Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment

Sang H. Son* and Chun-Hyon Chang**

* Department of Computer Science
University of Virginia
Charlottesville, VA 22903, USA

** Department of Computer Science
Kon-Kuk University
Seoul, Korea

ABSTRACT

Real-time systems must maintain data consistency while minimizing the number of tasks that miss the deadline. To satisfy both the consistency and real-time constraints, there is the need to integrate synchronization protocols with real-time priority scheduling protocols. In this paper, we address the problem of priority scheduling in real-time database systems. We first present a prototyping environment for investigating distributed software. Specific priority-based real-time locking protocols are then discussed, together with a performance study which illustrates the use of the prototyping environment for evaluation of synchronization protocols for real-time database systems.

1. Introduction

The growing importance of real-time computing in a wide range of applications such as aerospace and defense systems, industrial automation, and nuclear reactor control, has resulted in an increased research effort in this area. Distributed systems greatly exacerbate the difficulty of developing real-time systems as delays associated with interprocess communications and remote database accesses must be taken into account [Wat88]. Researchers working on developing real-time systems based on distributed system architecture have found out that database managers are assuming much greater importance in real-time systems. In the recent workshops sponsored by the Office of Naval Research [IEEE89, ONR89], developers of real-time systems pointed to the need for basic research in database systems that satisfy timing constraint requirements in collecting, updating, and retrieving shared data. Further evidence of its importance is the recent growth of research in this field [Shin87, Son88].

Compared with traditional databases, real-time database systems have a distinct feature: they must satisfy the timing constraints associated with transactions. In other words, "time" is one of the key factors to be considered in real-time database systems. The timing constraints of a transaction typically include its ready time and deadline, as well as temporal consistency of the data accessed by it. Transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query on a tracking data of a missile must be processed within the given deadlines; otherwise, the information provided could be of little value. In such a system, transaction processing must satisfy not only the database consistency constraints but also the timing constraints.

In addition to providing shared data access capabilities, distributed real-time database systems offer a means of loosely coupling communicating processes, making it easier to rapidly update software, at least from a functional perspective. However, with respect to time-driven scheduling and system timing predictability, they present new problems. One of the characteristics of current database managers is that they do not schedule their transactions to meet response time requirements and they commonly lock data tables to assure database consistency. Locks and time-driven scheduling are basically incompatible. Low priority transactions holding locks required by higher priority transactions can and will block the higher priority transactions, leading to response requirement failures. New techniques are required to manage data consistency which are compatible with time-driven scheduling.

One of the primary reasons for the difficulty in successfully developing and evaluating new database management techniques suitable for real-time applications is that it takes a long time to develop a system, and evaluation is complicated because it involves a large number of system parameters that may change dynamically. For example, although new approaches for synchronization and database recovery have been developed recently [Son88, Son89], experimentation to verify their properties and to evaluate their performance has not been performed due to the lack of appropriate test tools.

A prototyping technique can be applied effectively to the evaluation of database management techniques for distributed real-time systems. A database prototyping environment is a software package that supports the evaluation of database management techniques in an environment other than that of the target database system. The advantages of such an environment are obvious [Son90]. Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation, especially for distributed real-time systems.
time database systems. Recently, simulators have been developed for investigating performance of several priority-based concurrency control algorithms for real-time applications [Abb88, Abb89, Raj89]. However, they do not provide a module hierarchy composed from reusable components as in our prototyping environment. Software developed in our prototyping environment will execute in a given target machine without modification of any layer except the hardware interface. In the operating system area, the ARTS real-time kernel and its toolset, being developed at Carnegie-Melon University, attempts to provide a "predictable, analyzable, and reliable distributed real-time computing environment" which is an excellent foundation for a real-time system [Tok89]. It implements different prioritized and non-prioritized scheduling algorithms and prioritized message passing. The major difference between our prototyping environment from ARTS is that ours is portable since it is implemented in a host environment, and our environment can support a spectrum of distributed database system functions without much overhead.

