OVERVIEW OF THE CARAT PROJECT

W.H. Kohler, J.A. Stankovic and D.F. Towsley

Distributed Computing Systems Laboratory
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 88-04

January 24, 1988

OVERVIEW OF THE CARAT PROJECT [1]

Walter H. Kohler John A. Stankovic Donald F. Towsley

Distributed Computing Systems Laboratory
University of Massachusetts
Amherst, Mass. 01003
January 24, 1988

1 INTRODUCTION

During the past decade, concurrency control and recovery in transaction oriented database systems has been the subject of intensive research. Even though a wide variety of concurrency control and recovery algorithms have been proposed, there is currently little quantitative knowledge on how the choice of algorithms impacts the performance of the overall transaction processing system.

The goals of the CARAT project are to develop methodologies for evaluating and modeling the impact of different design choices (both hardware and software) on the performance of distributed systems and to identify and develop protocols that improve system performance. Our research approach is based on a combination of empirical and analytical studies. It involves the design and implementation of alternative protocols, the measurement and evaluation of system performance under controlled load conditions, and the development and validation of analytic and simulation models which describe and explain the observed system behavior. We briefly summarize our project's results in seven areas: testbed system design and implementation, performance measurement and instrumentation, analytic modeling, algorithm design, implementation and evaluation of a database cache, transaction processing in a data sharing environment, and real-time transactions.

[1] This research was supported by grants from the National Science Foundation (ECS-8120931, ECS-8340029, and SDB-8418216) and from Digital Equipment Corporation.

2 Distributed Testbed Design and Implementation

The distributed testbed system we have implemented for the performance evaluation of concurrency control and recovery algorithms is called CARAT (Concurrency and Recovery Algorithm Testbed) [KOHL86b]. CARAT now runs on an Ethernet network of five DEC VAX computers under the VMS operating system with DECnet network support. The distributed database testbed is implemented as a collection of cooperating server processes which communicate via a uniform and efficient message passing mechanism. At each node there are two levels of servers. The top level server process, called the TM server, is the Transaction Manager. At the next lower level there is a pool of Data Managers, designated the DM servers. The TM and DM servers work cooperatively to service transaction requests from user application processes. CARAT contains the functional components: *transaction management*, *data management*, *log management*, *communication management*, and *catalog management*. Transaction management, inter-node communication and catalog management are the responsibilities of the TM server processes. The data management function is handled by the DM server processes. The TM and DM processes jointly manage the recovery log and concurrency control functions.

Some of the protocols currently implemented in CARAT are:

- a two-phase locking protocol with distributed deadlock detection.
- a distributed version of optimistic concurrency control.
- before-image and after-image write-ahead-logging mechanisms for transaction recovery.
- a two-phase commit protocol for atomicity of distributed transactions.

A configuration and initialization program was also implemented that allows the user to configure CARAT with various combinations of the above protocols, then automatically moves code images to various nodes of the system, initializes all CARAT processes, and starts the test. A program, called SPY, was implemented to monitor and display the state of the distributed system (including transactions) during tests and demonstrations.

3 Performance Measurement and Instrumentation

As our initial performance measurement study, we chose to examine and compare the performance of CARAT for two different combinations of concurrency control and recovery algorithms [JENQ86,KOHL86a]: two-phase locking with before-image journaling (2PL/BI), and optimistic concurrency control with after-image journaling (OPT/AI). Performance measures of both configurations of the testbed system were obtained for a two site partitioned database using various transaction workloads.

The basic system parameters that we measured are: communication overhead, cpu and disk I/O resource requirements, concurrency control costs, and the costs of journaling and aborting a transaction. These parameters are useful for performance modeling experiments. They have been used in developing a queueing network model for the CARAT system. In order to compare the overall system performance, measures such as transaction throughput, transaction response time, the probability a transaction is aborted, the total cpu utilization, and the total disk I/O rates were also measured.

