

12

AD-A257 773



**Annual Report
Grant No. N00014-91-J-1102**

October 1, 1991 - September 30, 1992

THE STARLITE PROJECT - PROTOTYPING REAL-TIME SOFTWARE

Submitted to:

**Office of Naval Research
Chief of Naval Research
Code 1267/Annual Report
Ballston Tower One
Room 528
800 N. Quincy Street
Arlington, VA 22217-5660**

**DTIC
ELECTE
NOV 6 1992
S C D**

Attention:

**Dr. James G. Smith, Program Manager
Information Systems**

Submitted by:

**Sang H. Son
Associate Professor**

**DISTRIBUTION STATEMENT A
Approved for public release
Distribution Unlimited**

**SEAS Report No. UVA/525449/CS93/102
November 1992**

DEPARTMENT OF COMPUTER SCIENCE

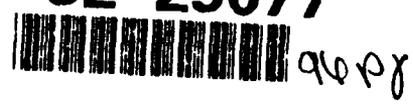
**SCHOOL OF
ENGINEERING & APPLIED SCIENCE**



**University of Virginia
Thornton Hall
Charlottesville, VA 22903**

401975

92-29077



92 11 05 135

UNIVERSITY OF VIRGINIA
School of Engineering and Applied Science

The University of Virginia's School of Engineering and Applied Science has an undergraduate enrollment of approximately 1,500 students with a graduate enrollment of approximately 600. There are 160 faculty members, a majority of whom conduct research in addition to teaching.

Research is a vital part of the educational program and interests parallel academic specialties. These range from the classical engineering disciplines of Chemical, Civil, Electrical, and Mechanical and Aerospace to newer, more specialized fields of Applied Mechanics, Biomedical Engineering, Systems Engineering, Materials Science, Nuclear Engineering and Engineering Physics, Applied Mathematics and Computer Science. Within these disciplines there are well equipped laboratories for conducting highly specialized research. All departments offer the doctorate; Biomedical and Materials Science grant only graduate degrees. In addition, courses in the humanities are offered within the School.

The University of Virginia (which includes approximately 2,000 faculty and a total of full-time student enrollment of about 17,000), also offers professional degrees under the schools of Architecture, Law, Medicine, Nursing, Commerce, Business Administration, and Education. In addition, the College of Arts and Sciences houses departments of Mathematics, Physics, Chemistry and others relevant to the engineering research program. The School of Engineering and Applied Science is an integral part of this University community which provides opportunities for interdisciplinary work in pursuit of the basic goals of education, research, and public service.

Annual Report
Grant No. N00014-91-J-1102

October 1, 1991 - September 30, 1992

THE STARLITE PROJECT - PROTOTYPING REAL-TIME SOFTWARE

Submitted to:

Office of Naval Research
Chief of Naval Research
Code 1267/Annual Report
Ballston Tower One
Room 528
800 N. Quincy Street
Arlington, VA 22217-5660

Attention:

Dr. James G. Smith, Program Manager
Information Systems

Submitted by:

Sang H. Son
Associate Professor

Department of Computer Science
UNIVERSITY OF VIRGINIA
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
THORNTON HALL
CHARLOTTESVILLE, VA 22903-2442

DTIC QUALITY INSPECTED 4

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

SEAS Report No. UVA/525449/CS93/102
November 1992

Copy No. _____

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE November 1992	3. REPORT TYPE AND DATES COVERED Annual Report 10/1/91 - 9/30/92	
4. TITLE AND SUBTITLE The Starlite Project - Prototyping Real-Time Software			5. FUNDING NUMBERS N00014-91-J-1102	
6. AUTHOR(S) Sang H. Son				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Virginia Department of Computer Science Thornton Hall Charlottesville, VA 22903-2442			8. PERFORMING ORGANIZATION REPORT NUMBER UVA/525449/CS93/102	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research 800 N. Quincy Street Arlington, VA 22217-5000			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The StarLite Project has the goal of discovering a set of design principles and developing efficient algorithms for distributed real-time systems. The initial focus of the project is on scheduling algorithms and database systems. The project also involves the construction of a prototyping environment that supports experimentation with concurrent and distributed/parallel algorithms for performance testing. One of the most important achievements in this project is the development of new scheduling algorithms based on the idea of adjusting the serialization order of active transactions dynamically. When compared with conventional transaction scheduling algorithms, our algorithms significantly improve the percentage of high priority transactions that meet the deadline. In addition, we have developed experimental database systems for performance evaluation of new technology. The RTDB server on ARTS kernel is extended to support application programmatic interface, graphic user interface, imprecise computing server, indexing mechanism, and distributed multiple server.				
14. SUBJECT TERMS real-time, operating systems, database, scheduling			15. NUMBER OF PAGES	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited	

TABLE OF CONTENTS

	<u>Page</u>
1. PRODUCTIVITY MEASURES	1
2. SUMMARY OF TECHNICAL PROGRESS	3
2.1. Real-Time Transactions	3
2.2. Fault-Tolerant Multiprocessor Scheduling	4
2.3. Experimental Systems and Prototyping Tools	4
3. PUBLICATIONS, PRESENTATIONS, AND REPORTS	6
4. TRANSITIONS AND DOD INTERACTIONS	9
5. SOFTWARE AND HARDWARE PROTOTYPES	10
APPENDIX: Publications	

Robert P. Cook and Sang H. Son
University of Virginia
(804) 982-2205
son@cs.virginia.edu
The StarLite Project
N00014-91-J-1102
10/1/91 - 9/30/92

1. Productivity Measures

- Refereed papers submitted but not yet published: 4
- Refereed papers published: 11
- Unrefereed reports and articles: 5
- Books or parts thereof submitted but not yet published: 3
- Books or parts thereof published: 4
- Patents filed but not yet granted: 0
- Patents granted: 0
- Invited presentations: 5
- Contributed presentations: 6
- Honors received: 11
- Prizes or awards received: 0
- Promotions obtained: 1
- Graduate students supported: 11
- Post-docs supported: 0
- Minorities supported: 0

Honors

- Son, Program Chairman, Tenth IEEE Workshop on Real-Time Operating Systems and Software, to be held in May 1993.
- Son, Program Committee, Sixth International Conference on Distributed Computing Systems, to be held in May 1993.
- Son, Program Committee, International Symposium on Database Systems for Advanced Applications, to be held in April 1993.
- Son, ACM National Lecturer, 1991-1993.

- Son, Program Committee, Ninth IEEE Workshop on Real-Time Operating Systems and Software, 1992.
- Son, Group Leader, Ninth IEEE Workshop on Real-Time Operating Systems and Software, Group 4: Architecture, Methodology, and Databases, 1992.
- Son, Program Committee, IEEE Workshop on Transaction and Query Processing, 1992.
- Son, Program Committee, ACM SIGMOD Conference on Management of Data 1991.
- Son, Chair, Technical Activities Committee, Korean Computer Scientists and Engineers Association, 1991.
- Son, Session Chair, IEEE Real-Time Systems Symposium, 1991.
- Son, Panelist, International Conference on Very Large Data Bases (VLDB '91), on the panel "Real-Time Databases," 1991.

Promotions

Sang H. Son was promoted to the rank of Associate Professor with the granting of tenure, effective on July 1, 1992.

Graduate Students

Juhnyoung Lee (Ph.D.), scheduling real-time transactions

Shi-Chin Chiang (Ph.D.), run-time monitoring in distributed real-time databases

Young-Kuk Kim (Ph.D.), OS support for real-time database systems

Lifeng Hsu (Ph.D.), real-time distributed resource management algorithms

Henry Oh (Ph.D.), fault-tolerant multiprocessor real-time systems

Rasikan David (Ph.D.), real-time operating systems issues

Ambar Sarkar (Ph.D.), RTDB server for RS/6000 platform

David George (M.S.), real-time database system development

Du-Won Kang (M.S.), MRDB development

Matt Lehr (M.S.), distributed real-time testbed development

Carmen Iannacone (M.S.), RTDB development

Robert Beckinger (M.S.), temporal data modeling

Spiros Kouloubis (M.S.), replication control in real-time databases

Stavros Yannopolous (M.S.), RTDB development

Savita Shamsunder (M.S.), optimistic concurrency control protocols

Robert P. Cook and Sang H. Son
University of Virginia
(804) 982-2205
son@cs.virginia.edu
The StarLite Project
N00014-91-J-1102
10/1/91 - 9/30/92

2. Summary of Technical Progress

During the past year, our research was directed towards discovering a set of design principles and developing efficient algorithms for distributed real-time systems and databases. Our efforts have been concentrated on three main areas: real-time transaction processing, fault-tolerant multiprocessor scheduling, experimental systems and prototyping tools.

2.1. Real-Time Transactions

One of the most important achievements in this project is the development of new scheduling algorithms based on the idea of adjusting the serialization order of active transactions dynamically. This is the first successful attempt to integrate benefits of the pessimistic and optimistic approaches for transaction scheduling. Two algorithms are developed based on the notion of dynamic serialization to control blocking and aborting in a more effective manner. One is based on a priority-locking mechanism that uses the phase-dependent control of optimistic approach, while the other is based on dynamic timestamp allocation. We have implemented the first lock-based algorithm using the StarLite environment for performance evaluation. When compared with conventional transaction scheduling algorithms, it significantly improves the percentage of high priority transactions that meet the deadline. Furthermore, it is shown that the algorithm provides a very high discriminating power which enables the system to support higher priority transactions at the expense of lower priority ones when a transient overload occurs. In addition, we have evaluated optimistic concurrency control protocols for real-time database systems. Our results indicate that optimistic or hybrid approaches may outperform the pessimistic approach in a wide operational range.

We also have developed algorithms for resource management in distributed real-time systems. They are priority-ordered deadlock avoidance algorithms, efficient deadlock detection/resolution algorithms using partial resource allocation graphs, and a synchronization scheme for replicated critical data in distributed real-time database systems. Those algorithms are very efficient for distributed real-time systems, in which critical resources should be managed to support consistency, while satisfying timing constraints. Especially for replication control, we have employed a new consistency criterion, less stringent than conventional one-copy serializability. This scheme is very flexible and practical, because no prior knowledge of the data requirements or the execution time of each transaction is required. Using our StarLite prototyping environment, we have implemented those algorithms and demonstrated that they provide higher level of concurrency and greater flexibility in meeting timing requirements.