This paper presents a database prototyping environment that supports evaluation of distributed real-time database systems. To illustrate its usefulness, a series of experimentation to evaluate priority-based real-time locking protocols has been performed.

One of the major problems in priority-based locking protocols is that, owing to the effect of blocking, a condition known as unbounded priority inversion, where a higher priority task is blocked by lower priority tasks for an indefinite period of time. To address this problem, the priority ceiling protocol was proposed in [Sha88]. It tries to achieve not only minimizing the blocking time of a transaction to at most one lower priority transaction execution time, but also preventing the formation of deadlocks. In this paper, we investigate the performance of the priority ceiling protocol and compare it with other synchronization protocols. We also discuss the performance of real-time locking protocols in distributed database environments.

The rest of the paper is organized as follows. Section 2 presents the design principles and the current implementation of the prototyping environment. Section 3 discusses priority-based real-time locking protocols and presents their experimental performance results using the prototyping environment. Section 4 presents two different priority-based locking protocols and their performance in distributed environments. Section 5 is the conclusion.

2. Structure of the Prototyping Environment

A prototyping environment, if properly structured, can reduce the time for evaluating new technologies and design alternatives. From our past experience, we assume that a relatively small portion of a typical database system's code is affected by changes in specific control mechanisms, while the majority of code deals with intrinsic problems, such as file management. Thus, by properly isolating technology-dependent portions of a database system using modular programming techniques, we can implement and evaluate design alternatives very rapidly. For this reason, our prototyping environment is designed as a modular, message-passing system to support easy extensions and modifications. Server processes can be created, relocated, and new implementations of server processes can be dynamically substituted. It provides a library of modules with different performance and reliability characteristics for an operating system as well as database management functions [Son88c, Son90]. Operating system facilities are included in the library because the correct functioning and timing behavior of database control algorithms depends on the appropriate support of the underlying operating system. For experimentation, the module library facilitates the construction of multiple system instances customized according to application-dependent requirements without much overhead.

The prototyping environment provides support for transaction processing, including transparency to concurrent access, data distribution, and atomicity. An instance of the prototyping environment can manage any number of virtual sites specified by the user. Modules that implement transaction processing are decomposed into several server processes, and they communicate among themselves through ports. The clean interface between server processes simplifies incorporating new algorithms and facilities into the prototyping environment, or testing alternate implementations of algorithms. A separate process for each transaction is created for concurrent execution of transactions.

Figure 1 illustrates the structure of the prototyping environment. The prototyping environment is based on a concurrent programming kernel, called the StarLite kernel, which supports process control to create, ready, block, and terminate processes. Based on the StarLite kernel, the environment consists of the modules for user interface, configuration management and transaction generation, transaction manager, message server, resource manager, and performance monitor.

User Interface (UI) is a front-end invoked when the prototyping environment begins. UI is menu-driven, and designed to be flexible in allowing users to experiment various configurations with different system parameters. A user can specify the following:

- system configuration: number of sites and topology, and the relative speed of CPU, I/O, and communication cost.

![Fig. 1. Structure of the prototyping environment](image-url)
database configuration: database at each site with user defined structure, size, granularity, and levels of replication.

* load characteristics: number of transactions to be executed, size of their read-sets and write-sets, transaction types (read-only/update and periodic/aperiodic) and their priorities, and the mean interarrival time of aperiodic transactions.

* concurrency control: locking, timestamp ordering, and priority-based.

UI initiates the Configuration Manager (CM) which initializes necessary data structures for transaction processing based on user specification. CM invokes the Transaction Generator at an appropriate time interval to generate the next transaction.

