PA_PB begins to miss deadlines when the transac-

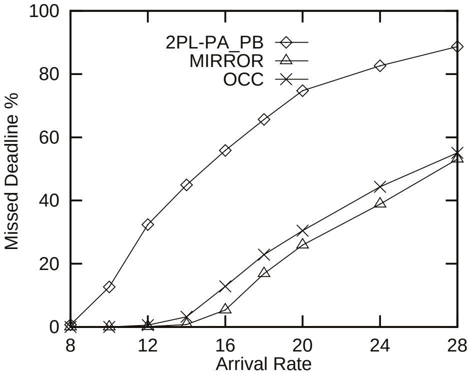

Fig. 16. MissPercent with BufHitRatio = 0.8.

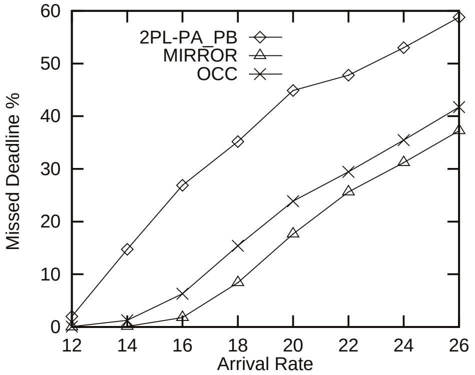

Fig. 17. MissPercent of parallel execution.

tion arrival rate is greater than 8 transactions/s, whereas OCC begins to miss deadlines when the transaction arrival rate is greater than 12 transactions/s. On the other hand, in Fig. 14, 2PL–PA_PB begins to miss deadlines when the transaction arrival rate is greater than 6 transactions/s, whereas OCC begins to miss deadlines when the transaction arrival rate is greater than 8 transactions/s. Compared to OCC and MIRROR, the behavior of 2PL–PA_PB is relatively worse with a higher buffer hit ratio. This is because 2PL–PA_PB aggravates CPU contention and the CPU is the primary bottleneck when the buffer hit ratio is high.

7.9. *Expt. 9: parallel execution of cohorts*

In all of the previous experiments, the cohorts of each transaction executed in *sequence*. We also conducted similar experiments for transaction workloads with *parallel* cohort execution and we report on those results here. Due to similarity of the experiments and results, we only illustrate one set of performance results. The goal of this experiment is to investigate the performance of MIRROR compared to OCC and 2PL–PA_PB.

In this experiment, all parameters are set identical to those in the previous experiment (Expt. 8) except that the *NumSites* and *DBSize* are again fixed at 8 and 800, respectively, while the number of replicas (*ReplDegree*) is fixed at 3 to model a partially replicated database. These changes were made to provide a system operational region of interest without having to model very high transaction arrival rates. The execution sites for a transaction's cohorts are determined in the following manner: If a page is present at the originating site of the transaction, use the local copy; otherwise, choose uniformly from among the sites that have remote copies. In the latter case, a site is favored if it has already been chosen by the same transaction. Fig. 17 presents the missed deadline percentage of MIRROR, OCC and 2PL–PA_PB when cohorts of a transaction are executed in *parallel*. It can be observed that MIRROR still outperforms OCC in parallel execution of cohorts. Further, both MIRROR and OCC outperform 2PL–PA_PB.

7.10. *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, could performance be further improved by replacing the PA_PB mechanism with the PA_PI mechanism described in Section 5, wherein PI is used for conflict resolution after the demarcation point, result in *even better* performance? This expectation is because, as mentioned earlier in Section 4, PI seems capable of providing earlier termination of the (priority-inversion) blocking condition than PB.

We conducted experiments to evaluate the above possibility and results are shown in Fig. 18. We 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 PI in the distributed environment involves excessive message costs and dissemination delay, thereby neutralizing its positive points. Further, PI comes into play only when a high-priority transaction is blocked by a low-priority transaction in commit phase. Due to message propagation delay in PI, it is too late for PI to be effective. The similar observation is also observed in [18].

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

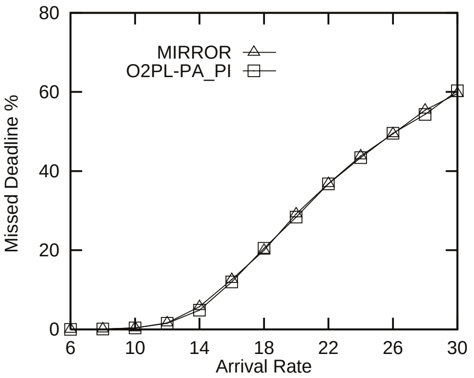

Fig. 18. MissPercent of MIRROR and O2PL–PA_PI.

