DRDB: A Distributed Real-Time Database Server for High-Assurance Time-Critical Applications

Sang H. Son, Robert C. Beckinger, and David A. Baker
Department of Computer Science
University of Virginia
Charlottesville, VA 22903, USA

Abstract

Many real-time database systems are now being used in safety-critical applications, in which human lives or expensive machinery may be at stake. Transactions in real-time databases should be scheduled considering both data consistency and timing constraints. In addition, a real-time database must adapt to changes in the operating environment and guarantee the completion of critical tasks. The effects of scheduling decisions and concurrency control mechanisms for real-time database systems have typically been demonstrated in a simulated environment. In this paper we present a functional real-time database server, called DRDB, which provides an operational platform for research in distributed real-time database issues.

1.Introduction

Real-time database systems (RTDBS) are essential for applications that are both data intensive and subject to real-time constraints, such as defense systems, industrial automation, aerospace, and network management. Appropriate methods and techniques for designing and implementing database systems that take timing constraints into account are playing an ever increasing role in determining the success or failure of real-time systems. In recent workshops [RTAS, RTDB], developers of real-time systems have pointed to the need for basic research in database systems that satisfy timing constraint requirements in collecting, updating, and retrieving shared data.

The distinct feature of RTDBS, as compared to traditional databases, is the requirement to satisfy the timing requirements associated with transactions. The correctness of the system depends not only on the logical results but also on the time within which the results are produced. Transactions must be scheduled in such a way that they can be completed before their corresponding deadline expires.Most database systems are not designed for real-time applications and lack features required supporting real-time transactions. Very few conventional database systems allow users to specify temporal constraints or ensure that the system meets those constraints. In RTDBS, it is not always possible to guarantee all temporal constraints because of the unpredictable data accesses, so the system must strive to minimize the number of constraints which are violated [BLS97, Kim96].

Applying real-time principles to a distributed environment complicates the problem [Son96]. The major issue to consider is the communication between the database servers and clients in a distributed system. The added communication cost makes the timing property of requests to remote servers more unpredictable. Therefore an operation on one relation at a local site will take a shorter period of time than the same operation on a relation located at a remote site.

The design and evaluation of RTDBS present challenging problems. In this paper we describe an operational distributed real-time database system, called DRDB. In addition, we examine issues involving the development of credible run-time estimates for use in real-time database scheduling decisions and the use of timing requirements in performing database operations.

2. Issues in Temporal Functionality

One of the major goals in designing RTDBS is to minimize the probability of transactions failing to meet their respective deadlines. Various approaches have been investigated to develop database systems to achieve this goal [Best96, Lam97, Lehr95, Son92]. They attempt to make scheduling decisions based mainly on transaction attributes such as priority, release time and deadline. These transaction attributes are critical pieces in the scheduling puzzle, but they are not the only attributes available for use in solving the problem. One key attribute absent from most scheduling decisions is a viable transaction *run-time estimate*. Numerous research efforts have explored the possibility of using run-time estimates in the scheduling decision process. Run-time estimates have been used in workload policies, priority assignment policies, conflict resolution policies, and I/O scheduling policies [Abb92]. These run-time estimates have typically been model-driven. The results derived have shown that run-time estimates are a credible option for use in scheduling decisions. However, the derivation and use of run-time estimates in a functional real-time database has not been explored extensively. If the scheduler can be provided with an estimate of transaction execution time, that information can be used in determining which transaction is closest to missing a deadline, and hence should be given higher priority, or which transaction can be delayed without risking violation of their timing constraints. In addition, run-time estimates can be used by the scheduler to initially screen transactions to determine eligibility. All transactions with feasible deadlines (release time plus run-time estimate is less than deadline) remain in the system, while all ineligible transactions are aborted.

This work was supported in part by ONR and NSA.