Transaction execution consists of read and write operations. Each read or write operation is preceded by an access request sent to the Resource Manager, which maintains the local database at each site. Each transaction is assigned to the Transaction Manager (TM). The TM issues service requests on behalf of the transaction and reacts appropriately to the request replies. For instance, if a transaction requests access to a data object that is already locked, the TM executes either blocking operation to wait until the data object can be accessed, or aborting the transaction, depending on the situation. TM executes the two-phase commit protocol to ensure that a transaction commits or aborts globally.

The prototyping environment is currently implemented on a single host. The distributed environment is simulated by the Message Server (MS) listening on a well-known port for messages from remote sites. When a message is sent to a remote site, it is placed on the message queue of the destination site and the sender can block itself on a private semaphore until the message is retrieved by the MS at the receiving site. If the receiving site is not operational, a time-out mechanism will unblock the sender process. When the MS retrieves a message, it makes the sender process and forwards the message to the proper servers or TM. The prototyping environment implements Ada-style rendezvous (synchronous) as well as asynchronous message passing. Inter-process communication within a site does not go through the Message Server; processes send and receive messages directly through their associated ports.

The Performance Monitor interacts with the transaction managers to record priority/timestamp and read/write data set for each transaction, time when each event occurred, statistics for each transaction in each node. The statistics for a transaction includes arrival time, start time, total processing time, blocked interval, whether deadline was missed or not, and the number of aborts.

3. Prototyping Real-Time Database Systems

In this section, we present a real-time database system prototyped using the prototyping environment. Two goals of our prototyping work are 1) evaluation of the prototyping environment itself in terms of correctness, functionality, and modularity, by using it in prototyping distributed database systems, and 2) performance evaluation of real-time locking and priority-based synchronization protocols through the sensitivity study of key parameters that affect performance.

Real-time databases are often used by applications such as tracking. Tasks in such applications consist of both computing (signal processing) and database accessing (transactions). A task can have multiple transactions, which consists of a sequence of read and write operations operating on the database. Each transaction will follow the two-phase locking protocol [Es976], which requires a transaction to acquire all the locks before it releases any lock. Once a transaction releases a lock, it cannot acquire any new lock. A high priority task will preempt the execution of lower priority tasks unless it is blocked by the locking protocol at the database. In this section we consider them in a single site environment. Real-time locking protocols in distributed environment is discussed in the next section.

3.1. Priority-Based Synchronization

In a real-time database system, synchronization protocols must not only maintain the consistency constraints of the database but also satisfy the timing requirements of the transactions accessing the database. To satisfy both the consistency and real-time constraints, it is necessary to integrate synchronization protocols with real-time priority scheduling algorithms. Due to the effect of blocking in lock-based synchronization protocols, a direct application of a real-time scheduling algorithm to transactions may result in a situation known as priority inversion. Priority inversion is said to occur when a higher priority task is forced to wait for the execution of a lower priority task for an indefinite period of time. When two transactions attempt to access the same data object, the access must be serialized to maintain consistency. If the transaction of the higher priority task gains access first, then the process priority order is maintained; however, if the transaction of the lower priority gains access first and then the higher priority task requests access to the data object, this higher priority task will be blocked until the lower priority transaction completes its access to the data object. Priority inversion is inevitable in transaction systems. However, to achieve a high degree of schedulability in real-time applications, priority inversion must be minimized. This is illustrated by the following example.

Example: Suppose T1, T2, and T3 are three transactions arranged in descending order of priority with T1 having the highest priority. Assume that T1 and T3 access the same data object O1. Suppose that at time t1 transaction T1 obtains a lock on O1. During the execution of T1, the high priority transaction T2 arrives, preempts T1, and later attempts to access the object O1. Transaction T2 will be blocked, since O1 is already locked. We would expect that T1, being the highest priority transaction, will be blocked no longer than the time for transaction T2 to complete and unlock O1. However, the duration of blocking may, in fact, be unpredictable. This is because transaction T2 can be blocked by the intermediate priority transaction T3 that does not need to access O1. The blocking of T2, as well as the blocking of T3, and hence that of T1, will continue until T3 and any other pending intermediate priority level transactions are completed.