2.2. Fault-Tolerant Multiprocessor Scheduling

To investigate feasible solutions for scheduling real-time tasks in parallel/distributed environments, we have developed a new paradigm for multiprocessor real-time systems, and implemented a parallel programming interface based on our paradigm. Our new paradigm has created new research opportunities for operating systems and databases for parallel computing systems with timing and fault-tolerance requirements. For example, using the new programming interface, we have developed PRDB, an experimental real-time database system that runs on an emulated tightly-coupled multiprocessor system in the StarLite environment. It provides a general paradigm for exploiting parallelism and different real-time scheduling policies. This experimental system has been used for investigating implementation techniques for parallel database systems and the impact of multiprocessor technology on operating systems design.

To support both real-time and fault-tolerance requirements, an algorithm to schedule a number of tasks with their timing and precedence constraints on a number of processors is necessary. We have developed a scheduling model under which timing and fault-tolerance constraints can be expressed. Using this model, a scheduling problem to tolerate one arbitrary task error or processor failure has been studied. Since most multiprocessor scheduling problems are NP-complete, we have developed heuristics to obtain near-optimal solutions to the problem. We assume that all the critical tasks are periodic, and they have hard deadlines. We use two versions of each critical task, one as the primary task and the other as the secondary. The scheduling algorithm is based on the first-fit decreasing bin packing heuristics. Using the StarLite environment, the algorithm was implemented and its performance was evaluated. It was shown that the algorithm performs very well, finding the optimal solution most of the time.

2.3. Experimental Systems and Prototyping Tools

We have developed a suite of database systems on several platforms, such as StarLite, ARTS, and UNIX, and utilized them as system integration testbeds. Since a real-time system must operate in the context of operating system services, correct functioning and timing behavior of the system depends heavily on the operating system interfaces. We have developed a multi-thread database server, called RTDB, for ARTS real-time operating system kernel. The RTDB now supports application programmatic interface and graphic user interface. The application programmatic interface (API) provides an easy way for the database application programmer to construct batch clients. The API currently provides Create, Insert, Select, and Update. With imprecise server, a client can specify a deadline by which a computation (query) must complete. If the server is unable to complete the entire query, the server will return imprecise result, provided the computation had proceeded to a point where the output would be meaningful and appropriate. One problem that hinders the transformation of a non-real-time database function to a real-time one for imprecise server is recursion. Recursive function are not amenable to being stopped as easily as iterative functions. To implement the imprecise server, we have used the state machine approach in representing the execution stages of each function. Necessary actions are performed with a measurable amount of time allotted to each stage of execution.

In addition, we have developed a separate experimental system, called MRDB, a real-time database kernel running on Sun/Unix environment. In MRDB, servers and clients can be created and removed dynamically. The servers use valid time attribute and run-time estimate of requests in transaction scheduling to reduce the number of deadline-missing transactions. Using MRDB, we have performed several experiments to evaluate design alternatives in real-time scheduling and concurrency control. The temporal database kernel on Sun/Unix environment is transported to IBM RS/6000 with AIX. Ada programming interface is then developed to support a set of basic access functions to the database. We have simulated RT-DOSE (Real-time Distributed Operating System Experiments) using the interface.

Our experimental systems achieve other goal of this project—to transfer technology developed under the StarLite project to Navy, DoD, and other research organizations. Currently, Naval Ocean Systems Center in San Diego, California, is using RTDB for their distributed real-time experiments.

Robert P. Cook and Sang H. Son
University of Virginia
(804) 982-2205
son@cs.virginia.edu
The StarLite Project
N00014-91-J-1102
10/1/91 - 9/30/92

3. Publications, Presentations, and Reports

• Books and Book Chapters

- (1) S. H. Son, J. Lee, and H. Kang, "Approaches to Design of Real-Time Database Systems," *Database Systems for Next-Generation Applications - Principles and Practice*, W. Kim, Y. Kambayashi, and I. Paik (eds.), World Scientific Publishing, 1993 (to appear).
- (2) S. H. Son, C. Chang, and Y. Kim, "Performance Evaluation of Real-Time Locking Protocols," *Database Systems for Next-Generation Applications - Principles and Practice*, W. Kim, Y. Kambayashi, and I. Paik (eds.), World Scientific Publishing, 1993 (to appear).
- (3) S. H. Son and S. Park, "Scheduling Transactions for Distributed Time-Critical Applications," in *Advances in Distributed Systems*, T. Casavant and M. Singhal (Editors), IEEE Computer Society, 1992(to appear).
- (4) S. H. Son, R. Cook, J. Lee, and H. Oh, "New Paradigms for Real-Time Database Systems," in "Real-Time Programming," K. Ramamritham and W. Halang (Editors), Pergamon Press, 1992.
- (5) R. Cook, L. Hsu, and S. H. Son, "Real-Time, Priority-Ordered, Deadlock Avoidance Algorithms," in *Foundations of Real-Time Computing: Scheduling and Resource Management*, A. Van Tilborg and G. M. Koob (Editors), Kluwer Academic Publishers, 1991, pp 307-324.
- (6) S. H. Son, Y. Lin, and R. Cook, "Concurrency Control in Real-Time Database Systems," in *Foundations of Real-Time Computing: Scheduling and Resource Management*, A. Van Tilborg and G. M. Koob (Editors), Kluwer Academic Publishers, 1991, pp 185-202.
- (7) R. P. Cook, "The StarLite Operating System," *Operating Systems for Mission-Critical Computing*, K. Gordon, P. Hwang, and A. Agrawala (Editors), ACM Press, 1991.

● **Refereed Journal Publications**

- (8) S. H. Son, J. Ratner, S. Chiang, "StarBase: A Simulation Laboratory for Distributed Database Research," *Journal of Computer Simulation*, (to appear).
- (9) 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*, 1992, vol. 4, no. 3, pp 269-276.
- (10) S. H. Son, "Scheduling Real-Time Transactions using Priority," *Information and Software Technology*, vol. 34, no. 6, June 1992, pp 409-415.
- (11) S. H. Son, "An Environment for Integrated Development and Evaluation of Real-Time Distributed Database Systems," *Journal of Systems Integration*, vol. 2, no. 1, February 1992

● **Refereed Conference Publications**

- (12) S. H. Son and S. Koloumbis, "Replication Control for Distributed Real-Time Database Systems," *12th International Conference on Distributed Computing Systems*, Yokohama, Japan, pp 144-151, June 1992.
- (13) S. H. Son, S. Yannopoulos, Y-K. Kim, C. Iannacone, "Integration of a Database System with Real-Time Kernel for Time Critical Applications," *Second International Conference on System Integration*, Morristown, New Jersey, pp 172-180, June 1992.
- (14) S. H. Son and J. Lee, "A New Approach to Real-Time Transaction Scheduling," *4th Euromicro Workshop on Real-Time Systems*, Athens, Greece, June 1992, pp 177-182.
- (15) Y. Oh and S. H. Son, "An Algorithm for Real-Time Fault-Tolerant Scheduling in Multiprocessor Systems," *4th Euromicro Workshop on Real-Time Systems* Athens, Greece, June 1992, pp 190-195.
- (16) S. H. Son, S. Park, and Y. Lin, "An Integrated Real-Time Locking Protocol," *Eighth IEEE International Conference on Data Engineering*, Phoenix, Arizona, February 1992, pp 527-534.
- (17) S. H. Son, J. Lee, and S. Shamsunder, "Real-Time Transaction Processing: Pessimistic, Optimistic, and Hybrid Approaches," *Second International Workshop on Transactions and Query Processing*, Tempe, Arizona, February 1992.
- (18) Y. Oh and S. H. Son, "Multiprocessor Support for Real-Time Fault-Tolerant Scheduling," *IEEE Workshop on Architectural Aspects of Real-Time Systems*, San Antonio, Texas, December 1991, pp 76-80.

● **Technical Reports**

- (19) Y. Oh and S. H. Son, "Fault-Tolerant Real-Time Multiprocessor Scheduling," *Technical Report TR-92-09*, Dept. of Computer Science, University of Virginia,

April 1992.

- (20) S. H. Son, J. Lee, and Y. Lin, "Hybrid Protocols using Dynamic Adjustment of Serialization Order," *Technical Report TR-92-07*, Dept. of Computer Science, University of Virginia, March 1992.
- (21) Y. Yang, L. Hsu, and S. H. Son, "Distributed Algorithms for Efficient Deadlock Detection and Resolution," *Technical Report TR-92-06*, Dept. of Computer Science, University of Virginia, February 1992.

• **Presentations**

- Son, Replication Control for Distributed Real-Time Database Systems, 12th International Conference on Distributed Computing Systems, Yokohama, Japan, June 1992.
- Son, A New Approach to Real-Time Transaction Scheduling, 4th Euromicro Workshop on Real-Time Systems, Athens, Greece, June 1992
- Son, An Integrated Real-Time Locking Protocol," IEEE International Conference on Data Engineering, Phoenix, Arizona, February 1992.
- Son, Real-Time Transaction Processing: Pessimistic, Optimistic, and Hybrid Approaches," International Workshop on Transactions and Query Processing, Tempe, Arizona, February 1992.
- Son, Multiprocessor Support for Real-Time Fault-Tolerant Scheduling, IEEE Real-Time Systems Symposium, San Antonio, Texas, December 1991.