Proceedings of the COMPSAC '97 - 21st International Computer Software and Applications Conference
0730-3157/97 $10.00 © 1997 IEEE

Often a significant portion of a real-time database is highly perishable in the sense that it has value only if it is used in time. In addition to deadlines, therefore, other kinds of temporal information should be associated with data as well as transactions in RTDBS. For example, each sensor input could be indexed by the time at which it was taken. Once entered in the database, data may become out-of-date if it is not updated within a certain period of time. To quantify this notion of *age*, data may be associated with a valid interval. The valid interval indicates the time interval after the most recent updating of a data object during which a transaction may access it with 100% degree of accuracy. What occurs when a transaction attempts to access a data object outside of its valid interval is dependent upon the semantics of data objects and the particular implementation. For some data objects, for instance, reading it out of its valid interval would result in 0% accurate values. In general, each data object can be associated with a *validity curve* that represents its degree of validity with respect to the time elapsed after the data object was last modified. The system can compute the validity of data objects at the given time, provided the time of last modification and its validity curve.

A real-time transaction should include its temporal consistency requirement which specifies the validity of data values accessed by the transaction. For example, if the temporal consistency requirement is 10, it indicates that data objects accessed by the transaction cannot be *older* than 10 time units relative to the start time of the transaction. This temporal consistency requirement can be specified as either hard or soft, just as deadlines are. If it is hard, an attempt to read an invalid data object (i.e., out of its valid interval) will cause the transaction to be aborted.

While a deadline can be thought of as providing a time interval as a constraint in the future, temporal consistency specifies a temporal window as a constraint in the past. As long as the temporal consistency requirement of a transaction can be satisfied, the system can be able to provide an answer using available (may not be up-to-date) information. The answer may change as valid intervals change with time. In a distributed database system, sensor readings may not be reflected to the database at the same time, and may not be reflected consistently due to the delays in processing and communication. A temporal data model for real-time database systems must therefore be able to accommodate the information that is partial and out-of-date. One of the aspects that distinguishes a temporal data model for a real-time database systems from that of conventional database systems is that values in a real-time database system are not necessarily correct all the time, and hence the system must be selective in interpreting data values [Ozso95].

3.Implementation of DRDB

DRDB is designed along the client-server paradigm. It has a multiple-threaded servers that are capable of accepting requests from multiple clients. DRDB was designed with the goal of providing a temporal platform for conducting research on real-time database issues. This is a major and natural step forward from performance analysis conducted in a simulated database environment. Simulated environments are not a substitute for functional systems. They fail to account for all factors found in an operational system, and tend to be more subjective in the sense that system parameters can be readily modified. An operational system cannot be modified to fit the real-time mechanism being analyzed. The results derived from an operational real-time database system provide us with a set of more realistic performance measurements.

The DRDB server is the heart of the database management system. The server contains an infinite loop that accepts high-level database requests from multiple clients. The requests come in as packets. The DRDB system provides two different types of packets: *call* packets and *return* packets. The call packet is created by the client and is the database transaction. The call packet contains all the information that the server needs to carry out the desired database access operation, including the timing constraint and temporal consistency specifications associated with the transaction. A different timing constraint can be specified for each transaction submitted, or the client can use the default timing constraint previously established. The DRDB client thread passes the call packet forward to the DRDB server.

Once a command is transferred to the server, it is passed to the DRDB scheduler running on that server. The DRDB scheduler uses a run-time estimate evaluation technique to determine if the system can provide the client with the information requested within the timing constraint specified. The DRDB server spins-off a separate DRDB thread to execute the transaction if the deadline is feasible. The client will be informed, and no thread spun-off if the deadline can not be met. The thread executes until completion and then forwards the call packet back to the client or the requesting server which forwarded the command to this server. A transaction is not preempted by the DRDB thread even if the deadline is missed. Along with the results of the transaction, the client will be informed that the deadline is missed, along with the results of the transaction.