The blocking duration in the example above can be arbitrarily long. This situation can be partially remedied if transactions are not allowed to be preempted; however, this solution is only appropriate for very short transactions, because it creates unnecessary blocking. For instance, once a long low priority transaction starts execution, a high priority transaction not requiring access to the same set of data objects may be needlessly blocked.
An approach to this problem, based on the notion of priority inheritance, has been proposed [Sha87]. The basic idea of priority inheritance is that when a transaction T of a task blocks a higher priority task, it executes at the highest priority of all the transactions blocked by T. This simple idea of priority inheritance reduces the blocking time of a higher priority transaction. However, this is inadequate because the blocking duration for a transaction, though bounded, can still be substantial due to the potential chain of blocking. For instance, suppose that transaction T1 needs to sequentially access objects O1 and O2. Also suppose that T2 preempts T1 which has already locked O2. Then, T2 locks O1. Transaction T1 arrives at this instant and finds that the objects O1 and O2 have been respectively locked by the lower priority transactions T2 and T3. As a result, T1 would be blocked for the duration of two transactions, once to wait for T2 to release O1, and again to wait for T3 to release O2. Thus a chain of blocking can be formed.

One idea for dealing with this inadequacy is to use a total priority ordering of active transactions [Sha88]. A transaction is said to be active if it has started but not yet completed its execution. A transaction can be active in one of two states: executing or being preempted in the middle of its execution. The idea of total priority ordering is that the real-time locking protocol ensures that each active transaction is executed at some priority level, taking priority inheritance and read/write semantics into consideration.

### 3.2. Total Ordering by Priority Ceiling

To ensure the total priority ordering of active transactions, three priority ceilings are defined for each data object in the database: the write-priority ceiling, the absolute-priority ceiling, and the rw-priority ceiling. The write-priority ceiling of a data object is defined as the priority of the highest priority transaction that may write into this object, and absolute-priority ceiling is defined as the priority of the highest priority transaction that may read or write the data object. The rw-priority ceiling is set dynamically. When a data object is write-locked, the rw-priority ceiling of this data object is defined to be equal to the absolute priority ceiling. When it is read-locked, the rw-priority ceiling of this data object is defined to be equal to the write-priority ceiling.

The priority ceiling protocol is premised on systems with a fixed priority scheme. The protocol consists of two mechanisms: priority inheritance and priority ceiling. With the combination of these two mechanisms, we get the properties of freedom from deadlock and a worst case blocking of at most a single lower priority transaction.

When a transaction attempts to lock a data object, the transaction's priority is compared with the highest rw-priority ceiling of all data objects currently locked by other transactions. If the priority of the transaction is not higher than the rw-priority ceiling, the access request will be denied, and the transaction will be blocked. In this case, the transaction is said to be blocked by the transaction which holds the lock on the data object of the highest rw-priority ceiling. Otherwise, it is granted the lock. In the denied case, the priority inheritance is performed in order to overcome the problem of uncontrollable priority inversion. For example, if transaction T blocks higher transactions, T inherits P_T, the highest priority of the transactions blocked by T.

Under this protocol, it is not necessary to check for the possibility of read-write conflicts. For instance, when a data object is write-locked by a transaction, the rw-priority ceiling is equal to the highest priority transaction that can access it. Hence, the protocol will block a higher priority transaction that may write or read it. On the other hand, when the data object is read-locked, the rw-priority ceiling is equal to the highest priority transaction that may write it. Hence, a transaction that attempts to write it will have a priority no higher than the rw-priority ceiling and will be blocked. Only the transaction that read it and have priority higher than the rw-priority ceiling will be allowed to read-lock it, since read-locks are compatible. Using the priority ceiling protocol, mutual deadlock of transactions cannot occur and each transaction can be blocked by at most one lower priority transactions until it completes or suspends itself. For a more detailed discussion, readers are referred to [Sha88]. The next example shows how transactions are scheduled under the priority ceiling protocol.