Robert P. Cook and Sang H. Son
University of Virginia
(804) 982-2205
son@cs.virginia.edu
The StarLite Project
N00014-91-J-1102
10/1/91 - 9/30/92

4. Transitions and DOD Interactions

- Son, The Chronus Project, Proposal submitted to James Smith, Information Systems Division, Office of Naval Research, September 1992.
- Son, Replication Control for Distributed Real-Time Database Systems, presentation at the Sogang University, June 1992.
- Son, Real-Time Systems and Databases, presentation at the Seoul National University, June 1992.
- Son, the real-time database server, version 2.0, installed at NRaD distributed real-time testbed, April 1992.
- Son, real-time database project coordination meeting with IBM, Charlottesville, Virginia, April 1992.
- Son, Real-Time Systems and Databases, Kingston, Rhode Island, presentation at the University of Rhode Island, March 1992.
- Son, Real-Time Systems and Databases, presentation at the Boston University, Boston, Massachusetts, March 1992.
- Son, Advanced Real-Time Database Systems Project, Proposal submitted to Les Anderson, NRaD, February 1992.
- Son, Real-Time Database Systems, NOSC Code 413 DC² Review Meeting, San Diego, California, January 1992.
- Son, Real-Time Systems and Databases: Issues and Research Directions, presentation at the KSEA Symposium on Science and Technology, Washington, DC, December 1991.
- Son, StarLite project research coordination meeting with ONR, Charlottesville, Virginia, November 1991.
- Son, presentation at the ONR Foundations of Real-Time Computing Workshop, Washington, DC, October 1991.
- Son, meeting with ONR patents office staff, Charlottesville, Virginia, October 1991.
- Son, ARTS Real-Time Systems Project Review Meeting, Carnegie-Mellon University, Pittsburgh, Pennsylvania, October 1991.

Robert P. Cook and Sang H. Son
University of Virginia
(804) 982-2205
son@cs.virginia.edu
The StarLite Project
N00014-91-J-1102
10/1/91 - 9/30/92

5. Software and Hardware Prototypes

The RTDB real-time database system has been upgraded and delivered to NRaD. However, we still have a tremendous amount of work to do in fixing minor problems and identifying performance bottlenecks. The StarLite prototyping environment has been distributed to several universities as beta test sites. Both RTDB and StarLite still need a lot of work for providing proper documentation.

APPENDIX: Publications

An Integrated Real-Time Locking Protocol

Sang H. Son, Seog Park†, and Yi Lin

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

† Department of Computer Science, Sogang University, Seoul, Korea

Abstract

Database systems for real-time applications must satisfy timing constraints associated with transactions, in addition to maintaining the consistency of data. In this paper we examine a priority-driven locking protocol called integrated real-time locking protocol. We show that this protocol is free of deadlock, and in addition, a high priority transaction is not blocked by uncommitted lower priority transactions. The protocol does not assume any knowledge about the data requirements or the execution time of each transaction. This makes the protocol widely applicable, since in many actual environments such information may not be readily available. Using a database prototyping environment, it is shown that the proposed protocol offers performance improvement over the two-phase locking protocol.

1. Introduction

Real-time database systems (RTDBS) are transaction processing systems where transactions have explicit timing constraints. Typically, a timing constraint is expressed in the form of a *deadline*, a certain time in the future by which a transaction needs to be completed. A deadline is said to be *hard* if it cannot be missed or else the result is useless. If a deadline can be missed, it is a *soft* deadline. With soft deadlines, the usefulness of a result may decrease after the deadline is missed. In RTDBS, the correctness of transaction processing depends not only on maintaining consistency constraints and producing correct results, but also on the time at which a transaction is completed. Transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query on the tracking data for a missile must be processed within given deadlines. RTDBS are becoming increasingly important in a wide range of applications, such as computer integrated manufacturing, traffic control systems, robotics, and in stock market trading.

Conventional data models and databases are not adequate for time-critical applications, since they are not designed to provide features required to support real-time transactions. They are designed to provide good average performance, while possibly yielding unacceptable worst-case response times. Very few of them allow users to specify or ensure timing constraints. Recently, interest in this new application domain is growing in database community [Abb88, Abb89, Har90, Kor90, Lin90, Sha88, Sha91, Son88b, Son91].

While the theories of concurrency control in database systems and real-time task scheduling have both advanced, little attention has been paid to the interaction between concurrency control protocols and real-time scheduling algorithms [Stan88]. In database concurrency control, meeting the deadline is typically not addressed. The objective is to provide a high degree of concurrency and thus faster average response time without violating data consistency. In real-time task scheduling, on the other hand, it is customary to assume that tasks are independent. The objective here is to maximize resources, such as CPU utilization, subject to meeting timing constraints. In addition, it is assumed that the resource and data requirements of tasks are known. Scheduling in RTDBS is a combination of the two scheduling mechanisms.

Real-time task scheduling methods can be extended for real-time transaction scheduling while concurrency control protocols are still needed for operation scheduling to maintain data consistency. The general approach is to utilize existing concurrency control protocols, especially two-phase locking (2PL), and to apply time-critical transaction scheduling methods that favor more urgent transactions [Abb88, Sha91]. Such approaches have the inherent disadvantage of being limited by the concurrency control method upon which they are based, since all existing concurrency control methods synchronize concurrent data access of transactions by the combination of two measures: blocking and roll-backs of transactions. Both are barriers to meeting time-critical schedules.

Concurrency control protocols induce a serialization order among conflicting transactions. In non-real-time concurrency control protocols, timing constraints are

This work was supported in part by ONR contract # N00014-91-J-1102, by NOSC, by DOE, and by VCIT.

not a factor in the construction of this order. This is obviously a drawback for RTDBS. The conservative 2PL uses blocking, but in RTDBS, blocking may cause *priority inversion*. Priority inversion is said to occur when a high priority transaction is blocked by lower priority transactions [Sha88]. The alternative is to abort low priority transactions when priority inversion occurs. This wastes the work done by the aborted transactions and in turn also has a negative effect on time-critical scheduling.

Satisfying the timing constraints while preserving data consistency requires the concurrency control algorithms to accommodate timeliness of transactions as well as to maintain data consistency. This is the very goal of our work. If the information about data requirements and execution time of each transaction is available beforehand, off-line preanalysis can be performed to avoid conflicts [Sha91]. However, such approaches may delay the starting of some transactions, even if they have high priorities, and may reduce the concurrency level in the system. This, in return, may lead to the violation of the timing constraints and degrade system performance.

In this paper we examine a priority-driven two-phase locking protocol called the integrated real-time locking protocol. It is an integrated locking protocol, since it decomposes the problem of concurrency control into two subproblems, namely read-write synchronization and write-write synchronization, and integrates the solutions to two subproblems consistently to yield a correct solution to the entire problem [Bern87]. We show that this protocol is free of deadlock. The protocol is similar to optimistic concurrency control protocols [Kung81] in the sense that each transaction has three phases, but unlike the optimistic approach, there is no validation phase. While other optimistic concurrency control protocols resolve conflicts in the validation phase, this protocol resolves them in the read phase using transaction priority.

The remainder of this paper is organized as follows. The details of the locking protocol are described in Section 2. The properties of the protocol is discussed in Section 3. Section 4 presents performance evaluation of the real-time locking protocol. Finally, concluding remarks appear in Section 5.

2. The Integrated Real-Time Locking Protocol

2.1. Basic Concepts

A RTDBS is often used by applications such as tracking. Since we cannot predict how many objects need to be tracked and when they appear, we assume randomly arriving transactions. Each transaction is assigned an *initial priority* and a *start-timestamp* when it is submitted to the system. The initial priority can be based on the deadline and the criticality of the transaction. The start-timestamp is appended to the initial priority to form the *actual priority* that is used in scheduling. When we refer

to the priority of a transaction, we always mean the actual priority with the start-timestamp appended. Since the start-timestamp is unique, so is the priority of each transaction. The priority of transactions with the same initial priority is distinguished by their start-timestamps.

With two-phase locking and priority assignment, we can encounter the problem of priority inversion. What we need is a concurrency control algorithm that allows transactions to meet the timing constraints as much as possible without reducing the concurrency level of the system in the absence of any *a priori* information. The integrated real-time locking protocol presented in this paper meets these goals. It has the flavor of both locking and optimistic methods.

Transactions write into the database only after they are committed. By using a priority-dependent locking protocol, the serialization order of active transactions is adjusted dynamically, making it possible for transactions with higher priorities to be executed first so that higher priority transactions are never blocked by uncommitted lower priority transactions, while lower priority transactions may not have to be aborted even in face of conflicting operations. The adjustment of the serialization order can be viewed as a mechanism to support time-critical scheduling.

Example 1: Assume T_1 and T_2 are two transactions with T_1 having a higher priority. T_2 writes a data object x before T_1 reads it. Using 2PL, even in the absence of any other conflicting operations between these two transactions, T_1 has to either abort T_2 or be blocked until T_2 releases the write lock.

In Example 1, T_1 can never precede T_2 in the serialization order, because the serialization order $T_2 \rightarrow T_1$ is already determined by the past execution history. In our protocol, when such conflict occurs, the serialization order of the two transactions will be adjusted in favor of T_1 , i.e. $T_1 \rightarrow T_2$, and neither is T_1 blocked nor is T_2 aborted. Together with priority-based blocking, the real-time locking protocol is free from deadlocks.

All transactions that can be scheduled are placed in a ready queue, R_Q . Only transactions in R_Q are scheduled for execution. When a transaction is *blocked*, it is removed from R_Q . When a transaction is *unblocked*, it is inserted into R_Q again, but may still be waiting to be assigned the CPU. A transaction is said to be *suspended* when it is not executing, but still in R_Q . When a transaction is doing I/O operations, it is *blocked*. Once it completes, it is usually *unblocked*.

The execution of each transaction is divided into three phases: the read phase, the wait phase and the write phase. During the read phase, a transaction reads from the database and writes to its local workspace. After it completes, it waits for its chance to commit in the wait phase. If it is committed, it switches into the write phase

during which all its updates are made permanent in the database. A transaction in any of the three phases is called *active*. We take an approach of integrated schedulers in that it uses 2PL for read-write conflicts and the Thomas' Write Rule (TWR) for write-write conflicts. The TWR *ignores* a write request that has arrived late, rather than *rejects* it [Bern87].

