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

Sang H. Son and Chun-Hyon Chang

Computer Science Report No. TR-89-13
November 16, 1989

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

Sang H. Son
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903
USA

Chun-Hyon Chang
Department of Computer Science
Kon Kuk University
Seoul, Korea

ABSTRACT

Real-time systems must maintain 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. One of the reasons for the difficulty in developing and evaluating database synchronization techniques 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. This paper describes a prototyping environment for investigating distributed real-time database systems, and its use for performance evaluation of priority-based locking protocols for real-time database systems.

Index Terms - real-time system, distributed database, prototyping, synchronization, transaction, priority

This work was supported in part by the Office of Naval Research under contract number N00014-88-K-0245, by the Department of Energy under contract number DEFG05-88-ER25063, and by the Federal Systems Division of IBM Corporation under University Agreement WG-249153.

1. Introduction

As computers are becoming essential part of real-time systems, *real-time computing* is emerging as an important discipline in computer science and engineering [Shin87]. The growing importance of real-time computing in a large number 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 inter-process communications and remote database accesses must be taken into account. Most existing real-time architectures are, in fact, stripped down and optimized time sharing architectures [Wat88]. There is no support for the explicit management of time and, in most cases, there are features in the architecture which actually hinder the implementation of scientific real-time scheduling algorithms.

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 [ONR88, IBM88], developers of "real" 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 and the announcements by some vendors of database products that include features achieving high availability and predictability [Son88b].

In addition to providing relational access capabilities, distributed real-time database systems offer a means of loosely coupling software processes; therefore, 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 requirements and they commonly lock data tables indiscriminately to assure database consistency. Locks and time-driven scheduling are basically incompatible. Low priority transactions can and will block higher priority transactions leading to response requirement failures. New techniques are required to manage database consistency which are compatible with time-driven scheduling and the essential system response predictability/analyzability it brings. One of the primary reasons for the difficulty in successfully developing and evaluating new database techniques 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.

A prototyping technique can be applied effectively to the evaluation of database techniques for distributed real-time systems. A *database prototyping environment* is a software package that supports the investigation of the properties of database techniques in an environment other than that of the target database system. The advantages of an environment that provides prototyping capability are obvious. First, it is cost effective. If experiments for a twenty-node distributed database system can be executed in a software environment, it is not necessary to purchase a twenty-node distributed system, reducing the cost of evaluating design alternatives. Second, design alternatives can be evaluated in a uniform environment with the same system parameters, making a fair comparison. Finally, as technology changes, the environment need only be updated to provide researchers with the ability to perform new experiments.

A prototyping environment can reduce the time of 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. Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation, especially for distributed real-time database systems.

Another important use of a prototyping environment is to analyze the reliability of database control mechanisms and techniques. Since distributed systems are expected to work correctly under various failure situations, the behavior of distributed database systems in degraded circumstances needs to be well understood. Although new approaches for synchronization and database recovery have been developed recently [Son87, Son88, Son89], experimentation to verify their properties and to evaluate their performance has not been performed due to the lack of appropriate test tools.

This paper describes a message-based approach to prototyping study of distributed real-time database systems, and presents a prototyping software implemented for a series of experimentation to evaluate priority-based real-time locking protocols.

When prototyping distributed database systems, there are two possible approaches: sequential programming and distributed programming based on message-passing. Message-based simulations, in which events are message-communications, do not provide additional expressive power over standard simulation languages; message-passing can be simulated in many discrete-event simulation languages including SIMSCRIPT [Kiv69] and GPSS [Sch74]. However, a message-based simulation can be used as an effective tool for developing a distributed system because the simulation "looks" like a distributed program, while a simulation program written in a traditional simulation language is inherently a sequential program. Furthermore, if a simulation program is developed in a systematic way such that the principles of modularity and information hiding are observed, most of the simulation code can be used in the actual system, resulting in a reduced cost for system development and evaluation.

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

2. Structure of the Prototyping Environment

For a prototyping tool for distributed database systems to be effective, appropriate operating system support is mandatory. Database control mechanisms need to be integrated with the operating system, because the correct functioning of control algorithms depends on the services of the underlying operating system; therefore, an integrated design reduces the significant overhead of a layered approach during execution.