**Example:** Consider the same situation as in the previous example. According to the protocol, the priority ceiling of O3 is the priority of T1. When T2 tries to access a data object, it is blocked because its priority is not higher than the priority ceiling of O3. Therefore, T2 will be blocked only once by T3 to access O3 regardless of the number of data objects it may access.

The total priority ordering of active transactions leads to some interesting behavior. As shown in the example above, the priority ceiling protocol may forbid a transaction from locking an unlocked data object. At first sight, this seems to introduce unnecessary blocking. However, this can be considered as the "insurance premium" for preventing deadlock and achieving block-at-most-once property.

Using the prototyping environment, we have been investigating issues associated with this idea of total ordering in priority-based scheduling protocols. One of the critical issues related to the total ordering approach is its performance compared with other design alternatives. In other words, it is important to figure out what is the actual cost for the "insurance premium" of the total priority ordering approach. In our experiments, all transactions are assumed to be hard in the sense that there will be no value in completing a transaction after its deadline. Transactions that miss the deadline are aborted, and disappear from the system.

### 3.3. Performance Evaluation

Various statistics have been collected during the experiments for comparing the performance of the priority ceiling protocol with other synchronization control algorithms. Transaction throughput and the percentage of deadline missing transactions are the most important performance measures in real-time database systems. This section presents these performance measures in a single site database system. Performance in distributed environments will be discussed in the next section.

Transaction are generated with exponentially distributed interarrival times, and the data objects updated by a transaction are chosen uniformly from the database. The total processing time of a transaction is directly related to the number of data objects accessed. Due to space considerations, we do not present all our results but have selected the graphs which best illustrate the difference and performance of the algorithms. For example, we have omitted the results of an experiment that
varied the size of the database, and thus the probability of con-

flicts, because they only confirm and not increase the
knowledge yielded by other experiments.

For each experiment and for each algorithm tested, we
collected performance statistics and averaged over the 10 runs.
The percentage of deadline-missing transactions is calculated
with the following equation: \[ \%\text{missed} = 100 \times (\text{number of}
\text{deadline-missing transactions}) / (\text{number of transactions}
\text{processed}). \]
A transaction is processed if either it executes com-
pletely or it is aborted. In our experiments, all transactions are
assumed to be hard in the sense that there will be no value
in completing a transaction after its deadline. Transactions that
miss the deadline are aborted, and disappear from the system.

We have used transaction size (the number of data
objects a transaction needs to access) as one of the key vari-
ables in the experiments. It varies from a small fraction up to a
relatively large portion (10%) of the database so that conflicts
would occur frequently. The high conflict rate allows synchron-
ization protocols to play a significant role in determining sys-
tem performance. We also chose the average arrival rate so that
protocols are tested in a heavily loaded rather than lightly
loaded system. For designing real-time systems, one must con-
sider high load situations. Even though they may not arise fre-
quently, one would like to have a system that misses as few
deadlines as possible when such peaks occur. In other words,
when a crisis occurs and the database system is under pressure
it is precisely when making a few extra deadlines could be most
important [Abb88].

We normalize the transaction throughput in terms of data
objects accessed per second for successful transactions, not in
transactions per second, in order to account for the fact that
bigger transactions need more database processing. The nor-
malization rate is obtained by multiplying the transaction com-
pletion rate (transactions/second) by the transaction size (data
objects accessed/transaction).

In Figure 2, the throughput of the priority ceiling proto-
col (C), the two-phase locking protocol with priority mode (P),
and the two-phase locking protocol without priority mode (L),
is shown for transactions of different sizes. Since we chose
the average arrival rate to make the system heavily loaded, both
CPU and I/O were very heavily loaded when the average tran-
saction size reaches 20. As the transaction size increases, there
is little impact on the throughput of the priority ceiling protocol
over a range of transaction sizes shown in Figure 2. This is
because in the priority ceiling protocol, the conflict rate is
determined by ceiling blocking rather than direct blocking, and
the frequency of ceiling blocking is not sensitive to the transac-
tion size.