In our protocol, there are various data structures that need to be read and updated in a consistent manner. Therefore we assume the existence of critical sections to guarantee that only one process at a time updates these data structures. We assume critical sections of various classes to group the various data structures and allow maximum concurrency. We also assume that each assignment statement of global data is executed atomically.

2.2. Read Phase

The read phase is the normal execution of a transaction except that write operations are performed on private data copies in the local workspace of the transaction instead of on data objects in the database. We call such write operations *prewrites*, denoted by $pw_T[x]$. A write request from a transaction is performed by a prewrite operation. Since each transaction has its own local workspace, a prewrite operation does not write into the database, and if a transaction previously wrote a data object, subsequent read operations to the same data object retrieve the value from the local workspace.

The read-prewrite or prewrite-read conflicts between active transactions are synchronized during this phase by a *priority-based locking protocol*. Before a transaction can perform a read (resp. prewrite) operation on a data object, it must obtain the read (resp. write) lock on that data object first. A read (resp. write) lock on x by transaction T is denoted by $rlock(T,x)$ (resp. $wlock(T,x)$). If a transaction reads a data object that has been written by itself, it gets the private copy in its own workspace and no read lock is needed. In the rest of the paper, when we refer to read operations, we exclude such read operations because they do not induce any dependencies among transactions.

The locking protocol is based on the principle that higher priority transactions should complete before lower priority transactions. That is, if two transactions conflict, the higher priority transaction should precede the lower priority transaction in the serialization order. Using an appropriate CPU scheduling policy for RTDBS, a high priority transaction can be scheduled to commit before a low priority transaction in most cases [Lin90]. If a low priority transaction does complete before a high priority transaction, it is required to wait until it is sure that its commitment will not lead to the higher priority transaction being aborted.

Suppose active transaction T_1 has higher priority than active transaction T_2 . We have four possible conflicts and the transaction dependencies they require in the serialization order as follows:

(1) $r_{T_1}[x], pw_{T_2}[x]$

The resulting serialization order is $T_1 \rightarrow T_2$, which satisfies the priority order, and hence it is not necessary to adjust the serialization order.

(2) $pw_{T_1}[x], r_{T_2}[x]$

Two different serialization orders can be induced with this conflict; $T_2 \rightarrow T_1$ with immediate reading, and $T_1 \rightarrow T_2$ with delayed reading. Certainly, the later should be chosen for priority scheduling. The delayed reading means that $r_{T_2}[x]$ is blocked by the write lock of T_1 on x .

(3) $r_{T_2}[x], pw_{T_1}[x]$

The resulting serialization order is $T_2 \rightarrow T_1$, which violates the priority order. If T_2 is in the read phase, it is aborted because otherwise T_2 must commit before T_1 and thus block T_1 . If T_2 is in its wait phase, avoid aborting T_2 until T_1 commits, in the hope that T_2 gets a chance to commit before T_1 commits. If T_1 commits, T_2 is aborted. But if T_1 is aborted by some other conflicting transaction, then T_2 is committed. With this policy, we can avoid unnecessary and useless aborts, while satisfying priority scheduling.

(4) $pw_{T_2}[x], r_{T_1}[x]$

Two different serialization orders can be induced with this conflict; $T_1 \rightarrow T_2$ with immediate reading, and $T_2 \rightarrow T_1$ with delayed reading. If T_2 is in its write phase, delaying T_1 is the only choice. This blocking is not a serious problem for T_1 because T_2 is expected to finish writing x soon. T_1 can read x as soon as T_2 finishes writing x in the database, not necessarily after T_2 completes the whole write phase. If T_2 is in its read or wait phase, choose immediate reading.

As transactions are being executed and conflicting operations occur, all the information about the induced dependencies in the serialization order needs to be retained. To do this, we retain two sets for each transaction, *before_trset* and *after_trset*, and a count, *before_cnt*. The set *before_trset* (resp. *after_trset*) contains all the active lower priority transactions that must precede (resp. follow) this transaction in the serialization order. *before_cnt* is the number of the higher priority transactions that precede this transaction in the serialization order. When a conflict occurs between two transactions, their dependency is set and their values of *before_trset*, *after_trset*, and *before_cnt* will be changed accordingly.

By summarizing what we discussed above, we define the real-time locking protocol as follows:

LP1. Transaction T requests a read lock on data object x .

```

for all transactions  $t$  with  $wlock(t,x)$  do
  if ( $priority(t) > priority(T)$ 
     or  $t$  is in write phase)
    /* Case 2, 4 */
    then deny the lock and exit;
  endif
enddo
for all transactions  $t$  with  $wlock(t,x)$  do
  /* Case 4 */
  if  $t$  is in before  $trset_T$  then abort  $t$ ;
  else if ( $t$  is not in after  $trset_T$ )
    then
      include  $t$  in after  $trset_T$ ;
       $before\_cnt_t := before\_cnt_t + 1$ ;
    endif
  endif
enddo
grant the lock;

```

LP2. Transaction T requests a write lock on data object x .

```

for all transactions  $t$  with  $rlock(t,x)$  do
  if  $priority(t) > priority(T)$ 
    then /* Case 1 */
      if ( $T$  is not in after  $trset_t$ )
        then
          include  $t$  in after  $trset_t$ ;
           $before\_cnt_T := before\_cnt_T + 1$ ;
        endif
      else
        if  $t$  is in wait phase /* Case 3 */
          then
            if ( $t$  is in after  $trset_T$ )
              then abort  $t$ ;
            else
              include  $t$  in before  $trset_T$ ;
            endif
          else if  $t$  is in read phase
            then abort  $t$ ;
          endif
        endif
      endif
    enddo
grant the lock;

```

LP1 and LP2 are actually two procedures of the lock manager that are executed when a lock is requested. When a lock is denied due to a conflicting lock, the request is suspended until that conflicting lock is released. Then the locking protocol is invoked once again from the very beginning to decide whether the lock can be granted now. With our locking protocol, a data object may be both read locked and write locked by several

transactions simultaneously.

2.3. Wait Phase

The wait phase allows a transaction to wait until it can commit. A transaction can commit only if all transactions with higher priorities that must precede it in the serialization order are either committed or aborted. Since $before_cnt$ is the number of such transactions, the transaction can commit only if its $before_cnt$ becomes zero. A transaction in the wait phase may be aborted due to two reasons; if a higher priority transaction requests a conflicting lock, or if a higher priority transaction that must follow this transaction in the serialization order commits first. Once a transaction in the wait phase gets its chance to commit, i.e. its $before_cnt$ goes to zero, it switches to the write phase and release all its read locks. The transaction is assigned a final-timestamp, which is the absolute serialization order.

2.4. Write Phase

Once a transaction is in the write phase, it is considered to be committed. All committed transactions are serialized by the final-timestamp order. Updates are made permanent to the database while applying Thomas' Write Rule (TWR) for write-write conflicts [Ber87]. After each operation the corresponding write lock is released.

3. Properties and Correctness

Having described the basic concepts and the protocol, we now present some properties and prove the correctness of the protocol. First, we give the simple definitions of *history* and *serialization graph* (SG). For the formal definitions, readers are referred to [Bern87]. A history is a partial order of operations that represents the execution of a set of transactions. Any two conflicting operations must be comparable. Let H be a history. The *serialization graph* for H , denoted by $SG(H)$, is a directed graph whose nodes are committed transactions in H and whose edges are all $T_i \rightarrow T_j$ ($i \neq j$) such that one of T_i 's operations precedes and conflicts with one of T_j 's operations in H . To prove a history H serializable, we only have to prove that $SG(H)$ is acyclic [Bern87].

Theorem 1: Every history H produced by the protocol is serializable.

Proof: Let T_1 and T_2 be two committed transactions in a history H produced by the algorithm. We argue that if there is an edge $T_1 \rightarrow T_2$ in $SG(H)$, then $ts(T_1) < ts(T_2)$. Since $T_1 \rightarrow T_2$, the two must have conflicting operations. There are three cases.

Case 1: $w_1[x] \rightarrow w_2[x]$

Suppose $ts(T_2) < ts(T_1)$. Therefore T_2 enters into the write phase before T_1 . If $w_1[x]$ is sent to the data

manager first, T_2 's write lock on x must be released before $w_1[x]$ is sent to the data manager. If $w_2[x]$ is sent to the data manager first, it will either be processed before $w_1[x]$ is sent to the data manager, or be discarded when the data manager receives $w_1[x]$, because $w_2[x]$ has a smaller timestamp. Therefore $w_1[x]$ is never processed before $w_2[x]$. Such conflict is impossible. A contradiction.

Case 2: $r_1[x] \rightarrow w_2[x]$

If T_2 holds the write lock on x when T_1 requests the read lock, we must have $priority(T_1) > priority(T_2)$ and T_2 is not in the write phase, because otherwise T_1 would have been blocked by LP1. By LP1, T_2 is in *after_trset* $_{T_1}$. T_2 will not switch into the write phase before T_1 does, because *before_cnt* $_{T_2}$ cannot be zero with T_1 still in the read or wait phase. Therefore $ts(T_1) < ts(T_2)$. If T_1 holds read lock on x when T_2 requests the write lock, by LP2, we have either T_2 is in *after_trset* $_{T_1}$ or T_1 is in *before_trset* $_{T_2}$, depending on the priorities of the two transactions. In either case, T_1 must commit before T_2 . Hence we also have $ts(T_1) < ts(T_2)$.

Case 3: $w_1[x] \rightarrow r_2[x]$

Since T_1 is already in the write phase before T_2 reads x , we must have $ts(T_1) < ts(T_2)$.

Suppose there is a cycle $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$ in $SG(H)$. By the above argument, we have $ts(T_1) < ts(T_2) < \dots < ts(T_n) < ts(T_1)$. This is impossible. Therefore no cycle can exist in $SG(H)$ and the algorithm only produces serializable histories. \square

Theorem 2: There is no mutual deadlock under the real-time locking protocol.