Although the complexity of a distributed database system makes experimental comparisons of concurrency control and recovery algorithms difficult, we have succeeded in collecting experimental data and presenting qualitative explanations for the transaction processing performance of a system that implements two of the most common approaches, two-phase locking with before-image journaling (2PL/BI) and optimistic concurrency control with after-image journaling (OPT/AI). For our hardware environment, which was I/O bound, OPT/AI performed better than 2PL/BI on most of the workloads, especially in the cases with high transaction conflict rates. However, this may not be true for other workloads and hardware environments that have more or faster disks. We also showed that, in our two node distributed environment, the effects of distributed deadlock detection on system performance were not significant compared to that of concurrency control itself. This is primarily due to the high network bandwidth, the implementation of an efficient distributed deadlock detection algorithm, and the I/O boundedness of the workloads.

The two major restrictions of our initial study are that the measurements were limited to a two node system and there was only one disk for both the database and log at each node. We are now conducting experiments for other workloads (cpu bound as well as I/O bound, larger databases on more nodes) and other hardware environments (separate disks for log and database, different speed disks, separate disk controllers, local area network with up to five nodes) and other combinations and variations of concurrency control and recovery algorithms.

In order to perform the performance experiments and collect data, CARAT contains instrumentation software. The instrumentation software includes three phases. The first phase is an initialization phase that transfers database code images to nodes in the distributed system, and initiates CARAT and the on-line instrumentation processes. The second phase occurs at execution time and consists of a SPY program that displays high level information about the state of transactions in the entire distributed system, and a data collection program that records appropriate system and transaction data for post-test analysis. The third phase operates at the end of the experiment and collects the recorded data from each node into a control file and *cleans up* each site involved in the test. Plans are underway to significantly enhance the instrumentation capabilities of CARAT including entering post-test data into a relational database and enhancing the SPY program. There are many interesting open issues with regard to instrumenting distributed systems both for performance evaluation and debugging.

4 Analytic Modeling

Our analytic modeling results [JENQ86,JENQ87] differ from previous work in three major aspects. First, rather than modeling a locking scheme in a centralized database environment, we consider the effects of a two-phase locking scheme and a two-phase commit protocol for distributed transaction executions. In this respect, we have extended the centralized model used by Irani and Thomas to distributed environments. Second, our model is more general and integrated because the effects of journalling and rolling back transactions due to detected deadlocks are considered. We also take

into account both shared and exclusive locks. Third, in order to validate the model against performance measurements, the model closely reflects transaction execution in the CARAT testbed.

The queueing network model contains two levels. The high-level model, called the Site Processing Model, represents transaction execution at a single site of the distributed CARAT system. A distributed CARAT system is represented as a set of interacting Site Processing Models. The interactions of the Site Processing Models for distributed transactions are isolated in the arrival and processing of remote requests. The arrivals of remote requests from other sites are represented by a service center, NET, in the Site Processing Model. The remote requests of distributed transactions are modeled as a separate type of transaction, called a distributed slave. Since the communication delay for inter-site message passing depends on the characteristics of the underlying network and the loads from each site in the system, we employ a low-level model, called the Communication Network Model, to calculate the communication delay for inter-site messages. We used a simple Ethernet model as the Communication Network Model, since our performance measurements were carried out in an Ethernet environment. However, the characteristics of other local or wide area networks with different bandwidths can be captured in the low-level model.

We have successfully validated the queueing network model for a variety of workloads using the data obtained from performance measurements [JENQ87]. This modeling study does not answer all the important questions, but it establishes a framework for further modeling studies. Work is in progress to apply the same modeling framework to model the CARAT testbed system when a distributed version of optimistic concurrency control is used with after-image journaling. We are also investigating other modeling methodologies.

5 Algorithm Design

5.1 Distributed Deadlock Detection

Many algorithms have been proposed for detecting deadlocks in distributed database systems. The problem in a distributed database system is, in

essence, to find the cycles in a distributed graph where no single site knows the entire graph. Some algorithms detect deadlocks by first constructing and then finding cycles in the transaction "wait for" graph (a directed graph where nodes represent transactions and edges represent the wait-for relationships). Other algorithms use a probe technique. Probes are special messages used to detect deadlocks. Probes follow the edges of the wait-for graph to search for a cycle without constructing a separate representation of the graph.