Although an integrated approach is desirable, the system needs to support flexibility which may not be possible in an integrated approach. In this regard, the concept of developing a library of modules with different performance and reliability characteristics for an operating system as well as database control functions seems promising. Our prototyping environment follows this approach [Cook87, Son88c]. It 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 efficiently supports a spectrum of distributed database functions at the operating system level, and facilitates the construction of multiple "views" with different characteristics. For experimentation, system functionality can be adjusted according to application-dependent requirements without much overhead for new system setup.

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. To permit concurrent transactions on a single site, there is a separate process for each transaction that coordinates with other server processes.

Figure 1 illustrates the structure of the prototyping environment. The prototyping environment is based on a concurrent programming kernel, called the StarLite kernel, written in Modula-2. The StarLite kernel supports process control to create, ready, block, and terminate processes. Scheduler in the kernel maintains the simulation clock and provides the **hold** primitive to simulate the passage of time.

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:

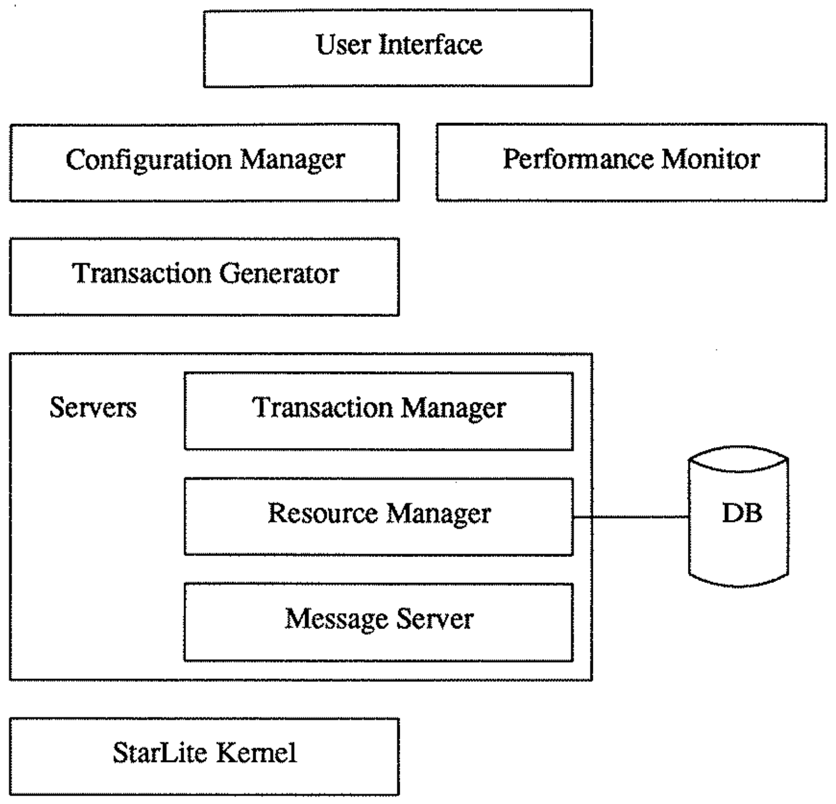


Fig. 1. Structure of the prototyping environment

- system configuration: number of sites and the number of server processes at each site.
- 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 or update) and their priorities, and the mean interarrival time of 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 to form a Poisson process of transaction arrival. When a transaction is generated, it is assigned an identifier that is unique among all transactions in the system.

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). 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 file and that file is locked, TM executes either blocking operation to wait until the data object can be accessed, or aborting procedure, depending on the situation. If granting access to a resource will produce deadlock, TM receives an abort response and aborts the transaction. Transactions commit in two phases. The first commit phase consists of at least one round of messages to determine if the transaction can be globally committed. Additional rounds may be used to handle potential failures. The second commit phase causes the data objects to be written to the database for successful transactions. TM executes the two commit phases to ensure that a transaction commits or aborts globally.

The Message Server (MS) is a process 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 blocks itself on a private semaphore until the message is retrieved by MS. If the receiving site is not operational, a time-out mechanism will unblock the sender process. When MS retrieves a message, it wakes 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 and cpu hold interval 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 number of aborts.