However, the performance of the two-phase locking pro-
tocol with or without priority degrades very rapidly. This
phenomenon is more pronounced as the transaction workload
becomes more I/O bound, since there are few conflicts for the
small transactions in the two-phase locking protocol, and the
concurrency is fully achieved with an assumption of parallel
I/O processing. Poor performance of the two-phase locking pro-
tocol for bigger transactions is due to the high conflict rate.

Another important performance statistic is the percen-
tage of deadline-missing transactions, since the synchronization
protocol in real-time database systems must satisfy the timing
constraints of individual transaction. In our experiments, each
transaction's deadline is set in proportion to its size and system
workload (number of transactions), and the transaction with the
earliest deadline is assigned the highest priority. As shown in
Figure 3, the percentage of deadline-missing transactions
increases sharply for the two-phase locking protocol as the tran-
saction size increases. A sharp rise was expected, since the pro-
bability of deadlocks would go up with the fourth power of the
transaction size [Gray81]. However, the percentage of
deadline-missing transactions increases more slowly as the
transaction size increases in the priority ceiling protocol. Since
there is no deadlock in priority ceiling protocol, the response
time is proportional to the transaction size and the priority rank-
ing.

4. Priority Ceiling in Distributed Environments

In this section, we discuss the use of the priority ceiling
approach as a basis for real-time locking protocol in a distri-
buted environment. The priority ceiling protocol might be
implemented in a distributed environment by using the global
celling manager at a specific site. In this approach, all decisions
for ceiling blocking is performed by the global ceiling manager.
Therefore all the information for ceiling protocol is stored at
the site of the global ceiling manager.

The advantage of this approach is that the temporal con-
sistency of the database is guaranteed, since every data object
maintains most up-to-date value. While this approach ensures
consistency, holding locks across the network is not very attrac-
tive. Owing to communication delay, locking across the net-
work will only enforce the processing of a transaction using
local data objects to be delayed until the access requests to the
remote data objects are granted. This delay for synchronization,
combined with the low degree of concurrency due to the strong
restrictions of the priority ceiling protocol, is counter-
productive in real-time database systems.

An alternative to the global ceiling manager approach is
to have replicated copies of data objects. An up-to-date local
copy is used as the primary copy, and remote copies are used as
the secondary read-only copies. In this approach, we assume a
single writer and multiple readers model for distributed data
objects. This is a simple model that effectively models applica-
tions such as distributed tracking in which each radar station
maintains its view and makes it available to other sites in the
network. For this approach to work, the following restrictions
are necessary:

1. Every data object is fully replicated at each site.
2. Data objects to be updated must be a primary copy at
the same site with the updating transaction.
3. Every transaction must be committed before updating
remote secondary copies.

Under these restrictions, the local ceiling manager at each site
can enforce the priority ceiling protocol for the synchronization
of not only the local data objects (primary or replicated copies),
but also remote primary copies and local replicated copies. The
first restriction is necessary because in a distributed database
environment, holding locks across the network will occur if all
the data objects requested by a transaction do not reside at the
local site. If we allow each transaction to update its local copy
without synchronizing with other transactions, transaction roll
back and subsequent abort may result as in optimistic
concurrency control. This situation is not acceptable in real-time applications. The second restriction prevents it by providing only a single primary copy.

If we insist that copies of a data objects must be identical with respect to all references, a transaction updating the primary copy cannot commit until all the remote copies are also updated. However, this solution requires locking data objects across the network, which can lead to long durations of blocking. The third restriction solves this problem by allowing remote copies to be historical copies of the primary copy; the primary and remote copies can be updated asynchronously. The third restriction, however, may cause a temporal inconsistency, owing to the delays in the network. That is, some of the views can be out of date. Even with this potential problem of reading out of date values, the third restriction is very critical in improving the system responsiveness in distributed environments. This also solves the problem of distributed deadlock. Since we do not have deadlocks at each site, and locks are not allowed to be held across the network, we cannot have distributed deadlocks.