Since DRDB is a distributed database, a client can issue a command to manipulate data at a remote site. As part of the preprocessing, the server first examines the command and identifies all relations needed for processing it. The server then looks in the relation table which maps all relations in the database to the site where they reside. The server forwards the command if the specified relation in the command resides at a remote site.

A relation table for a server is created when that server is initiated. The first operation a new server performs is reading the relation files in its own directory and storing these relations in the relation table. The server then reads a global address file which contains the name and port numbers of all currently running servers. The new server uses these port numbers to notify each server it has begun execution. The new server sends its list of relations so that other servers can update their relation tables. Other servers send back their lists of relations to the new server to be included in its relation table. The new server is then ready to accept requests from clients.

0730-3157/97 $10.00 © 1997 IEEE

DRDB also associates a temporal *'valid time'* attribute with each relation. This is inherent to the system, requiring no client involvement. The temporal attribute is attached to each tuple of a relation and is comparable to a timestamp that represents the *valid time* that the stored information models reality. The client cannot set or modify the values associated with the *valid time* attribute. However, this attribute can be manipulated for use in specifying transaction temporal consistency requirements. For example,

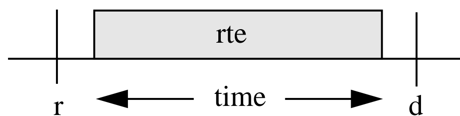
select trk_num from trackfile where *valid time* < 1

will return only the track numbers (trk_num) of tuples inserted or updated in the database relation (trackfile) within the last second of the transaction release time. DRDB also allows users of the system to manipulate the valid time attribute in output displays and in creating and manipulating relations that are similar to those found in *historical* temporal database systems [Ozso95]. A relation without the temporal attribute *valid time* attribute can be formed by *projecting* or *selecting* attributes other than *valid time* into a new relation. The DRDB system processes such relations without attaching any temporal meaning to them.

The DRDB system employs a *strict* two-phase locking (2PL) protocol for concurrency control [Ber87]. The *strict* locking protocol was selected for concurrency control because of its prevalence in commercial systems and because of its desirable characteristic of being recoverable and avoiding cascaded aborts.

DRDB allows a user to enter multiple commands and have the results for all these commands returned together. The transaction facility therefore gives the user the ability to treat a group of commands as one atomic action. Adding the transaction facility requires the DRDB server to hold all command results until all commands of a transaction are completed. We follow the two-phase commit protocol for atomic commitment [Ber87].

A transaction in DRDB is characterized by its timing constraints and its computation requirements. The timing constraints are a release time *'r'* and deadline *'d'*. The release time is the time associated with the transmittal of the transaction by a client site. A computation requirement is represented by a run-time estimate *'rte'* which approximates the amount of computation, IO, and communication costs associated with processing a transaction. The deadline corresponds to the client-specified timing constraint.



The release time and deadline are known to the DRDB scheduler when a transaction arrives. The computation requirements are calculated based on the operation being performed and the physical characteristics of the data involved. This information is made available prior to the scheduling decision being made. We think it is viable to estimate the execution time of a transaction without having prior knowledge of the exact data access pattern of a transaction.

The goal of our system is to minimize the number of transactions that miss their deadlines, i.e., that finish after time *'d'*. If transactions can miss deadlines, one must address the issue as to what happens to transactions that have already missed their deadlines but have not yet finished. There are two alternatives. One is to assume that a transaction that has missed its deadline can be aborted. This may be reasonable where the value of a transaction is dependent on the timeliness of the return response. For example, suppose that a transaction is submitted to update the ballistic path of a projectile based on a radar sensing. If the deadline is missed, it may be more desirable not to perform the operation of updating the ballistic path, but instead to re-submit the update request based on a newer sensor reading. The conditions that led to the triggering of the transaction may have changed. The initiator of the transaction may be better served if the transaction is re-submitted.