7.11. Summary

Apart from the experiments described above, we have conducted a variety of experiments that cover a range of workloads and system configurations. Table 2 summarizes these results under both tight and loose slack factors with low to moderate update frequencies: In the table, system parameters, i.e., load, message cost and data access ratio (DAR) 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.

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 at low to moderate update frequencies.
3. OCC achieves better performance than most of the O2PL-based and 2PL-based algorithms, except for MIRROR, when the update frequencies are low to moderate in value.
4. Protocols integrated with only PB or PI (e.g., O2PL–PB, O2PL–PI) do not perform very well. Thus, they are not suited to distributed real-

Table 2
Performance of algorithms

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

time databases. A similar poor performance of these mechanisms has also been observed earlier for centralized real-time databases [16].

5. MIRROR (O2PL–PA_PB) performs best for low to moderate update frequencies. Since, we expect most replicated RTDBS applications will belong to the category of low to moderate update frequencies, MIRROR appears to be the best overall choice for implementation in these systems.

8. Related work

CC algorithms and real-time conflict resolution mechanisms for RTDBS have been studied extensively (e.g. [9,13,16,17]). However, CC for replicated DRTDBS has only been studied in [6–9]. An algorithm for maintaining consistency and improving the performance of replicated DRTDBS is proposed in [8]. 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 [6,7]. These algorithms employ Epsilon-serializability (ESR) [26] 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 CC protocols.

The performance of the classical distributed 2PL locking protocol (augmented with the PA and PI conflict resolution mechanisms) and of OCC algorithms in replicated DRTDBS was studied in [9] for real-time applications with “soft” deadlines. [11] 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 include an investigation of the O2PL algorithm which has not been studied before in the real-time context. We demonstrated that MIRROR, which is O2PL enhanced with PA_PB, performs best in most situations.

In [17], a *conditional PI* mechanism is proposed to handle priority inversion. This mechanism capitalizes on the advantages of both PA and PI in real-time data conflict resolution. It outperforms both PA and PI 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 PB* and *state-conscious PI* protocols resolve real-time data conflicts based on the *states* of transactions rather than their lengths.

9. Conclusions

In this paper, we have addressed the problem of accessing replicated data in DRTDBS where transactions have firm deadlines, a framework under which many current real-time 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, O2PL and OCC-

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

based CC algorithms. Our performance studies show the following:

1. The relative performance characteristics of replica CC 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 RTDBS.
2. The MIRROR protocol provides the best performance in both fully and partially replicated environments for real-time applications with low or moderate update frequencies. Given that most of the distributed real-time applications that we are aware of fall into this category, MIRROR appears to be an attractive choice for designers of replicated RTDBS.
3. As mentioned earlier, MIRROR implements a state-conscious PB-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 PI*. 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.

In summary, our study indicates that MIRROR is simple, practical, and efficient, making it an attractive candidate for high-performance replicated DRTDBS.

Appendix A. Distributed validation using locks

The validation process can be carried out in either of two ways: *backward validation* or *forward validation* [13,16]. In real-time databases, to provide flexibility for priority-based conflict resolution, a transaction should be validated against active transactions instead of committed ones, i.e., forward validation is preferable.

As in [13], a *lock-based* implementation strategy for OCC is used—the lock types are *read-phase lock* (R-lock) and *validation-phase lock* (V-lock). An R-lock for a data item is set by a transaction in its read phase while a V-lock is set by a transaction only in its validation phase. An R-lock is compatible with another R-lock, but an R-lock is incompatible with a V-lock and a V-lock is incompatible with another V-lock. The pseudocode of the OCC algorithm which is derived from [13] is given below (critical sections are bracketed by “” and “”):

```
• Read phase of a cohort/updater of transaction $T_i$:
    for every data object to be read or written do
        { place it in ReadSet($T_i$) or WriteSet($T_i$);
         <set an R-lock;> }
• Validation phase of a cohort/updater of transaction $T_i$:
    Valid := TRUE;
    <for every data object in WriteSet($T_i$) do
        { release $T_i$'s R-lock on the data object;
         if another transaction has R-locked it
              then Valid:=FALSE & request a V-lock;
              else set a V-lock; }
    if not Valid
        then invoke real-time conflict resolution;>
```

If a validating transaction attempts to hold a V-lock on an object currently held by R-lock(s), then a real-time conflict resolution mechanism from [16] is invoked to determine which lock(s) should prevail. In the centralized OCC algorithm, R-locks of a transaction can be released when it gets validated. However, in a distributed system all the R-locks of a cohort/updater must be held until commit time, i.e., when a cohort/updater receives the COMMIT message from its parent. Release of R-locks when a cohort/updater gets validated but before the commit time may result in the violation of serializability. All the V-locks must be held until the write phase is finished.