We have investigated the performance characteristics of the global ceiling approach and the local ceiling approach with replication in a distributed environment. The real-time database system we have prototyped for the experiment consists of three sites with fully interconnected communication network. To focus on the impact of the transaction mix and the communication cost on the number of deadline-missing transactions, we did not include any I/O cost for the experiments. In other words, a memory-resident database system in a distributed environment was simulated. As in the single-site experiments, transactions enter the system with the exponentially distributed interarrival times and they are ready to execute when they appear in the system. Update transactions are assigned to a site based on their write-set, and read-only transactions are distributed randomly. The objects updated by a transaction are chosen uniformly from the database.

Figure 4 shows the ratio between the throughput of the global ceiling approach and that of the local ceiling approach, based on different transaction mix and communication delays. Even without considering the communication delay (i.e., communication delay = 0), the local ceiling approach achieves the throughput between 1.5 and 3 times higher than that of the global ceiling approach, over the wide range of transaction mix. The reason for this difference is that the degree of concurrency among the transactions at each site can be greatly improved due to the decoupling effect of data replication. If we consider communication delays, this performance ratio will increase accordingly to the communication delay as shown in Figure 4.

Figure 5 illustrates the ratio of the percentage of deadline missing transactions between the global and the local ceiling approach, based on different communication delays for a specific transaction mix (50% read-only and 50% update transactions). There is a significant difference between the two approaches in the number of deadline missing transactions, although the increase rate of this performance ratio varies with the communication delay. In the range of small communication delays (up to 2 time units), this ratio increases rapidly, and then rather slowly after that. As the communication delay increases, the performance ratio increases beyond 16. This implies that the global ceiling approach is more than 16 times likely to miss the real-time constraints than the local ceiling approach, for a given set of real-time transactions. Performance improvement of the local ceiling approach is more substantial with small communication delays than with large delays. This is because as communication delay increases, the concurrency achieved by the local ceiling approach is limited by the the communication cost due to data replication. Figure 6 shows the percentage of deadline-missing transactions for two specific communication delays. As shown in Figure 5, the performance difference in terms of deadline-missing transactions between two approaches increases as the communication delay increases over a wide range of transaction mix. As the proportion of read-only transactions increases, the number of deadline-missing transactions decreases since the conflict rate will decrease.

Our performance results have illustrated the superiority of the local ceiling approach over the global ceiling approach, at least under one representative distributed real-time database and transaction model. Hence, from this experimentation, we believe that, even with the potential problem of temporal inconsistency (i.e., reading out of date values), the local ceiling approach is a very powerful technique for real-time concurrency control in distributed database systems.

There are applications where a temporally consistent view is more important than just the latest information that can be obtained at each site. For example, in an application like tracking, a local track would be updated periodically in conjunction with repetitive scanning. In order to provide a temporally consistent view in a distributed environment, we can utilize the periodicity of the update transaction as a timestamp mechanism. If the system provide multiple versions of data objects, ensuring a temporally consistent view becomes real-time scheduling problem in which the time lags in the distributed versions need to be controlled. Once the time lags can be controlled by the timestamps of data objects, transactions can read the proper versions of distributed data objects, and ensure that decisions are based on temporally consistent data.

5. Conclusions

Prototyping large software systems is not a new approach. However, methodologies for developing a prototyping environment for distributed database systems have not been investigated in depth in spite of its potential benefits. In this paper, we have presented a prototyping environment that has been developed based on the StarLite concurrent programming kernel and message-based approach with modular building blocks. Although the complexity of a distributed database system makes prototyping difficult, current implementation of the prototyping environment has proven satisfactory for experimentation of design choices, different database techniques and protocols, and even an integrated evaluation of database systems. It supports a very flexible user interface to allow a wide range of system configurations and workload characteristics. Since our prototyping environment is designed to provide a spectrum of database functions and operating system modules, it facilitates the development of multiple system instances with different characteristics without much overhead. Expressive power and performance evaluation capability of our prototyping environment has been demonstrated by prototyping a distributed real-time database system and investigating its performance characteristics.