The probe method is a very elegant approach and its proof of correctness is conceptually very simple. The disadvantage of the basic form of this approach is that after a deadlock is detected, the constituents of the deadlock cycle remain to be discovered. Chandy, Haas, and Misra suggest a method for determining this information but their method has some drawbacks: First, the number of messages required is greater than that required to detect the deadlock and therefore, its average case performance is poor. Second, it ultimately alerts all the transactions in the cycle without suggesting a method to resolve the cycle. Sinha and Natarajan proposed an algorithm based on a similar technique that has better performance. In their scheme the number of messages is reduced by assigning priorities to transactions and then only forwarding high priority messages. Also, in this scheme not all the transactions in the cycle are alerted. The highest priority transaction in a cycle detects the deadlock cycle. The probes carry additional information to resolve the deadlock.

In [CHOU86a,CHOU86b] we showed that in many situations Sinha and Natarajan's algorithm either fails to detect a deadlock cycle or detects a false deadlock. We then presented a modified probe algorithm based on priorities that corrects these problems. We observed that the major cause for detecting false deadlock or failing to detect some deadlocks is that the algorithm overlooks the possibility of transactions waiting transitively on a deadlock cycle. Also, the algorithm failed to account for those cases in which, after a transaction releases locks, new transactions which now acquire those locks may get involved in a deadlock. Although we have not formally proven that the modified algorithm works in all possible cases, extensive simulation evidence leads us to believe that it is correct. We have also used simulation to compare the performance of two distributed deadlock detection algorithms [CHOU86a,CHOU88].

5.2 Network Partitioning

We have also been interested in recovery from network partitions. Traditionally, strategies to allow a distributed database system to continue functioning in the face of network partitions can be classified as either *(a) deterministic*, which guarantee that conflicts will be avoided by limiting access (to a data item) to at most one partition, or *(b) optimistic*, which allow each partition to perform updates that might conflict with those occurring in another partition, under the assumption that the conflict rate will be small. Strategies can also be categorized with respect to knowledge of the partition. Some of them *avoid* the need for explicit knowledge of the partition, but as a result, performance may be degraded. Most of them *require* knowledge of the partition but have largely ignored the problem of detecting it.

Existing protocols which require partition knowledge have focused on what to do during the partition and at partition repair time. The exact steps necessary to *initiate* the protocol have not been defined. They have also assumed that each partition knows the identity of the nodes which are members of it, but little work has been done on how to acquire this knowledge. We have developed a consensus protocol which offers a solution to the problems of identifying both the *existence* of a partitioned network and the *constituents* of each partition [STAN87a].

In addition, we have developed two timestamp-based optimistic protocols for recovery from network partitions [GOOD86]. One does not rely on having partition information and operates optimistically *at all times* (rather than only during the time of a partition). The other relies on having partition information (e.g. knowing that a partition exists) and our plan is to integrate our consensus protocol with this recovery protocol and compare the resultant performance with that of the recovery protocol which does not make use of the partitioning information.

6 Database Cache

Database systems use a main memory buffer to hold a portion of the database that is in active use by transactions. Traditional database systems provide a private buffer for each process that is executing a transaction.

In contrast, the database cache serves as a shared buffer for all processes executing transactions against the database. This technique has the effect of reducing the number of database file I/O operations performed by a transaction. At the current level of technology, the time required for a file access operationis several orders of magnitude greater than that of a main memory access operation. Consequently, the use of a database cache results in faster transaction response time and a higher throughput rate.

We have implemented and evaluated a database cache scheme [NOTA87]. Two data structures are used to manage the database cache: the *cache table*, which keeps track of the state information of each page in the cache, and the *hash table*, which provides information to indicate which of the database pages are already in the cache. These data structures are shared by all the processes accessing the database and hence must only be modified within a critical section. Two addition data structures are used to manage the replacement of cache pages, a list of unfixed unmodified pages and a list of unfixed modified pages. Finally, each process executing a transaction manages a register lock table. This data structure contains the cache location and state of each page accessed by a transaction process. At the time of commit or rollback, the information is used to update the state of the cache.

A variety of experiments using synthetic transaction workloads were performed to evaluate the database cache implementation. The tests demonstrated that the shared cache reduced database file I/O. The results compared well with the predictions of a simple analytic model. Various extensions to this work are now being considered.

7 Transaction Processing in a Data Sharing Environment

There are two basic approaches for supporting access to a large database when multiple computer systems are coupled for transaction processing: the partitioned approach and the data sharing approach. In the partitioned approach, each system owns some of the disk storage devices and the database is partitioned among them. Transactions that need to access data in multiple partitions must execute on multiple systems. This ap-

proach is used by distributed database systems like IBM's R*. In contrast, with the data sharing approach all processors access a pool of disks that holds the entire database. Consequently, transactions execute completely on one processor.