Proof: In the algorithm, a high priority transaction can be blocked by a low priority transaction only if the low priority transaction is in the write phase. Suppose there is a cycle in the wait-for graph (WFG), $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n \rightarrow T_1$. For any edge $T_i \rightarrow T_j$ in the cycle, if $priority(T_i) > priority(T_j)$, T_j must be in the write phase, thus it cannot be blocked by any other transactions and cannot appear in the cycle. Therefore we must have $priority(T_i) < priority(T_j)$ and thus $priority(T_1) < priority(T_2) < \dots < priority(T_n) < priority(T_1)$. This is impossible. Hence a deadlock cannot exist. \square

We now discuss some properties of the protocol. First, the protocol provides a desirable property beyond the serializability, namely the strictness. From a practical viewpoint, serializability of transactions is not always enough. To ensure the correctness in the presence of failures, the concurrency control protocol must produce execution histories that are not only serializable but also recoverable. A history H is called *recoverable* if, whenever T_i reads from T_j in H and T_i commits, T_j must

commit before T_i . A history H is called *strict* if, whenever $w_i[x]$ precedes $o_j[x]$ where $o_j[x]$ is either $w_j[x]$ or $r_j[x]$, T_i must either commit or abort before $o_j[x]$. It is known that strictness is a stronger condition than recoverability, i.e., a set of strict histories is a proper subset of recoverable histories, and it is more desirable for practical reasons [Bern87].

The strictness of the histories produced by the algorithm follows obviously from the fact that a transaction applies the results of its write operations from its local workspace into the database only after it commits. This property makes the transaction recovery procedure simpler than other concurrency control protocols that do not support strictness.

Another property to be discussed is the degree of concurrency provided by the protocol. The compatibility depends on the priorities of the transactions holding and requesting the lock and the phase of the lock holder as well as the lock types. Unlike 2PL, locks are not classified simply as shared locks and exclusive locks. Even with the same lock types, different actions may be taken, depending on the priorities of the lock holder and the lock requester. With the real-time locking protocol, a data object may be both read locked and write locked by several transactions simultaneously, and hence it is less restrictive than 2PL, and can provide higher degree of concurrency by incurring less blocking and fewer aborts. In the real-time locking protocol, a high priority transaction is never blocked or aborted due to conflict with an uncommitted lower priority transaction. The probability of aborting a lower priority transaction should be less than that in 2PL under the same conditions. An analytical model may be used to estimate the exact probability, but that is beyond the scope of this paper.

4. Performance Evaluation

Since the integrated real-time locking protocol assumes that the data requirement or execution time of each transaction is not known, we should compare the protocol with other protocols with the same assumption. In this section, a comparative evaluation of the performance of the real-time locking protocol is presented. The results obtained through a simulation study indicate that the real-time locking protocol offers performance improvement over 2PL.

The performance of the real-time locking protocol was studied using a prototyping environment for database systems [Son90]. In our simulation, transactions are generated and put into the start-up queue. When a transaction is started, it leaves the start-up queue and enters the ready queue. Transactions in the ready queue are ordered from the highest to the lowest priority. The transaction with the highest priority is always selected to run. The current running transaction sends requests to the concurrency controller. The transaction may be blocked and

placed in the block queue. It may also be aborted and restarted. In such a case, it is first delayed for a certain amount of time and then put in the ready queue again. When a transaction in the block queue is unblocked, it leaves the block queue and is placed in the ready queue. Whenever a transaction enters the ready queue and its priority is higher than the current running transaction, it preempts the current running transaction.

When a transaction enters the start-up queue, it has the arrival time, the deadline, the priority, the read set and the write set associated with it. The transaction inter-arrival time is a random variable with exponential distribution. The deadline and the priority are computed by the following formulas:

$$\begin{aligned} \text{Deadline}_T &= \text{Arrival}_T + \text{Slack} * \text{Time}_T \\ \text{Priority}_T &= 1/\text{Deadline}_T \end{aligned}$$

where

$$\begin{aligned} \text{Deadline}_T &= \text{Deadline of transaction } T \\ \text{Arrival}_T &= \text{Arrival time of transaction } T \\ \text{Time}_T &= \text{Service time of transaction } T \\ \text{Slack} &= \text{Slack factor} \end{aligned}$$

The *slack factor* is a random variable between 3 and 5 with uniform distribution. The *service time* is the total time that the transaction needs for its data processing. This includes the CPU time and the I/O time. The deadline formula is designed to ensure that all transactions, independent of their service requirement, have the same chance of making their deadline. The transaction priority assignment policy is *Earliest Deadline*. Transactions with earlier deadlines have higher priority than transactions with later deadlines. A greater priority value means higher priority. The data objects in the read set and the write set are uniformly distributed across the entire database. A transaction consists of a sequence of read and write operations. A read operation involves a concurrency control request to get access permission, followed by a disk I/O to read the data object, followed by a period of CPU usage for processing the data object. Write operations are handled similarly except for their disk I/O. Since it is assumed that transactions maintain deferred update lists in buffers in main memory, disk activity of write access is deferred until the transaction has committed and switched into the write phase. A transaction can be discarded at any time if its deadline is missed. Therefore our model employs a hard deadline policy.

To ensure significance of the comparison, the classical two-phase locking needs to be augmented with a priority scheme to ensure that higher priority transactions are not delayed by lower priority transactions. We used the *High Priority* scheme [Abb88], in which all data conflicts are resolved in favor of the transaction with higher priority. When a transaction requests a lock on a data object held by other transactions in an incompatible

mode, if the requester's priority is higher than that of all the lock holders, the holders are restarted and the requester is granted the lock; if the requester's priority is lower, it waits for the lock holders to release the lock. This scheme has the advantage of deadlock prevention.

For each experiment, we collected performance statistics and averaged over 10 runs. We have used the 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 (15%) of the database so that conflicts would occur frequently. The high conflict rate allows concurrency control protocols to play a significant role in the system performance. We choose the average arrival rate so that protocols are tested in a heavily loaded rather than lightly loaded system. It is because 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 the system is under stress [Abb88].

The primary performance metric used in analyzing the experimental results is the *miss percentage* of the system, defined as the percentage of transactions that do not complete before their deadline. Miss percentage values in the range of 0 to 20 percent can be taken to represent system performance under "normal" loadings, while miss percentage values in the range of 20 to 100 percent represent system performance under "heavy" loading [Har90]. A secondary performance metric, *restarts*, is the number of restarts for a fixed number of transactions. We chose this metric because it provides insight into the system behavior. The advantage of the real-time locking protocol is that while high priority transactions are not blocked by low priority transactions, low priority transactions need not be restarted most of the time. We can verify this by using *restarts* as a performance metric.

Table 1 summarizes the key parameters of the simulation model and their default values. Transaction size (data access per transaction) is the total number of data access operations of each transaction. Among all the data access operations of a transaction, the percentage of write operations is specified by the write percentage.

By changing the mean inter-arrival time, we can study the system performance under normal load and heavy load. Fig. 4 shows that the real-time locking protocol performs better than 2PL under both normal load and heavy load. If we consider a miss percentage under 20% as "normal", the real-time locking protocol can keep the system operating satisfactorily when the mean inter-arrival time is as small as 40ms, while with 2PL the system can maintain a normal load only when the mean inter-arrival time is greater than 70 ms. Another interesting result is that under normal load, the restart number for each protocol is less than 10. A restart number greater than 10 indicates a degraded system performance for

- [Lin90] Lin, Y. and S. H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *11th IEEE Real-Time Systems Symposium*, Orlando, Florida, Dec. 1990.
- [Sha88] Sha, L., R. Rajkumar, and J. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record* 17, 1, March 1988, pp 82-98.
- [Sha91] Sha, L., R. Rajkumar, S. H. Son, and C. Chang, "A Real-Time Locking Protocol," *IEEE Transactions on Computers*, vol. 40, no. 7, July 1991, pp 793-800.
- [Son88] Son, S. H., "Semantic Information and Consistency in Distributed Real-Time Systems," *Information and Software Technology*, Vol. 30, Sept. 1988, pp 443-449.
- [Son88b] Son, S. H., guest editor, *ACM SIGMOD Record* 17, 1, Special Issue on Real-Time Database Systems, March 1988.
- [Son90] Son, S. H., "An Environment for Prototyping Real-Time Distributed Databases," *International Conference on Systems Integration*, Morristown, New Jersey, April 1990, pp 358-367.
- [Son91] Son, S. H., R. Cook, J. Lee, and H. Oh, "New Paradigms for Real-Time Database Systems," in *Real-Time Programming*, K. Ramamritham and W. Halang (Editors), Pergamon Press, 1991.
- [Stan88] Stankovic, J., "Misconceptions about Real-Time Computing," *IEEE Computer* 21, 10, October 1988, pp 10-19.