A second option is to assume that all transactions must eventually be completed, regardless of whether they have missed their deadlines. This may be a correct approach in an application such as banking where a customer would rather have his financial transaction done late rather than not at all. If the decision is made to process the transaction, there is still the issue of the priority of tardy transactions with respect to other transactions in the system. Transactions which cannot meet their deadlines could receive a higher priority as their lateness increases, or they could be postponed to a later more convenient time.

The DRDB implementation decision was a combination of the two approaches. When a transaction enters the system, a determination is made as to whether a transaction can be executed within the temporal constraint associated with it. If the transaction cannot meet its deadline, it is aborted. This has the nice property of not allowing computation time to be expended on transactions which cannot meet their deadlines, even with the best effort. To allow such transactions into the system can adversely affect overall system performance especially during high load periods. Aborting a few late transactions helps all other transactions meet their deadlines, by eliminating the competition for resources by tardy transactions. Once a transaction has been accepted for processing, it is executed to completion, regardless as to whether or not a deadline has been met. This approach was adopted as a means of validating the run-time estimates derived by the scheduler.

4.Scheduling Policies and Run-Time Estimates

The DRDB scheduling algorithms have three components: a policy to determine which tasks are eligible for service, a policy for assigning priorities to tasks, and a conflict resolution policy. Only the first two policies are explored in the remainder of this paper.

4.1.Scheduling Policies

The DRDB scheduler is invoked whenever a transaction enters the system or terminates. The scheduler can also be invoked to resolve contention (for either

0730-3157/97 $10.00 © 1997 IEEE

the CPU or data) when conflicts occur between transactions. The first task of the scheduler is to divide the set of ready transactions into two categories, those transactions that are capable of meeting their temporal constraints (eligible) and those that cannot meet their temporal constraints (ineligible). All ineligible transactions are aborted and the client is informed of the decision. This approach differs from the *non-tardy* policy [Abb92] which accepts transactions that are currently not late, but may be in a position where it is physically impossible to make their deadlines. Only those transactions with *feasible deadlines* are considered to be eligible. A transaction has a feasible deadline if its deadline is less than or equal to the current time plus its run-time estimate:

current time (**t**) + run-time estimate (**rte**) ≤ deadline (**d**)

This policy can be adapted to account for the amount of service time a transaction has already received. The modified policy would be as follows:

current time (**t**) + **rte** - service time(**p**) ≤ deadline (**d**)

where **p** equals the amount of service time a transaction has accumulated. This modified policy allows transactions to be screened for eligibility during the course of execution. Transactions that have been blocked, due to either data or CPU contention, could be re-evaluated to determine if they are still capable of meeting their temporal constraint. Note that the success of both of these policies is contingent on the accuracy of the run-time estimate. Erroneous run-time estimates which over-estimate the actual computational requirements will cause transactions to be aborted needlessly. Low estimates can degrade system performance by allowing transactions, which in reality cannot meet temporal constraints, to compete for system resources among transactions which are trying to meet deadlines.

There are many ways for assigning priorities to real-time tasks. Three policies extensively studied by earlier researchers include *First Come First Serve* (FCFS), *Earliest Deadline* (ED) and *Least Slack*.(LS) [Abb92]. The primary weakness of FCFS is that it does not make use of deadline information. It discriminates against a newly arrived task with an urgent deadline in favor of an older task which may not have such an urgent deadline. The ED policy has shown itself to be effective in certain applications, but it fails to take into account the run-time estimates. The LS priority assignment policy was adopted for DRDB. The slack time for a transaction is an estimate of how long we can delay the execution of a transaction and still meet its deadline. It is computed by subtracting the current time plus the run-time estimate from the deadline of the transaction:

Slack (**s**) = deadline (**d**) - (current time (**t**) + **rte**)

The smaller the slack, the higher the priority. A negative slack time is an indication that it is physically impossible for the transaction to meet its deadline. This priority assignment policy does not take the amount of prior service time into account. The assignment of priority is static, occurring once when the transaction enters the system. The priority computed at that time remains with the transaction throughout its execution life. A continuous LS policy could be used which does take service time into account. The continuous evaluation of priorities causes the LS of all active transaction to be recomputed whenever there is contention for processor or data.