Since each TM is a separate process, each has its own data area in which to keep track of the time when a service request is sent out and the time the response arrives, as well as the time when a transaction begins blocking, waiting for a resource, and the time the resource is granted. When a transaction commits, it calls a procedure that records the above measures; when the simulation clock has expired, these measures are printed out for all transactions.

3. Prototyping Real-Time Database Systems

The previous section described the structure of the prototyping environment. In this section, we present a real-time database system implemented 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 implementing 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 embedded transactions, which consists of a sequence of read and write operations operating on the database. Each embedded transaction will follow the two-phase locking protocol [Esw76], 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 protocols. A major source of problems in integrating the two protocols is the lack of coordination in the development of synchronization protocols and real-time priority scheduling protocols. 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 condition known as *priority inversion*. Priority inversion is said to occur when a higher priority process is forced to wait for the execution of a lower priority process for an indefinite period of time. When the transactions of two processes attempt to access the same data object, the access must be serialized to maintain consistency. If the transaction of the higher priority process gains access first, then the proper priority order is maintained; however, if the transaction of the lower priority gains access first and then the higher priority transaction requests access to the data object, this higher priority process 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 T_1, T_2, and T_3 are three transactions arranged in descending order of priority with T_1 having the highest priority. Assume that T_1 and T_3 access the same data object O_i. Suppose that at time t_1 transaction T_3 obtains a lock on O_i. During the execution of T_3, the high priority transaction T_1 arrives, preempts T_3 and later attempts to access the object O_i. Transaction T_1 will be blocked, since O_i is already locked. We would expect that T_1, being the highest priority transaction, will be blocked no longer than the time for transaction T_3 to complete and unlock O_i. However, the duration of blocking may, in fact, be unpredictable. This is because transaction T_3 can be blocked by the intermediate priority transaction T_2 that does not need to access O_i. The blocking of T_3, and hence that of T_1, will continue until T_2 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 process blocks higher priority processes, 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 T_1 needs to sequentially access objects O_1 and O_2. Also suppose that T_2 preempts T_3 which has already locked O_2. Then, T_2 locks O_1. Transaction T_1 arrives at this instant and finds that the objects O_1 and O_2 have been respectively locked by the lower priority transactions T_2 and T_3. As a result, T_1 would be blocked for the duration of two transactions, once to wait for T_2 to release O_1 and again to wait for T_3 to release O_2. 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 higher 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, it will be denied. Otherwise, it is granted the lock. In the denied case, the priority inheritance is performed in order to overcome the problem of uncontrolled priority inversion.

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. For a more formal discussion on the protocol, 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 O_i is the priority of T_1. When T_2 tries to access a data object, it is blocked because its priority is not higher than the priority ceiling of O_i. Therefore T_1 will be blocked only once by T_3 to access O_i, 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 technical 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. We discuss this performance issue of the real-time locking protocols in the next section. 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 immediately with some abort cost.

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 number of deadline-missing transactions are the most important performance measures in real-time database systems. This experiment investigates these performance characteristics 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. A transaction has an execution profile which alternates data access requests with equal computation requests, and some processing requirement for termination (either commit or abort). Thus the total processing time of a transaction is directly related to the number of data objects accessed. Due to space considerations, we cannot 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 number of conflicts, 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: *%missed* = 100 * (number of deadline-missing transactions / number of transactions processed). A transaction is processed if either it executes completely or it is aborted. We assume that all transactions are *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 immediately with some abort cost. We have used transaction size (the number of data objects a transaction needs to access) as one of the key variables 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 synchronization protocols to play a significant role in determining system performance. We chose the arrival rate so that protocols are tested in a heavily loaded rather than lightly loaded system. For designing real-time systems, one must consider high load situations. Even though they may not arise frequently, 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 is precisely when making a few extra deadlines could be most important [Abb88].

We normalize the transaction throughput in records 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 normalization rate is obtained by multiplying the transaction completion rate (transactions/second) by the transaction size (database records accessed/transaction). In Figure 2, the throughput of the priority-ceiling protocol (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 of different sizes with balanced workload (a), I/O bound workload (b), and CPU bound workload (c), respectively.