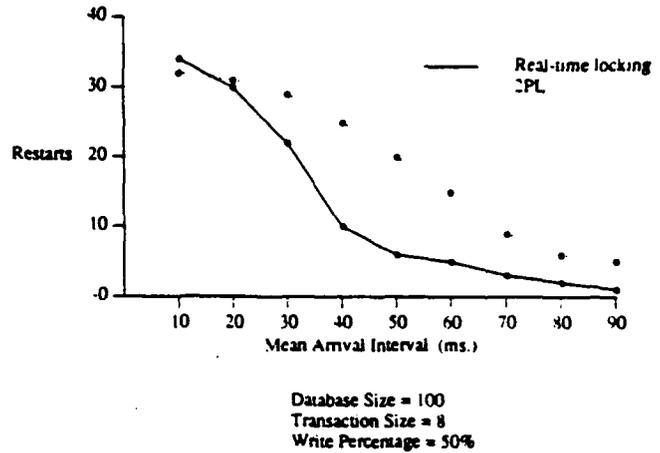


Fig.5 Sensitivity of Restart Number to Mean Arrival Interval

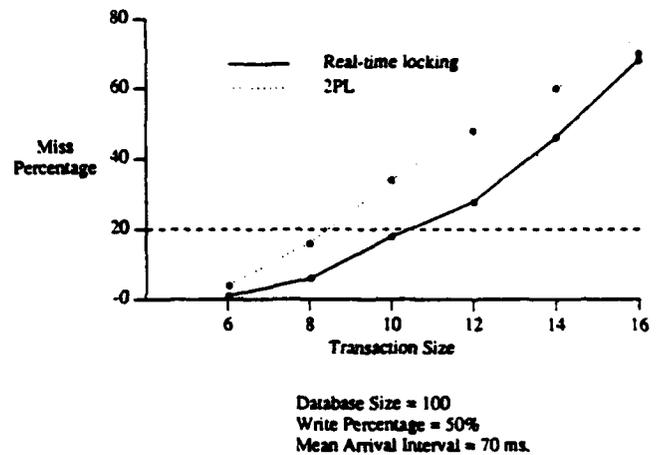


Fig.6 Sensitivity of Miss Percentage to Transaction Size

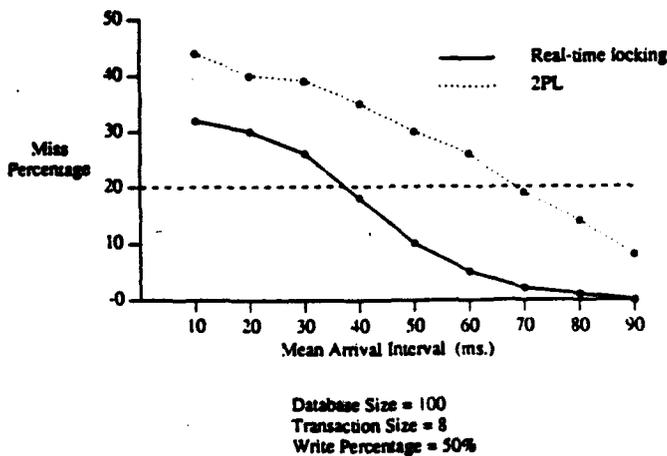


Fig.4 Sensitivity of Miss Percentage to Mean Arrival Interval

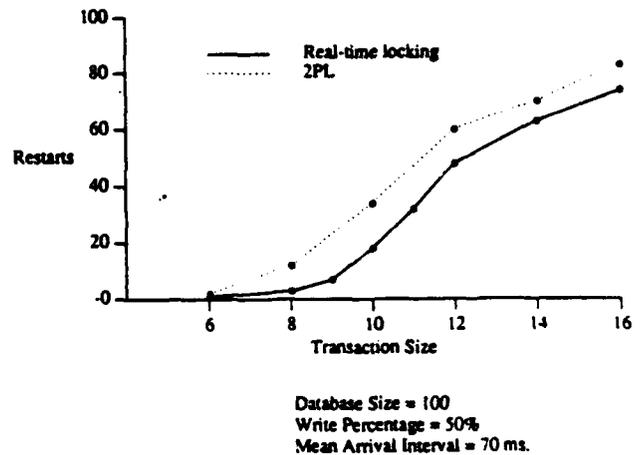


Fig.7 Sensitivity of Restart Number to Transaction Size

An Algorithm for Real-Time Fault-Tolerant Scheduling in Multiprocessor Systems

Yingfeng Oh and Sang H. Son

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

Abstract

In this paper, we consider using hardware and software redundancy to guarantee task deadlines in a hard real-time multiprocessor system even in the presence of processor failures. A set of scheduling requirements for the real-time fault-tolerant multiprocessor scheduling problem is first identified and a heuristic algorithm is then proposed to solve the problem. Experimental results show that the algorithm finds optimal solutions in most of the cases.

I. Introduction

Hard real-time systems are defined as those systems in which the correctness of the system depends not only on the logical results of computation, but also on the time at which the results are produced. Missing a hard deadline in such a system may result in catastrophic consequences, such as immediate danger to human life, severe damage to equipments, or waste of expensive resources. Since hard real-time systems are being increasingly used in many mission-critical and life-critical applications, the fault tolerance of these systems becomes extremely important.

A real-time system which possesses fault-tolerant capability is usually termed a real-time fault-tolerant system. The correctness of a real-time fault-tolerant system requires that timing constraints of computations in the system be met even in the presence of hardware or software faults. Hardware or software faults in a real-time system can lead to timing faults, such as missing a hard deadline, which should be prohibited in a hard real-time system. To tolerate timing faults, several studies have been carried out in achieving fault tolerance in real-time systems through the scheduling of redundant resources, such as replicated tasks and redundant processor power [Anders83] [Balaji89] [Bannis83] [Bertos91] [Liestm86] [Krishn86]. Most of these works, however, focus either on uniprocessor sched-

uling, or on achieving certain scheduling criteria, such as minimizing schedule length or local cost function, or balancing workload distribution. The problem of tolerating processor failures in a hard real-time multiprocessor system has not been sufficiently addressed in the literature.

The investigation of the problem is further necessitated by the fact that many hard real-time systems are being supported by multiprocessor systems. This is mainly due to the following two reasons. First, a multiprocessor system is generally more reliable than a uniprocessor system, because the failure of one processor in a multiprocessor system does not necessarily cause the whole system to fail if some fault tolerance techniques are provided. Second, a multiprocessor system can offer more computational power for hard real-time systems than a uniprocessor system. However, with these advantages also comes the disadvantage of more likelihood of processor failures as more processors are used. A multiprocessor system can be less reliable than a uniprocessor system if one processor failure can cause the whole system to fail. This can happen if no fault-tolerant capability is provided. Thus, a processor failure in a hard real-time multiprocessor system is a very serious problem, which ought to be tackled. In this paper, we address this problem by formally defining it as a real-time fault-tolerant scheduling problem and then propose a heuristic algorithm to deal with processor failures.

The rest of the paper is organized as follows: Section II defines the real-time fault-tolerant multiprocessor scheduling problem. A heuristic scheduling algorithm is presented in Section III. The analysis and performance evaluation of the algorithm are described in Section IV. Section V concludes the paper and suggests future work.

II. Problem Statements

We assume that processors fail in the fail-stop manner and the failure of a processor can be detected by other processors. All periodic tasks arrive at the system in one cycle T , i.e., having the same period and are ready to execute any

time within each cycle. We further assume that all periodic tasks have hard deadlines and their deadlines have to be met even in the presence of processor failures. We define a task's meeting its deadline as either its primary copy or its backup copy finishes before or at the deadline. Because the failure of processors is unpredictable and there is no optimal dynamic scheduling algorithm for multiprocessor scheduling [Dertou89], we focus on static scheduling algorithms to ensure that the deadlines of tasks are met even if some of the processors might fail. The scheduling problem is defined as follows:

The Scheduling Problem: A set of n periodic tasks $S = \{T_1, T_2, \dots, T_n\}$ is to be scheduled on a number of processors. For each task i , there are a primary copy P_i and a backup copy B_i associated with it. The computation time of a primary copy P_i is denoted as C_i , which is the same as the computation time of its backup copy B_i . The tasks are independent of each other. The scheduling requirements are given as follows:

- (1) Each task is executed by one processor at a time and each processor executes one task at a time.
- (2) All periodic tasks should meet their deadlines. Aperiodic tasks have soft deadlines.
- (3) Maximize the number of processor failures to be tolerated.
- (4) For each task i , the primary task P_i or the backup B_i is assigned to only one processor for the duration of C_i and once it starts, it runs to its completion unless a failure occurs.
- (5) The number of processors used should be minimized.

The deadlines of aperiodic tasks are assumed to be soft. However, as we will show later, the execution of aperiodic tasks are taken into account. Thus, in a normal execution situation, aperiodic tasks are able to meet their deadlines. We further assume that all the processors are identical. Requirement (1) specifies that there is no parallelism within a task and within a processor. Requirement (2) dictates that the deadlines of periodic tasks should be met, maybe at the expense of more processors. Requirement (3) is a very strong requirement. The primary and backup tasks should be scheduled on different processors such that any one or more processor failure will not result in the missing of the hard deadlines of the periodic tasks. Furthermore, the primary copy and the backup copy of a task should not overlap each other, as we shall see in Lemma 2. Requirement (4) implies that tasks are not preemptive. A processor is informed the failure of other processors only at the end of the execution of a task. Also, care has to be taken to ensure that exactly one of the two copies of a task is executed during a cycle to minimize the wasted work. Requirement (5)

states that the number of processors to be used to execute the tasks should be the smallest possible.

Since no efficient scheduling algorithm exists for the optimal solution of the fault-tolerant real-time multiprocessor scheduling problem as defined above, we resolve to a heuristic approach. A heuristic algorithm based on a bin packing algorithm is used to obtain approximate solutions. Before presenting the heuristic, we state the following Lemmas as the basic results upon which the scheduling heuristic is developed.

Lemma 1: In order to tolerate one or more processor failures and guarantee that the deadline of all the periodic tasks are met using the primary-backup copy approach, the longest computation time of the tasks must satisfy the following condition, where T is the period of tasks: $(C_j = \max_{1 \leq i \leq n} \{C_i\}) \leq T/2$.

Proof: Suppose that the deadline of the task T_j can still be met even if $C_j > T/2$. Suppose the processor which executes T_j fails at the time of $T/2$ and the backup task B_j is immediately started, then the finishing time of T_j is $BF_j = T/2 + C_j$. As $C_j > T/2$, we have $BF_j > T$, i.e., the deadline of the task is missed. This is a contradiction. Δ

Lemma 2: One arbitrary processor failure is tolerated and the deadlines of tasks are met with the minimum number of processors possible, if and only if the primary copy P_i and the Backup copy B_i of task i is scheduled on two different processors and there is no overlapping between them.

Proof: In [Lawler81], it is shown that a set of periodic tasks is schedulable on a multiprocessor if and only if there exists a valid schedule which is cyclic with a period T ; i.e., each processor does exactly the same thing at time t as it does at time $t+T$. Therefore it suffices to consider the execution of tasks within a period T only. We first prove the necessary condition. Suppose one arbitrary processor failure is tolerated. It is evident that the primary copy of a task and its backup copy should be scheduled on two different processors. To prove that there is no overlapping between the primary copy of a task and its backup copy, we define BB_i as the beginning time of the backup copy B_i and FP_i as the finishing time of the primary copy P_i . If there is an overlapping between the primary copy of task i and its backup copy, then $FP_i - BB_i > 0$. Suppose the processor k on which the backup copy B_i of task i is assigned has no unused time within a period and the processor j on which the primary copy is executed fails at time $t > BB_i$. Processor k can only be notified of the failure of processor j no earlier than t . Thus the finishing time of the whole schedule of processor k is lengthened by $t - BB_i > 0$, resulting in a missed deadline. To prove the sufficient condition, we have that any pair of primary and backup copies are scheduled on two processors and there is no overlapping between them.