Data sharing systems can be further classified into two types, depending on their processor and operating system architecture. The traditional tightly coupled multiprocessor approach is based on shared memory and a single operating system. Interprocessor communication is through shared memory, and processors also share access to disk storage devices. Due to memory contention and memory caching problems, the number of processors that can be effectively coupled for transaction processing applications has been found to be quite small. The loosely coupled multiprocessor approach uses separate processors and memories connected by a high-speed message-oriented protocol to access the disks that hold the shared database. Here there is one Front End Processor for handling all terminal communications, although there could be more, and several Transaction Processors for executing database transactions. The Disk Subsystem is used to store the database. Duplicated components may be used in the implementation to increase data availability. The High-Speed Interconnect supports multiple independent paths to increase availability and performance. DEC's VAXcluster architecture is one example of a loosely coupled data sharing system.

7.1 Database Buffering and Concurrency Control

In high performance transaction systems, performance is greatly influenced by the effective access time to disk storage where the data is stored. While CPU instruction rates continue to improve significantly, the speed of large capacity moving head disks has not changed much. Therefore, the gap between CPU and disk speed is becoming larger. Main memory buffering of a portion of the database is commonly used in centralized and partitioned database systems. In the loosely coupled data sharing environment, main memory buffering of data at each system is also necessary to reduce the effective access time and to reduce the I/O traffic between the processors and the shared disk storage system. However, a new problem arises due to data buffering in data sharing systems. Since each processor has its

own buffer in its own memory, care must be taken to insure that the data seen by transactions running in different processors is consistent. This problem is similar to the multicache coherence problem in shared memory multiprocessors but there are two major differences:

- Location of buffer in memory hierarchy. In tightly coupled shared memory multiprocessors, the private memory buffer is commonly called cache memory and it is used to hold part of the instructions and data from main memory. The cache is usually small in comparison to main memory and it is implemented using very high speed circuit technology. In loosely coupled data sharing systems for transaction processing, a main memory database buffer is used to hold a portion of the disk database. Therefore, the buffer in the data sharing system is functionally similar to the cache memory in a tightly coupled multiprocessor system but is at a lower level in the memory hierarchy.

- Frequency of coherence. In the case of shared memory multiprocessor caches, it must appear to the application program as if the processor is fetching instructions and data from shared memory. An update to a shared variable by any processor must be seen, if required, at the next memory cycle by all processors. Therefore, data in the caches must be consistent at each memory cycle. In transaction processing systems, due to transaction atomicity requirements, all update operations on data within a transaction are isolated (usually by means of locks) from other transactions. A concurrency control mechanism, like two-phase locking, is responsible for supporting this behavior. Therefore, since the concurrency control mechanism prohibits the data from being simultaneously updated by more than one transaction, database buffer coherence is only necessary at the end of transactions. Since transaction rates are many orders of magnitude smaller than memory cycle rates, the database buffer coherence problem in loosley coupled data sharing systems may be more manageable.

In cache systems, there are two common update policies for shared memory: write-back and write-through. In the context of buffer management for data sharing systems, these policies would have the following general definitions:

- Buffer write-through policy. Data that is updated in the buffer and is part of a committed transaction is reflected on the disk version of the database at the end of transaction. This means that the database data on the disk is consistent with committed data in the buffer.

- Buffer write-back policy. Data that is updated in the buffer and is part of a committed transaction is kept in the database buffer and is not necessarily reflected on the disk version of the database at the end of transaction. This means that the database data on the disk is not necessarily consistent with committed data.

We have investigated both of these policies using a detailed simulation model of a data sharing system[IISU88a].

Another problem arises in the context of concurrency control. In data sharing systems, concurrency control is usually based on locking and the lock manager is a potential performance bottleneck. The lock manager may have a centralized or a distributed implementation, but it must support systemwide synchronization. In the centralized approach there is one lock manager and all lock requests are routed to the site of the lock manager.