As the transaction size increases, there is little impact on the throughput of the priority-ceiling protocol over a range of transaction sizes and over the workload type 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 transaction size.

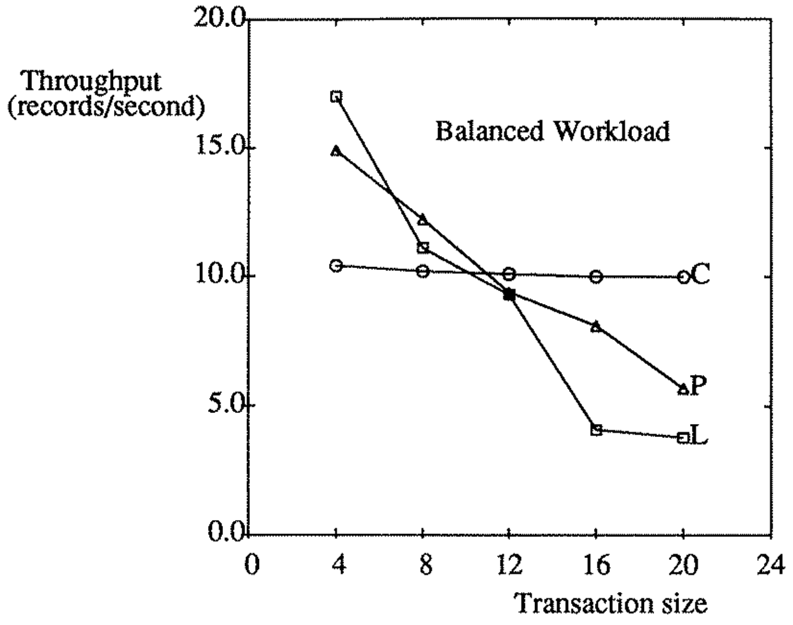
However, the performance of the two-phase locking protocol with or without priority degrades very rapidly. This phenomenon is more pronounced as the transaction workload becomes more I/O bound, since there are few conflicts for the small transactions in the two-phase locking protocol, and the concurrency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

Another important performance statistic is the percentage 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 transaction size increases. A sharp rise was expected, since the probability 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 ranking.

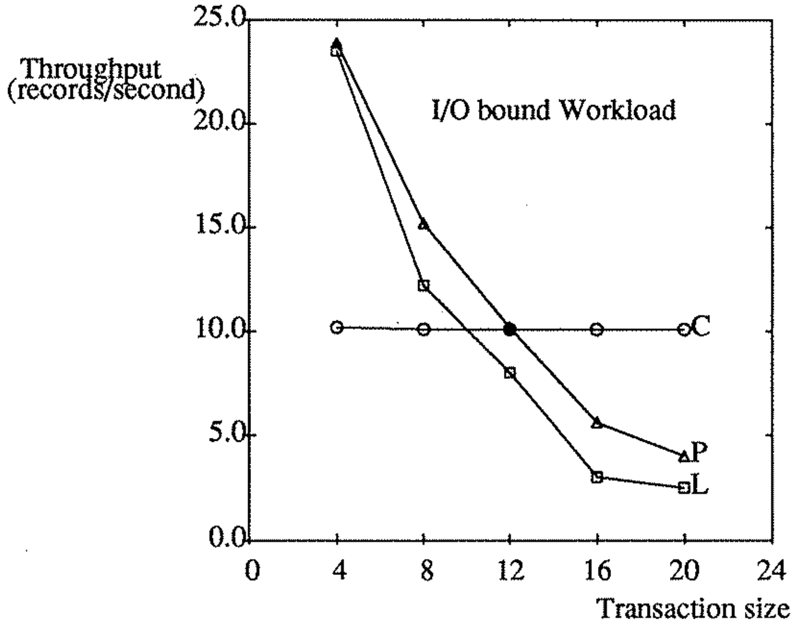
4. Priority Ceiling in Distributed Environments

In this section, we discuss the use of the priority ceiling approach as a basis for real-time locking protocol in a distributed environment. The priority ceiling protocol might be implemented in a distributed environment by using the global ceiling 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.