Then the failure of any one of the two processors will trigger the execution of the backup tasks on another processor. Thus the deadline of the tasks will be met. Δ

III. The Scheduling Algorithm

The basic idea of using primary-backup copy approach to tolerate processor failures is that there are two copies associated with each task, i.e., the primary copy and the backup copy. Once the primary copy fails, the backup copy is activated. Since the possible execution of the backup copies should also be finished before the deadline, enough time must be reserved on each processor to execute the backup copies. The reservation of enough time for the execution of backup copies implies that redundant processors have to be used to execute the primary task set earlier enough so that once a processor failure occurs, there will be time to execute the backup copies.

Our scheduling algorithm is based on the First-Fit Decreasing (FFD) bin packing heuristic. In the FFD algorithm for bin packing, the bins are numbered from 1 to M and the items, pre-sorted into decreasing order of size, are packed sequentially, each going into the lowest numbered bin in which it will fit. In our algorithm, we regard processors as bins and tasks as items having sizes equal to their computation times. As shown in Lemma 1, the computation times of all tasks should be less than half of the period in order to tolerate at least one arbitrary processor failure. Because the deadline of the tasks are known a priori to be T , T is used as the size of bins for the FFD heuristic.

The scheduling algorithm proceeds as follows: First, the primary tasks are arranged in the order of decreasing computation times, denoted as P_1, P_2, \dots, P_n . Second, the FFD heuristic is used to schedule the primary copies of the tasks into bins with size T . More specifically, we begin with one processor. Once the assignment of a task fails for the existing processors, a new processor is added. Tasks are assigned to processors in the order of their decreasing computation time. In other words, task P_i is scheduled before task P_j , where $i < j$. Task P_i is assigned to the lowest-indexed processor on which its finishing time is less than the period T . The schedule thus obtained is called the primary schedule. Let the number of processors required be m . It is apparent that though the tasks are schedulable to finish before the deadline, at least one of the tasks will miss its deadline if there is a failure. Therefore, the following steps are necessary. Third, the primary schedule is duplicated on another set of m processors to form the backup schedule. The tasks in the backup schedule are swapped based on the swapping rules to be defined below. Fourth, the tasks in the two schedules--primary and backup schedules are all renamed according to the following renaming rule, such that the primary schedule uses $2 \times m$ processors

and precedes the backup schedule, and there is no overlapping between any pair of primary and backup copies of tasks.

By summarizing what we described above, we state the algorithm as follows.

procedure scheduler (Task Set, Period T);

Sort the set of tasks in the order of decreasing computation time and rename them P_1, P_2, \dots, P_n ;

Apply FFD (First-Fit Decreasing) to assign the set of tasks into m processors;

Duplicate the schedule on m backup processors to form the backup schedule;

Applying swapping rules to the backup schedule;

Applying the renaming rule to both the primary schedule and the backup schedule;

end scheduler;

In the following, we define the rules precisely and prove that a schedule produced by applying these rules can tolerate one arbitrary processor failure.

Definition 1: For the schedule on each processor, L_p is defined as the length of schedule less than or equal to half of the period T such that it is the sum of the computation times of those tasks whose finishing times are less than or equal to half of the period. L_q is the length of schedule for a processor. L_r is defined as the $L_q - L_p$. Obviously, $L_q \leq T$ and $L_p \leq T/2$, as illustrated in Figure A. From now on, where no confusion can be incurred, L_p is also used to denote the time interval whose length is L_p . L_q and L_r are also used in the similar manner.

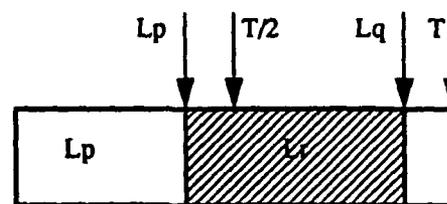


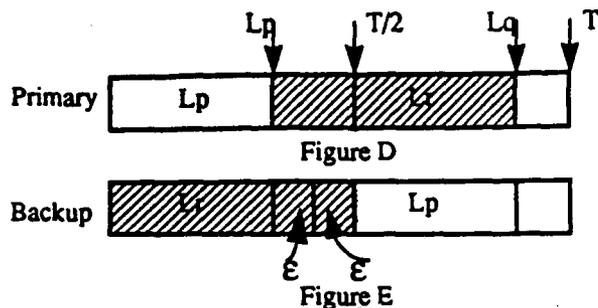
Figure A

In Figure 1, for example, L_p and L_q are equal to 5 and 9 respectively for processor 1. For processor 2, L_p is 4, and L_q is 9.

With the definition of L_p as above, the swapping rules for each processor in the backup schedule can be described as follows:

Swapping Rules:

- (1) Tasks in L_p and tasks in L_r are swapped together.



Figures D & E: Any twin schedules after swapping but before renaming

Case 2: $L_p \leq L_r$, as shown in Figures D & E.

The tasks in L_p are swapped with the tasks in L_r . First we claim that there are at least two tasks in L_r . Suppose there is only one task in L_r . Because $L_p < L_r$, i.e., the computation time of any task in L_p is shorter than the computation time of the only task in L_r , this contradicts the FFD algorithm for assigning tasks in the order of decreasing computation time to processors. Therefore there are at least two tasks in L_r . We further claim that if there is no overlapping between the first primary copy in L_r of Figure E and its backup copy in L_r of Figure D, there is no overlapping between the primary copy of any task and its backup copy on the twin processors. Suppose task w is one of the tasks, but not the first task in L_r of Figure E and its primary copy overlaps with its backup copy in L_r of Figure D. Then the computation time of task w must be longer than that of the first task in L_r . This again contradicts the rules used by FFD to assign tasks in the order of decreasing computation time to processors. Now suppose that the first primary task in L_r of Figure E overlaps with its backup task in L_r of Figure D for the length of $\epsilon > 0$ time unit, then the computation time of this task is $L_p + \epsilon > L_p$ which again contradicts the rule used by FFD to assign tasks to processors. We have shown that there is no overlapping between the primary copy of any task in L_r of Figure E and its backup copy in L_r of Figure D. Since $L_p \leq L_r$, the primary copy of any task in L_p of Figure D can not overlap with its backup copy in L_p of Figure E.

From the above two cases, it is clear that for any pair of twin processors, one arbitrary processor failure is tolerated and the deadlines of the tasks are guaranteed. Δ

IV. Analysis and Performance Evaluation

It is apparent that the scheduling algorithm meets the scheduling requirements identified in Section II. In the worst case, only one processor failure can be tolerated. In the best case, up to $\lfloor m/2 \rfloor$ processor failures can be tolerated, where m is the total number of processors used.

Though the main focus of our scheduling algorithm is to guarantee tasks with hard deadlines to meet their deadlines even in the presence of processor failures, tasks with soft deadlines still have ample time for execution if there is no processor failure or the number of processor failures is small. This is achieved through the scheduling of primary copies to finish around half of the period.

The time complexity of the algorithm is $O(nm)$ if the tasks have already been sorted according to their computation times. The sorting can be done in $O(n \log n)$ time. Thus, the complexity of this algorithm is dominated by the sorting process.

Because the multiprocessor scheduling problem is known to be *NP*-complete, we are hopeless in finding an optimal solution to the problem even when the number of tasks is small (e.g. 10). Thus, we consider the most ideal case, which we call "best possible". The number of processors used in the most ideal case is the result of taking the ceiling of the result of dividing the sum of computation times of all the tasks (primary and backup) by the cycle. The performance of the scheduling algorithm and the "best possible" case is shown in Figure 4. The computation time of each task is randomly generated from the range of [1, 20]. The period T is 90 time units. As we can see, there is only one processor difference in most of the cases. In other words, the scheduling algorithm finds near optimal solutions.

V. Conclusion

In this paper, we have identified the real-time fault-tolerant multiprocessor scheduling problem and proposed an efficient scheduling algorithm to solve it. Experiment results show that the scheduling algorithm finds near-optimal solutions. We have also shown that one arbitrary processor failure can be tolerated by the scheduler.

There are many open questions which need to be answered in order to design extremely reliable hard real-time systems. The case where tasks have different periods in the scheduling problem is still an open problem. Another problem remains open where the processors available in the system are all uniform processors (the speeds of the processors have linear relations). These are the topics for our future research.

References

[Anders83] Anderson, T. and Knight, J.C., A Framework for Software Fault Tolerance in Real-time Systems, IEEE on Software Engineering, SE-9(3), May 1983, pp. 355-364.
 [Balaji89] Balaji, S. et al., Workload Redistribution for

Fault-Tolerance in a Hard Real-Time Distributed Computing System, FTCS-19, Chicago, Illinois, pp. 366-373, June 1989.

[Bannis83] Bannister, J.A. and K. S. Trivedi, K.S., Task Allocation in Fault-Tolerant Distributed Systems. Acta Informatica, 20, Springer-Verlag, 1983.

[Bertos91] Bertossi, A.A. and Mancini, L., Fault-Tolerant Task Scheduling in Multiprocessor Systems, Tech. Report, Universita di Pisa, Italy, 1991.

[Dertou89] Dertouzos, M. and Mok, A.K-L, Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks, IEEE Trans. on Computer, 15(12), December 1989, pp. 1497-1506.

[Krishn86] Krishna, C.M. and Shin, J.C., On Scheduling Tasks with a Quick Recovery from Failure, IEEE Transactions on Computers, C-35(5), May 1986, pp. 448-454.