A negative slack time could occur if a transaction has already missed its deadline or is about to miss its deadline. The possibility of a negative slack time does not exist if a feasible deadline eligibility screening policy is implemented, and the LS priority assignment policy is static. The initial screening conducted to determine eligibility will eliminate any transaction which cannot physically meet their deadlines and the static LS priority will prevent the slack associated with an eligible transaction from ever becoming negative. The current DRDB version uses static LS as the means of assigning priorities to transactions for scheduling, in an attempt to expedite those transactions which can least afford to be delayed.

4.2.Run-Time Estimates

Conventional real-time systems typically deal with processes that have predictable resource requirements. These predictable requirements allow for a static evaluation of computation costs. Real-time databases normally deal with transactions which have unpredictable resource requirements. The random nature of such data accesses complicates the scheduling process in real-time database systems. A considerable amount of research effort has been focused on real-time database scheduling issues and the use of run-time estimates. The use of run-time estimates in scheduling decisions have been examined in workload screening, priority assignment, conflict resolution and IO scheduling policies. The results from the research conducted to date have indicated that run-time estimates are a viable option for improving scheduling decisions [Abb92, Sha91, Son92]. The fact that critical information such as run-time costs can improve scheduling decisions and subsequently overall system performance is quite intuitive. However, the derivation of run-time estimates is not straightforward, and have typically been derived from simulation models. The derivation and use of run-time estimates in a functional real-time database system has not been appropriately explored.

One of the goals in the design of DRDB was to derive credible run-time estimates and to integrate those estimates in scheduling decisions. The approach we used was to exploit the physical characteristics of the data (such as attribute types, number of attributes in a relation, and the numbers of tuples in a relation) being manipulated, along with the type of database operation being performed (such as union, set difference and project), in an attempt to derive credible run-time estimates. While the arrival and types of transactions entering the system and the data which they access may be random, the computation steps involved in providing the appropriate response are not unpredictable. The steps required to execute any DRDB command is static in nature, and in a simplified outlook, only the number of iterations involved is dynamic.

The dynamic nature of the computation is dependent on the number and types of attributes involved,

0730-3157/97 $10.00 © 1997 IEEE

along with the number of tuples which constitute a relation. For example, the run-time cost for selecting values from a relation consisting of only a single tuple is minimal. It consists of basic start-up costs (such as transmitting the command, preprocessing, opening of relations, and reading in the data from disk), the actual computation cost in selecting that single value, and basic terminating operations (such as providing the transaction results). The run-time cost for selecting the same set of values from a relation of five hundred tuples entails the same basic costs associated with opening and closing operations for a single tuple relation, only the computation costs increase in relation to the number of tuples that have to be processed.

Other factors such as system load and data conflicts do not affect the run-time costs associated with a given transaction. Such factors only increase the competition for system resources, such as the CPU and IO access. For example, given the cost for selecting values from a given relation is '2' time units. If half that time is consumed, and that select operation is subsequently blocked by a higher priority transaction whether it be for CPU or data contention reasons, it will still require '1' time unit to complete once it becomes unblocked.

With this approach, we ran numerous performance measurements tests to capture the run-time costs. The results indicate that viable run-time estimates could be derived based on the physical characteristics of the data being manipulated and the operation being performed. The results which follow are a small extract from those numerous run-time cost analysis experiments.

4.3.Performance Results

The results of run-time estimate performance measurements for four basic DRDB commands (project, select, union, average) are given in Figure 1. The x-axis of Figure 1 is the total number of tuples processed by the operation. The y-axis is the total elapsed time from the start of the operation until the final result is received at the client node. The performance measurements were conducted in an attempt to isolate the cost factors attributable to the operations performed, and the size of the data processed (measured by the number of tuples in the relations). The operations were initiated from a separate client node, transmitted to the server node, and the appropriate results returned back to the client. The run-time costs account for activities from the initiation of the operation to the receipt of the appropriate result. The results shown are based on 200 performance measurements for each of the operations and relation sizes shown. The large sample measurement size was required to validate the results produced.