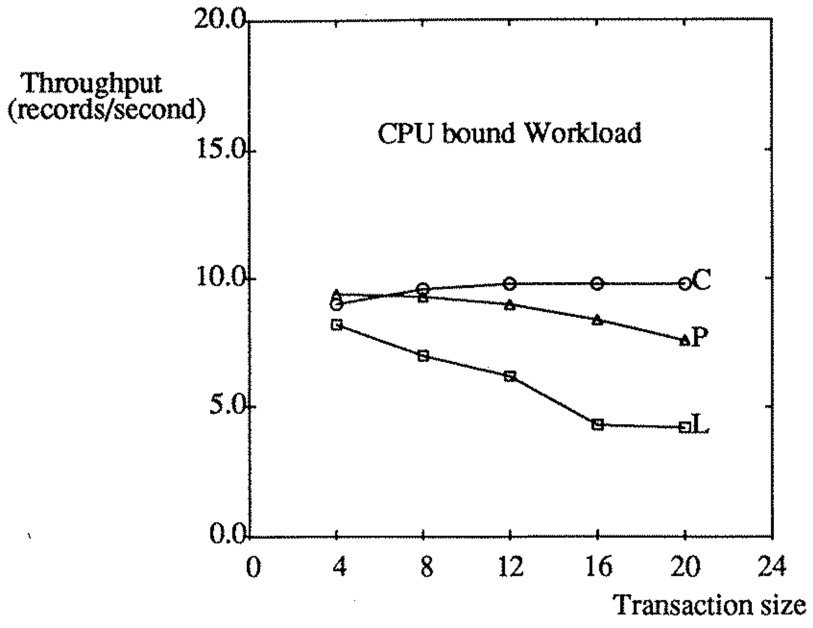
The advantage of this approach is that the temporal consistency of the database is guaranteed, since every data object maintains most up-to-date value. While this approach ensures consistency, holding locks across the network is not very attractive. Owing to communication delay, locking across the net-



a) balanced workload transaction



b) I/O bound workload transaction



c) CPU bound workload transaction

C: priority_ceiling protocol
P: 2-phase locking protocol with priority mode
L: 2-phase locking protocol without priority mode

Fig. 2 Transaction Throughput.

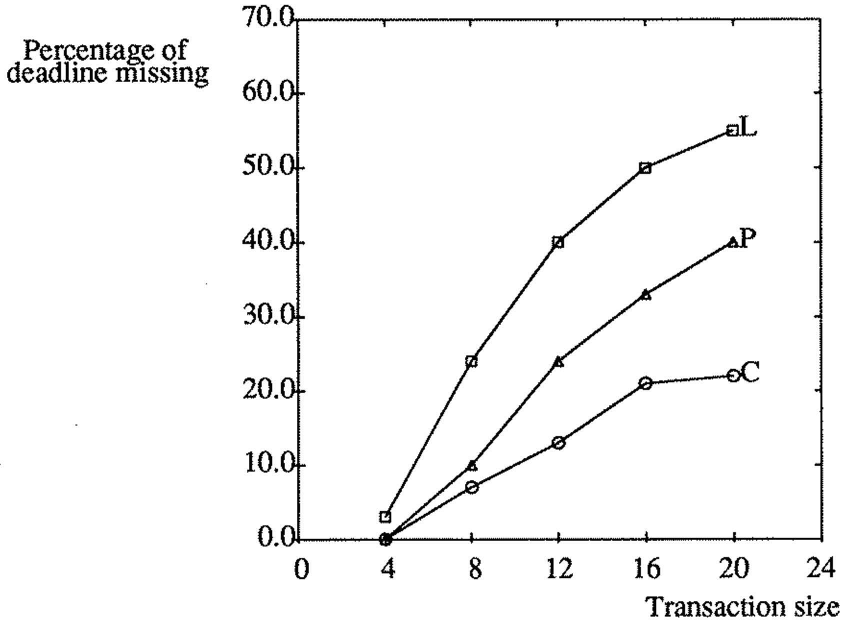


Fig. 3 Percentage of Deadline Missing Transactions.