Another approach is to distribute the global locking mechanism among all transaction processors so that the lock processing overhead is shared by all processors. This is the approach taken by DEC's VAXcluster. In VAXcluster systems the lock responsibility for a data item is dynamically assigned to a master node and lock requests issued from the master can be handled locally without network messages. However, this makes the locking performance sensitive to the processor on which the transaction is assigned. Good distributed locking perfromance requires good locality of data reference and dynamic assignment of locking responsibility. As succeeding transactions on the same processor reference smililar parts of the database, the processor is granted the temporary right to acquire and release the lock locally without network communication overhead. This temporary right can be called back if it is blocking some other transaction processor's access.

Another problem associated with the concurrency control mechanism is the recovery of lock information after a system failure. In the case of a centralized locking mechanism, the failure of a processor other than the lock manager has no effect on the conccurrency control information, since

all the lock information is stored at the lock manager's site. In the case of lock manager failure, if the current lock information is not replicated, the operation of the whole system must be suspended. If the concurrency control implementation is distributed among multiple lock managers and each is supported by a different processor, then upon any lock processor failure, some lock information will be lost. This is especially true in the distributed locking mechanism like the one implemented in DEC's VAXcluster system. When a processor failure occurs, all the surviving processors must cooperate to reconstruct the volatile locking information of the failed processor. During this recovery time, the system is unavailable for normal processing. In the case of two-level locking mechanisms in which there is a global coordinator, the effect of one transaction processor failure can be isolated from other transaction processors. But the failure of the global coordinator will result in the total loss of the volatile locking information. This is similar to the failure of the lock manager in the centralized locking implementation. The volatile locking information that is lost must be reconstructed. In the case of solutions based on distributing the locking over all hosts, as the number of hosts increases so does the probability that some failure will occur. Therefore, the system availability is challenged by the efficiency and isolation of the recovery action.

We have studied protocols that integrate buffer strategies and concurrency control policies [HSU88a]. We are now attempting to integrate such solutions with database load balancing.

7.2 Load Balancing

In the data sharing environment since the database is shared by all systems at the disk storage I/O level, transactions can be executed completely on any transaction processor in the system. A performance modeling study by Reuter showed that load control and load balancing in such an environment can be effective and is also critical for good performance. Transaction load control and load balancing can be implemented by the front end processor which can withhold or assign a new transaction to a specific transaction processor based on the state of the system, the transaction type, and the control policy. Once a new transaction has been assigned to a transaction processor, we assume that it executes to completion (or aborts) on that

processor. The goals of the control policies are:

- Load Control. Control the multiprogramming level, i.e., the number of active transactions on each transaction processor to utilize all available resources without overloading them.

- Load Balancing. Assign new transactions to specific transaction processors to guarantee response time limits and maximize overall transaction throughput.

However, high CPU and I/O utilization is not effective when a large percentage of the processing is consumed by transactions that are aborting due to data contention or by overhead due to buffer invalidation and concurrency control.

In order to obtain the maximum performance benefit from the buffers in a multibuffer, multiprocessor data sharing system, we should assign transactions to specific transaction processors to maximize the database buffer hit probability. In other words, transactions should frequently find the data they need to access in the buffer of their system. When this can be achieved, the transactions and data naturally partition. In the ideal case the database is logically partitioned among the transaction processors, the intersection of the contents of all buffers is minimal, and the transaction processor associated with each partition can handle the load.

To effectively use a data sharing system for transaction processing, it must be possible to partition the transaction workload as discussed above. Due to the dynamic change in the function and data requirements of the transactions that form the system workload, the optimum partition is not static.

Some of the problems that must be solved to achieve the potential advantages of loosely coupled data sharing systems for transaction processing are:

- Reduce database I/O through improved database buffering schemes.

- Reduce concurrency control overhead and delay through improved lock management schemes.

- Reduce impact of processor failures on overall system through improved recovery management schemes.

- Reduce overhead and improve transaction throughput performance through improved transaction load control and load balancing schemes.

We are investigating current and new solutions to each of these problems and intend to evaluate their impact on system performance.