The results show that the project and select operation run-time costs grow in a linear fashion in relation to the size of the data being processed. The union operation run-time cost grows exponentially in relation to the size of the data being processed. The run-time cost is the mean of the 200 performance measurements. There was minimal deviation between the mean run-time cost and the performance measurements used in deriving the mean, usually with 90% of the performance measurements falling within ±10% of the mean. The deviation which did occur between measurements can be attributed to the limited clock granularity of the hardware involved, and to unpredictable behavior of the underlying operating system.

Our run-time cost results did show that run-time estimates could be derived not only based on the database operation being performed and relation size, but also on the number and types of attributes which make-up relations. However, the results also showed that such derived run-time estimates were heuristic in nature, and that no guarantee could be made that a given transaction's actual run-time cost would be as estimated. One of the primary contributing factors was the support of the underlying operating system. However, it is still possible to establish functions which generate acceptably accurate run-time estimates based on the physical characteristics of the data and the operations being executed, and that is what is implemented in the DRDB system.

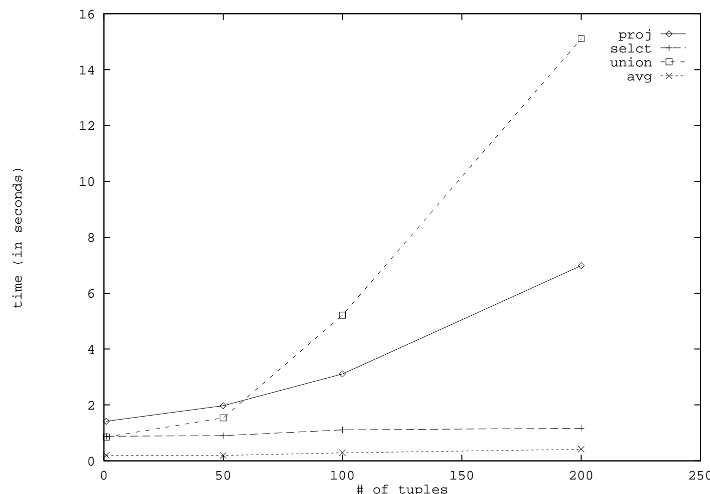


Figure 1: DRDB run-time costs for local operations

Proceedings of the COMPSAC '97 - 21st International Computer Software and Applications Conference
0730-3157/97 $10.00 © 1997 IEEE

The DRDB system maintains data on the physical characteristics of the relations in the database. When the scheduler is invoked it extracts the physical characteristics data for the relations being processed by a given transaction. This information is used in conjunction with the operation being performed to derive a run-time estimate. The run-time estimate is subsequently used in system scheduling decisions. While no guarantee can be made for a given transaction, it is possible to state that a given percentage of transactions can complete within the run-time estimate generated by the system. Additionally, it can be stated that the run-time estimates generated will be within a given percentage of the actual run-time costs. For example, raising the system generated run-time estimates by 10% resulted in approximately 90% of the transactions accepted for processing by DRDB having actual run-time costs within the system generated values. The down-side of raising the estimate is that some transactions, whose actual run-time cost is below the system generated estimate, may be needlessly aborted. The percentage of these depends on the tightness of the temporal deadlines attached to the transactions.

5.Conclusions

A real-time database server is one of the critical components of real-time systems, in which tasks are associated with deadlines and a significant portion of data is highly perishable in the sense that it has value to the system only if it is used quickly. To satisfy the timing requirements, transactions must be scheduled considering not only the consistency requirements but also their temporal constraints. In addition, the system should be predictable, such that the possibility of missing a deadline for a given transaction can be determined prior to the execution of that transaction or before that transaction's deadline expires.