References

[1] M. Xiong, K. Ramamritham, J. Haritsa, J. Stankovic, MIRROR: a state-conscious concurrency control protocol

in replicated real-time databases, Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium, Vancouver, Canada, June, 1999.

[2] B. Purimetla, R. Sivasankaran, J.A. Stankovic, K. Ramamritham, A study of distributed real-time active database applications, IEEE Workshop on Parallel and Distributed Real-Time Systems, Newport Beach, California, 1993.

[3] O. Torbjornsen, S.O. Hvasshovd, Y.K. Kim, Towards real-time performance in a scalable, continuously available telecom DBMS, in: Proceedings of the First International Workshop on Real-Time Databases, Newport Beach, CA, April, 1996.

[4] The ClustRa White Paper, http://www.clustra.com.

[5] Y. Yoon, Transaction scheduling and commit processing for real-time distributed database systems, Ph.D. Thesis, Korea Adv. Inst. of Science and Technology, May, 1994.

[6] S. Son, S. Kouloumbis, A real-time synchronization scheme for replicated data in distributed database systems, Inform. Systems 18 (6) (1993) 375–389.

[7] S. Son, F. Zhang, Real-time replication control for distributed database systems: algorithms and their performance. Fourth International Conference on Database Systems for Advanced Applications, Singapore, April, 1995.

[8] S. Son, Using replication for high performance database support in distributed real-time systems, Proceedings of the Eighth IEEE Real-Time Systems Symposium, San Jose, CA, 1987, pp. 79–86.

[9] O. Ulusoy, Processing real-time transactions in a replicated database system, Distributed Parallel Databases 2 (1994) 405–436.

[10] M. Carey, M. Livny, Conflict detection tradeoffs for replicated data, ACM Trans. Database Systems 16 (1991) 703–746.

[11] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, Los Altos, CA, 1992.

[12] J. Gray, Notes on database operating systems, in: R. Bayer, R. Graham, G. Seegmuller (Eds.), Operating Systems: An Advanced Course, Springer, Berlin, 1979.

[13] J. Huang, J.A., Stankovic, K. Ramamritham, D. Towsley, Experimental evaluation of real-time optimistic concurrency control schemes, Proceedings of the 17th International Conference on Very Large Data Bases, Barcelona, September, 1991.

[14] R. Abbott, H. Garcia-Molina, Scheduling real-time transactions: a performance evaluation, Proceedings of the 14th VLDB Conference, Los Angeles, CA, 1988.

[15] L. Sha, R. Rajkumar, J.P. Lehoczky, Priority inheritance protocols: an approach to real-time synchronization, IEEE Trans. Comput. 39 (1990) 1175–1185.

[16] J.R. Haritsa, M. Carey, M. Livny, Data access scheduling in firm real-time database systems, J. Real-Time Systems 4 (1992) 203–241.

[17] J. Huang, J.A. Stankovic, K. Ramamritham, D. Towsley, B. Purimetla, Priority inheritance in soft real-time databases, J. Real-Time Systems 4 (1992) 243–268.

[18] J. Haritsa, K. Ramamritham, R. Gupta, The PROMPT real-time commit protocol, IEEE Trans. Parallel Distributed Systems 11 (2) (2000) 160–181.

[19] K.Y. Lam, Concurrency control in distributed real-time database systems, Ph.D. Dissertation, City University of Hong Kong, October, 1994.

[20] M. Xiong, R. Sivasankaran, J.A. Stankovic, K. Ramamritham, D. Towsley, Scheduling transactions with temporal constraints: exploiting data semantics, Proceedings of the 17th IEEE Real-Time Systems Symposium, Washington, DC, December 1996, pp. 240–251.

[21] C. Liu, J. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, J. ACM 20 (1) (1973) 46–61.

[22] R. Agrawal, M. Carey, L. McVoy, The performance of alternative strategies for dealing with deadlocks in database management systems, IEEE Trans. Software Eng. 13 (12) (1987) 1348–1363.

[23] J. Gray, P. Homan, R. Obermarck, H. Korth, A strawman analysis of probability of waiting and deadlock in a database system, IBM Res. Rep. RJ 3066, San Jose, CA.

[24] T. Anderson, Y. Breitbart, H. Korth, A. Wool, Replication, consistency, and practicality: are these mutually exclusive? Proceedings of the ACM-SIGMOD 1998 International Conference on Management of Data, Seattle, WA, 1998, pp. 484–495.

[25] B. Ciciani, D.M., Dias, P.S., Yu, Analysis of replication in distributed database systems, IEEE Trans. Knowledge Data Eng. 2 (2) (1990) 247–261.

[26] K.L. Wu, P.S. Yu, C. Pu, Divergence control for epsilon-serializability, in: Proceedings of Eighth International Conference on Data Engineering, Phoenix, February, 1992.