In real-time database systems, transactions must be scheduled to meet their timing constraints. In addition, the
system should support a predictable behavior such that the pos-
sibility of missing deadlines of critical tasks could be informed
ahead of time, before their deadlines expire. Priority ceiling
protocol is one approach to achieve a high degree of schedula-
bility and system predictability. In this paper, we have investi-
gated this approach and compared its performance with other
techniques and design choices. It is shown that this technique
might be appropriate for real-time transaction scheduling since
it is very stable over the wide range of transaction sizes, and
compared with two-phase locking protocols, it reduces the
number of deadline-missing transactions.

There are other technical issues associated with
priority-based scheduling protocols that need further investiga-
tion. For example, the analytic study of the priority ceiling pro-
tocol provides an interesting observation that the use of read
and write semantics of a lock may lead to worse performance
in terms of schedulability than the use of exclusive semantics of a
lock. This means that the read semantics of a lock cannot be
used to allow several readers to hold the lock on the data object,
and the ownership of locks must be mutually exclusive. Is it
necessarily true? We are investigating this and other related
issues using the prototyping environment.

Transaction scheduling options for real-time database
systems also need further investigation. In priority ceiling pro-
tocol and many other database scheduling algorithms, preemp-
tion is usually not allowed. To reduce the number of deadline-
missing transactions, however, preemption must be re-
considered. The preemption decision in a real-time database system
must be made very carefully, and as pointed out in [Stan88], it
should not necessarily be based only on relative deadlines. Since
preemption implies not only that the work done by the
preempted transaction must be undone, but also that later on, if
restarted, must redo the work. The resultant delay and the
wasted execution may cause one or both of these transactions,
as well as other transaction to miss the deadlines. Several
approaches to designing scheduling algorithms for real-time
transactions have been proposed [Liu87, Stan88, Abb88], but their
performance in distributed environments is not studied.
The prototyping environment described in this paper is an
appropriate research vehicle for investigating such new tech-
niques and scheduling algorithms for distributed real-time data-
base systems.

References

Abb88 Abbott, R. and H. Garcia-Molina, "Scheduling
IEEE89 Sixth IEEE Workshop on Real-Time Operating Sys-
Liu87 Liu, J. W. S., K. J. Lin, and S. Natarajan, "Schedul-
Sha88 Sha, L., R. Rajkumar, and J. Lehoczky, "Concur-
tency Control for Distributed Real-Time Data-
bases," ACM SIGMOD Record 17, 1, Special Issue on Real-Time Database Systems, March 1988, 82-98.
Son89 Son, S. H. and A. Agrawala, "Distributed Checkpointing for Globally Consistent States of Databases," IEEE Transactions on Software Engineer-
Son90 Son, S. H., "An Environment for Prototyping Real-
Time Distributed Databases," International Confer-
ence on Systems Integration, Morristown, New Jer-
sey, April 1990.
Stan88 Stankovic, J. and W. Zhao, "On Real-Time Transac-
tions," ACM SIGMOD Record 17, 1, Special Issue on Real-Time Database Systems, March 1988, 4-18.
Wat88 Watson, P., "An Overview of Architectural Direc-
tions for Real-Time Distributed Systems," Fifth IEEE Workshop on Real-Time Software and Operat-
Fig. 2 Transaction Throughput.

Fig. 3 Percentage of Deadline Missing Transactions.

Fig. 4 Transaction Throughput Ratio.

Fig. 5 Deadline Missing Ratio.

Fig. 6 Deadline Missing Transaction Percentage.