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 applications 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 implemented for the experiment, using the prototyping environment, 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 simulated a memory resident database system. 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. For reasons of space, we do not present all our results but have selected the graphs which best illustrate the performance difference between the two approaches.

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 according 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 a 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, the implementation 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 construction of multiple system instances with different characteristics without much overhead. Expressive power and performance evaluation capability of our prototyping environment has been demonstrated by implementing a distributed real-time database system and investigating its performance characteristics.

Compared with traditional databases, real-time database systems have a distinct feature: they must satisfy not only the database consistency constraints but also the timing constraints associated with transactions. In other words, "time" is one of the key factors to be considered in real-time database systems. Transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. Priority ceiling protocol is one approach to achieve a high degree of schedulability and system predictability. In this paper, we have investigated 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 reduce the number of deadline-missing transactions.

There are many technical issues associated with priority-based scheduling protocols that need further investigation. For example, the analytic study of the priority ceiling protocol 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 protocol and many other database scheduling algorithms, preemption is usually not allowed. To reduce the number of deadline-missing transactions, however, preemption may need to be 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 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 techniques and scheduling algorithms for distributed real-time database systems.

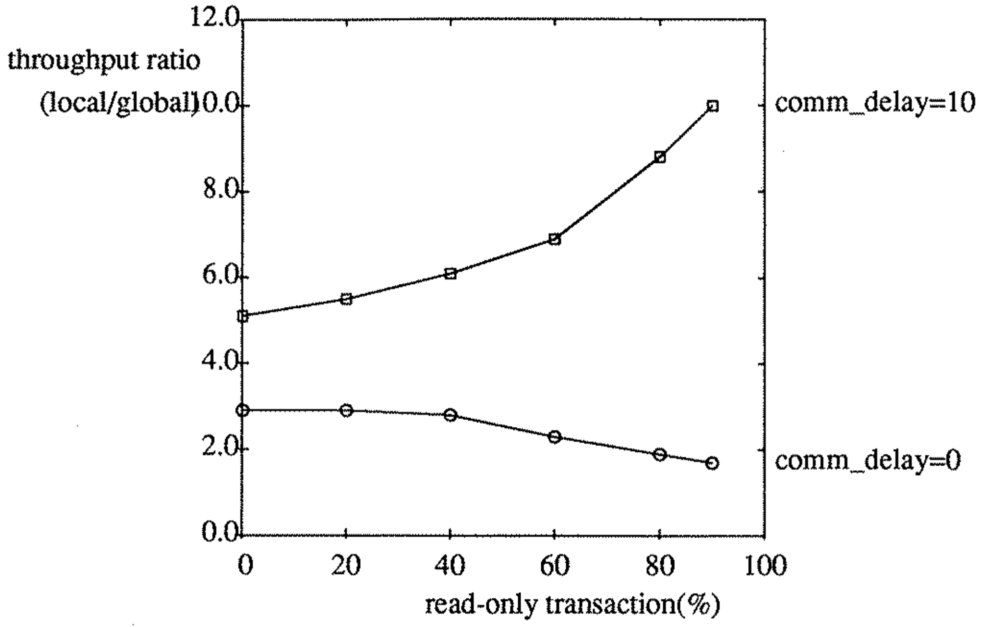


Fig. 4 Transaction Throughput Ratio

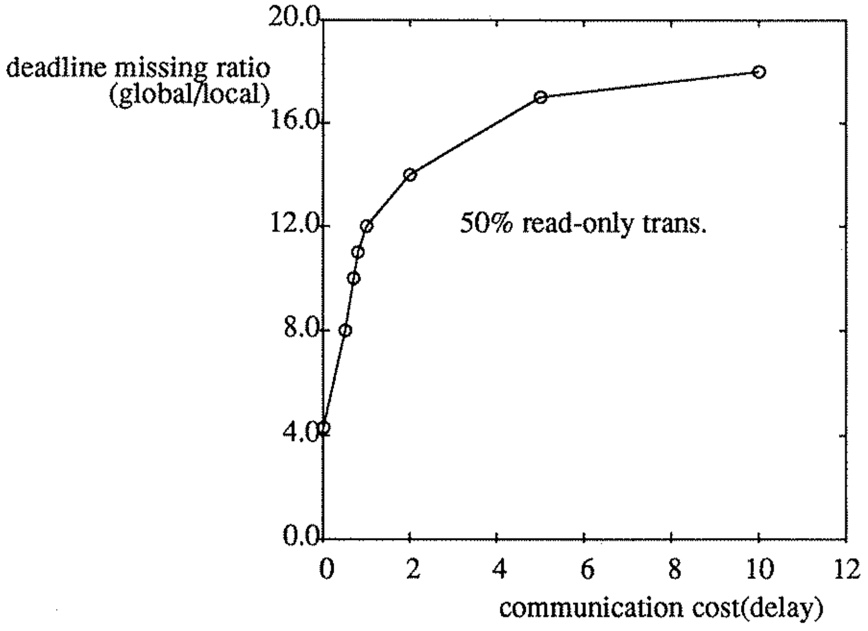
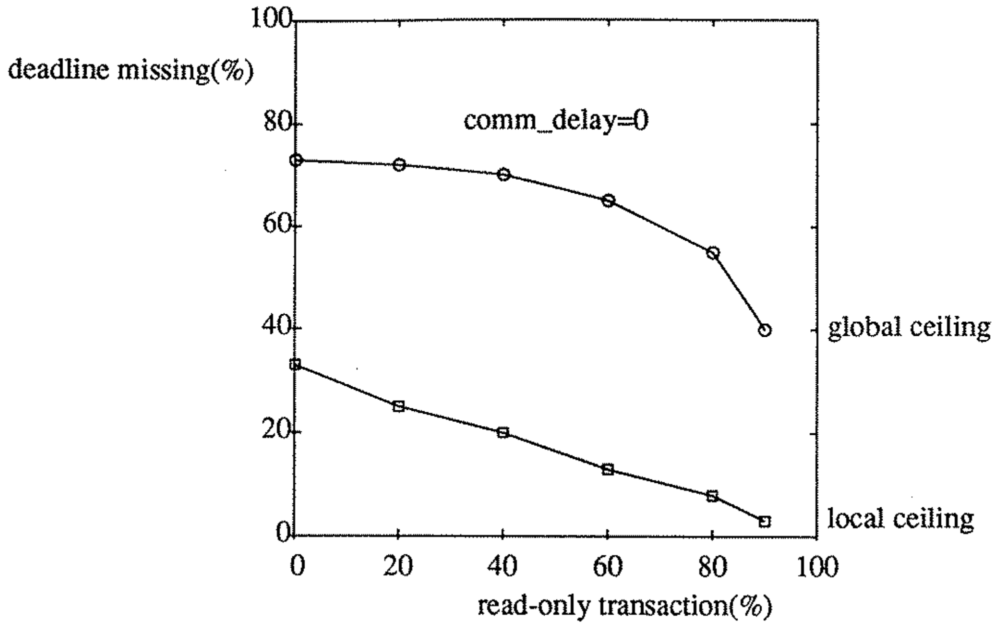