In this paper, we have presented a relational database server which possesses temporal functionality, developed for investigating real-time database issues. Since the characteristics of a real-time database manager are distinct from conventional database managers, there are different issues to be considered in developing a real-time database server. For example, the use of run-time estimates in scheduling policies, and the ability to place temporal consistency constraints on database operations are important in real-time databases. DRDB was designed with the goal of providing an operational platform for conducting research on real-time database issues. Previous studies using simulated environments have provided valuable information with respect to real-time database issues. However, performance results in some of the simulated studies are sometimes contradictory with each other since they made different assumptions about system environments [Best96]. We believe that an operational environment for investigating real-time database issues will eliminate some of the problems associated with simulated systems and provide valuable and applicable insights to real-time database issues.

The DRDB system provides a foundation for studying real-time database issues in an operational environment. The results achieved in deriving and applying heuristic run-time estimates and the ability to attach temporal consistency specifications are promising. However, there remains many technical issues associated with real-time databases that need further investigation. We plan to work on analyzing various conflict resolution mechanisms and improving the temporal functionality.

REFERENCES

[Abb92] R.Abbott, and H.Garcia-Molina, "Scheduling Real-time Transactions: a Performance Evaluation," ACM Trans. on Database Systems, vol. 17, no. 3, Sept. 1992, pp 513-560.

[Ber87] P.A.Bernstein, V.Hadzilacos, and N.Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley Publishing Co., 1987.

[BLS97] A. Bestavros, K. Lin, and S. H. Son, "Real-Time Database Systems: Issues and Applications," Kluwer Academic Publishers, 1997.

[Best96] A. Bestavros, "Advances in Real-Time Database Systems Research," ACM SIGMOD Record, vol. 25, no. 1, March 1996.

[Kim96] Y. Kim and S. H. Son, "Supporting Predictability in Real-Time Database Systems," IEEE Real-Time Technology and Applications Symposium (RTAS'96), Boston, MA, June 1996, pp 38-48.

[Lam97] K. Lam, S. H. Son, and S. Hung, "A Priority Ceiling Protocol with Dynamic Adjustment of Serialization Order," IEEE Conference on Data Engineering, Birmingham, UK, April 1997.

[Lehr95] M. Lehr, Y. Kim, and S. H. Son, "Managing Contention and Timing Constraints in a Real-Time Database System," 16th IEEE Real-Time Systems Symposium, Pisa, Italy, Dec. 1995, pp 332-341.

[Ozso95] G. Ozsoyoglu and R. Snodgrass, "Temporal and Real-Time Databases: A Survey," IEEE Trans. on Knowledge and Data Eng., August 1995, pp 513-532.

[RTAS] IEEE Real-Time Technology and Applications Symposium, Boston, MA, June 1996.

[RTDB] First Workshop on Real-Time Database Systems, Newport Beach, CA, Feb. 1996.

[Sha91] L.Sha, R.Rajkumar, S. H. Son, and C. Chang, "A Real-Time Locking Protocol," IEEE Transactions on Computers, vol. 40, no. 7, July 1991, pp 782-800.

[Son92] S. H. Son, J. Lee, and Y. Lin, "Hybrid Protocols using Dynamic Adjustment of Serialization Order for Real-Time Concurrency Control," Journal of Real-Time Systems, vol. 4, Sept. 1992, pp 269-276.

[Son96] Son, S. H., F. Zhang, and B. Hwang, "Concurrency Control for Replicated Data in Distributed Real-Time Systems," Journal of Database Management, Special Issue on Real-time Database Systems: Theory and Practice, vol. 7, no. 2, pp 12-23, March 1996.

Proceedings of the COMPSAC '97 - 21st International Computer Software and Applications Conference
0730-3157/97 $10.00 © 1997 IEEE