8 Real–Time Transactions

Next generation real–time systems will require greater *flexibility* and *predictability* than is commonly found in today's systems. These future systems include the space station, integrated vision/robotics/AI systems, collections of humans/robots coordinating to achieve common objectives (usually in hazardous environments such as undersea exploration or chemical plants), and various command and control applications. The complexity of such systems due to timing constraints, concurrency, and distribution is high. It is accepted that the synchronization, failure atomicity, and permanence properties of transactions aid in the development of distributed systems. However, little work has been done in exploiting transactions in a real–time context. We have been attempting to categorize real–time data into classes depending on their time, synchronization, atomicity, and permanence properties. Then, using the semantics of the data and the applications, we are developing special, tailored, real–time transactions that only supply the minimal properties necessary for that class. This reduces the system overhead in supporting access to various types of data. The eventual goal is to verify that timing requirements can be met. In this work aspect of our work we have developed two real–time access control protocols and a real–time stream primitive [STAN87b]. We intend to evaluate the access control primitives on the CARAT testbed.

References

[CHOU86a] A. N. Choudhary, "Two Distributed Deadlock Detection Algorithms and Their Performance", M.S. Thesis, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, February 1986.

[CHOU86b] A. N. Choudhary, W. H. Kohler, J. S. Stankovic, and D. Towsley, "A Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution," **Proc 7th Int. Conf. on Distributed Computer Systems**, October 1987, and to appear in **IEEE Transactions on Software Engineering.**

[CHOU88] A. N. Choudhary, "Two Distributed Deadlock Detection Algorithms and Their Performance Comparison", in preparation, January 1988.

[DAN88] A. Dan, D. Towsley, and W. Kohler, "Modeling the Effects of Data and Resource Contention on the Performance of Optimistic Concurrency Control Protocols," **Proceedings Fourth International Conference on Data Engineering,** Los Angles, CA, February 1988.

[GOOD86] A. Goodman, "Recovery Protocols in the Presence of Network Partitions," Ph.D. Thesis Proposal, Department of Computer and Information Science, University of Massachusetts, Amherst, December 1986.

[HSU88a] Y. P. Hsu, "Performance Evaluation of Data Sharing Transaction Processing Systems," M.S. Thesis, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, February 1988.

[HSU88b] Y. P. Hsu, W. Kohler, and J. Stankovic, "Buffer Coherency Control Protocols for Data Sharing Transaction Processing Systems," Technical Report, in preparation.

[JENQ86] B. P. Jenq, "Performance Measurement, Modelling, and Evaluation of Integrated Concurrency Control and Recovery Algorithms in Distributed Database Systems", Ph.D. Dissertation, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, 1986.

[JENQ87] B. P. Jenq, W. H. Kohler, and D. Towsley, "A Queueing Network Model for a Distributed Database Testbed System",

Proceedings Third International Conference on Data Engineering, Los Angeles, California, February 1987, also to appear in **IEEE Transactions on Software Engineering**.

[KOHL86a] W. H.Kohler and B. P. Jenq, "Performance Evaluation of Integrated Concurrency Control and Recovery Algorithms Using a Distributed Transaction Processing Testbed," **Proceedings Sixth International Conference on Distributed Computing Systems**, Cambridge, MA, May 1986, pages 130-139.

[KOHL86b] W. H. Kohler and B. P. Jenq, "CARAT: a Distributed Testbed for the Performance Evaluation of Integrated Concurrency Control and Recovery Algorithms," **Proceedings 1986 Fall Joint Computer Conference**, Dallas, Texas, November 1986, pages 1169-1178.

[NOTA87] R. Notaney, "Design, Implementation, and Evaluation of A Database Cache Scheme," Maters Thesis, Department of Electrical and Computer Enginerring, University of Massachusetts, Amherst, September 1987.

[NOTA88] R. Notaney, W. Kohler, D. Towsley, "Design and Implementation of a Database Cache Scheme," Technical Report, in preparation, January 1988.

[STAN85] J. A. Stankovic, K. Ramamritham, and W. H. Kohler, "A Review of Current Research and Critical Issues in Distributed System Software," **IEEE Distributed Processing Technical Committee Newsletter**, Vol. 7, No. 1, March 1985, pages 14-47, and in **Concurrency and Recovery in Distributed Systems**, Bharat Bargava, Editor, Van Nostrand, 1987.

[STAN87a] J. A. Stankovic, "A Consensus Protocol for Detecting Database Partitions," in preparation.

[STAN87b] J. A. Stankovic, and W. Zhao, "On Real-Time Transactions," submitted to SIGMOD RECORD, Special Issue on Real-Time Databases, November 1987.