Fig. 5 Deadline Missing Ratio



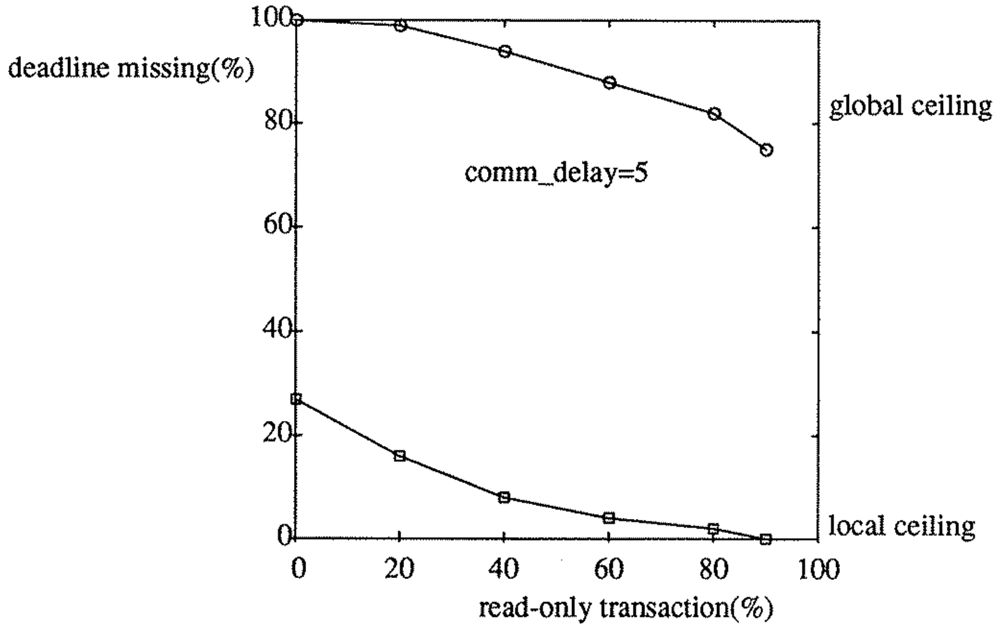


Fig. 6 Deadline Missing Transaction Percentage

References

[Abb88] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Study," *VLDB Conference*, Sept. 1988, 1-12.

[Esw76] Eswaran, K. P. et al, "The Notion of Consistency and Predicate Locks in a Database System," *Comm. of the ACM*, Nov. 1976, 624-633.

[Cook87] Cook, R. and S. H. Son, "The StarLite Project," *Fourth IEEE Workshop on Real-Time Operating Systems*, Cambridge, Massachusetts, July 1987, 139-141.

[Gray81] Gray, J. et al., "A Straw Man Analysis of Probability of Waiting and Deadlock," *IBM Research Report*, RJ 3066, 1981.

[Hask87] Haskin, R. et al., "Recovery Management in QuickSilver," *11th ACM Symposium on Operating Systems Principles*, Austin, Texas, Nov. 1987, 107-108.

[IBM88] *IBM Real-Time Systems Requirements and Issues Workshop*, Manassas, Virginia, April 1988.

[Kiv69] Kiviat, P., R. Villareau, and H. Markowitz, *The SIMSCRIPT II Programming Language*, Englewood Cliffs, NJ, Prentice-Hall, 1969.

[Liu87] Liu, J. W. S., K. J. Lin, and S. Natarajan, "Scheduling Real-Time, Periodic Jobs Using Imprecise Results," *Real-Time Systems Symposium*, Dec. 1987, 252-260.

[ONR88] *ONR Foundations of Real-Time Computing Research Initiative Workshop*, Fall Church, Virginia, Nov. 1988.

[Sch74] Schriber, T., *Simulation Using GPSS*, NY, Wiley, 1974.

[Sha87] Sha, L., R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocol: An Approach to Real-Time Synchronization," *Technical Report*, Computer Science Dept., Carnegie-Mellon University, 1987.

[Sha88] Sha, L., R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988, 82-98.

[Shin87] Shin, K. G., "Introduction to the Special Issue on Real-Time Systems," *IEEE Trans. on Computers*, Aug. 1987, 901-902.

[Son87] Son, S. H., "Synchronization of Replicated Data in Distributed Systems," *Information Systems 12*, 2, June 1987, 191-202.

[Son88] Son, S. H., "Semantic Information and Consistency in Distributed Real-Time Systems," *Information and Software Technology*, Vol. 30, September 1988, pp 443-449.

[Son88b] Son, S. H., "Real-Time Database Systems: Issues and Approaches," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988.

[Son88c] Son, S. H., "A Message-Based Approach to Distributed Database Prototyping," *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, May 1988, 71-74.

[Son89] Son, S. H. and A. Agrawala, "Distributed Checkpointing for Globally Consistent States of Databases," *IEEE Transactions on Software Engineering*, Vol. 15, No. 10, October 1989 (to appear).

[Stan88] Stankovic, J. and W. Zhao, "On Real-Time Transactions," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988, 4-18.

[Wat88] Watson, P., "An Overview of Architectural Directions for Real-Time Distributed Systems," *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, May 1988, 59-65.