[Lawler83] Lawler, E.L. and Martel, C.U., Scheduling Periodically Occurring Tasks on Multiple Processors, Information Processing Letters, 12(1), 1981, pp. 9-12.

[Liestm86] Liestman, A.L. and Campbell, R.H., A Fault Tolerant Scheduling Problem, IEEE Transactions on Software Engineering, SE-12(11), November 1986, pp. 1089-1095.

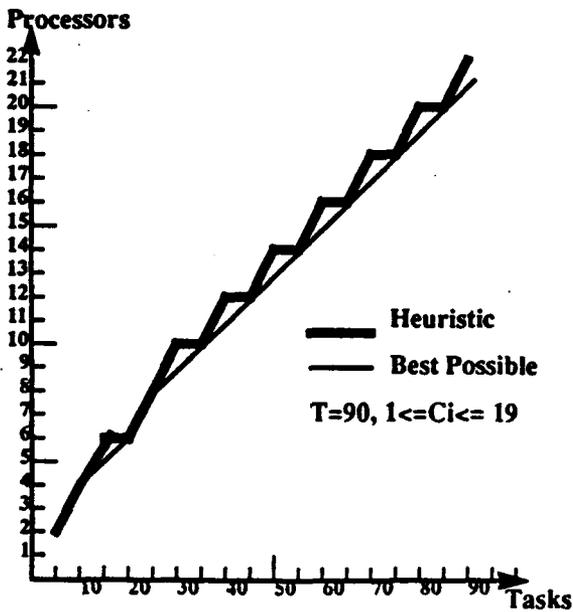


Figure 4: Performance of Heuristic

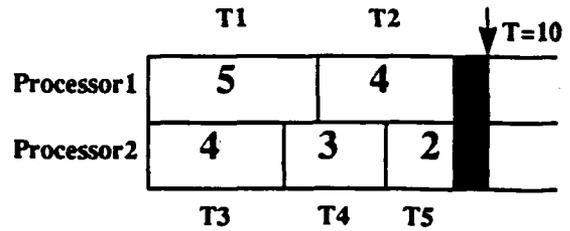


Figure 1

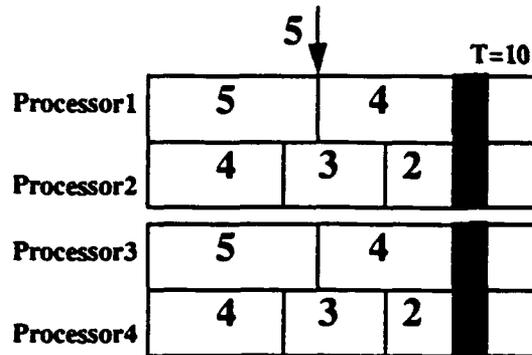


Figure 2

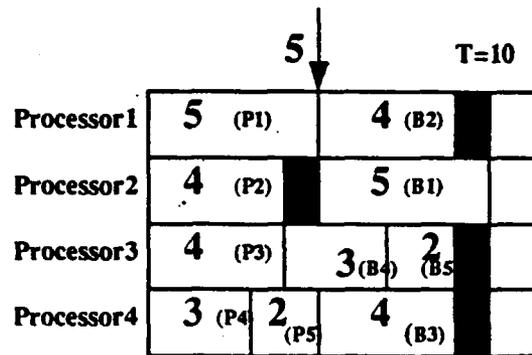


Figure 3

Integration of a Database System with Real-Time Kernel for Time-Critical Applications

Sang H. Son, Stavros Yannopoulos, Young-Kuk Kim, and Carmen C. Iannacone⁺

Dept. of Computer Science
University of Virginia
Charlottesville, VA 22903, USA

Abstract

Transactions in real-time database systems should be scheduled considering both data consistency and timing constraints. Since a database system must operate in the context of available operating system services, an environment for database systems development must provide facilities to support operating system functions and integrate them with database systems for experimentation. We chose the ARTS real-time operating system kernel. In this paper we present our experience in integrating a relational database manager with a real-time operating system kernel and our attempts at providing flexible control for concurrent transaction management. Current research issues involving the development of a programming interface and imprecise computing server are also discussed.

1. Introduction

Time is the key factor to be considered in real-time database systems, and 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 deadlines expire. For example, both the update and query on the tracking data for a missile must be processed within given deadlines, satisfying not only database consistency constraints but also timing constraints.

Conventional database systems are typically not used in real-time applications due to the inadequacies of poor performance and lack of predictability. They are designed to provide good average performance, while possibly yielding unacceptable worst-case response times. In addition, conventional database systems do not schedule their transactions to meet response requirements and they commonly lock data tables to assure

only the consistency of the database. Locks and time-driven scheduling are basically incompatible, resulting in response requirement failures when low priority transactions block higher priority transactions. New techniques are required to manage the consistency of real-time databases, and they should be compatible with time-driven scheduling and meet both the required temporal constraints and data consistency.

To address the inadequacies of current database systems, the transaction scheduler needs to be able to take advantage of the semantic and timing information associated with data objects and transactions. A model of real-time transactions needs to be developed which characterizes distinctive features of real-time databases and can contribute to the improved responsiveness of the system. The semantic information of the transactions investigated in the modeling study can be used to develop efficient transaction schedulers [Son90b, Son91].

A database system must operate in the context of available operating system services, because correct functioning and timing behavior of database control algorithms depend on the services of the underlying operating system. As pointed out by Stonebraker, operating system services in many systems are not appropriate for support of database functions [Ston81]. In many areas, such as buffer management, recovery, and concurrency control, operating system facilities have to be duplicated by database systems because they are too slow or inappropriate. An environment for database systems development must, therefore, provide facilities to support operating system functions and integrate them with database systems for experimentation.

The ARTS real-time operating system kernel under development at Carnegie-Mellon University attempts to provide a "predictable, analyzable, and reliable distributed real-time computing environment" which is an excellent foundation for a real-time database system [Tok89]. The ARTS system, which provides support for programs written in C and C++, implements different prioritized and non-prioritized scheduling algorithms and prioritized message passing as well as

This work was supported in part by ONR under contract N00014-91-J-1102, by Naval Ocean Systems Center, by DOE, and by IBM Federal Systems Division. ⁺ Currently employed with SRA Corporation.

supporting lightweight tasks. All of these features are important when considering a real-time database.

We have investigated the issues for integrating real-time database systems with operating system kernels, such as the features a database system requires from real-time operating system kernels to provide real-time transaction support. We have used the relational database technology since it provides the most flexible means of accessing distributed data. Our research effort resulted in a new database manager for distributed real-time systems on top of ARTS. The result, RTDB, consists of a multi-threaded server which accepts requests from several clients. It provides a three-tiered approach for supported media types, offering memory-resident data options, local disk storage, and access to the UNIX file system. This support of various media types provides developers the flexibility to choose appropriately those that best suit their needs. In addition, we have incorporated the notion of *imprecise computation* [Liu90] into RTDB to produce meaningful results before the deadline of a task by trading off the quality of the results for the computation time of the task. In this paper we present our experience in integrating a relational database manager with a real-time operating system kernel, and our attempts at providing flexible control for concurrent transaction management using a technique called *workload mediation*. Current research issues involving the development of a programming interface and temporal consistency are also discussed.

2. Comparison with Other Systems

One of the principal goals of the ARTS project is to provide a more easily extensible real-time environment than is currently enjoyed by programmers developing on other kernels. To that end, ARTS requires better data management facilities than many other kernels offer. RTDB on ARTS represents a combination of desirable aspects of database technology and development flexibility. In comparing RTDB with other existing systems, we note some differences between the approach we are taking and that of other research or commercial products.

In many cases, real-time database systems should provide facilities to process concurrent transactions from multiple users. It requires protocols and algorithms for transaction scheduling and concurrency control. In real-time transaction scheduling, the actual execution order of operations is determined by two factors: priority order and serialization order. The difficulties in real-time transaction scheduling arise from the fact that these two factors have different natures and are constructed in different ways. While the serializable execution order is strictly bound to the past execution

history, the priority order does not reflect the past execution history and may dynamically destroy the order set up in the past execution, hence serializability. Most research efforts on real-time database systems concentrate on scheduling algorithms to solve the conflicts among multiple real-time transactions in multi-user environment. However, some systems assume a single user, dedicated processor database system. In such an environment, there is no need to schedule multiple real-time tasks on a single processor.

For example, CASE-DB is developed as a single-user, disk-based, real-time relational DBMS, which uses the relational algebra as its query language [Ozso90]. In CASE-DB, it is assumed that the probability that the query is not executable within the deadline is near to 1. To process this kind of real-time queries by the given deadline, they restrict the queries using several techniques, such as sampling scheme for aggregate queries. Since real-time database grows by time quickly, even for periodic query, the processing time can be different depending on the number of scanned tuples or the size of the relation. Thus the worst case execution time of a transaction can be hard to determine or impractical. Nevertheless, the question is how practical the assumptions made in CASE-DB would be in actual real-time system environments. Furthermore, this system assigns priority to the part of a relation ("fragment set"), instead of assigning a priority to a query. Then, the remaining problem is how to agree a priori on semantically meaningful subset of each relation. RTDB diverges from this design philosophy in many ways, being a multi-user, distributed real-time DBMS.

Supported media types also differ among real-time database systems. HP-RTDB, one of Hewlett Packard's Industrial Precision Tools, provides software application developers with a tool to structure and access memory-resident data. Essentially, HP-RTDB is a library of routines used to define and manipulate a database schema, build the database in memory, as well as load and unload, and write or read data to and from it. It also provides mechanisms for archiving schema and data and storing timestamp information. ARTS-RTDB supports a three-tiered approach to data storage. The user's options for data storage include memory-resident relations, RAM-based disk relations, and storage on the UNIX file system. Each media has its own advantages and drawbacks in terms of predictability, performance, and recoverability. The relation media abstraction is demonstrated in Figure 1 which depicts the ARTS-RTDB testbed at the University of Virginia. Naturally, access times decrease along this continuum. This support of various media types provides developers the flexibility to choose appropriately those that best suit their needs. Also, we provide the ability to cross the

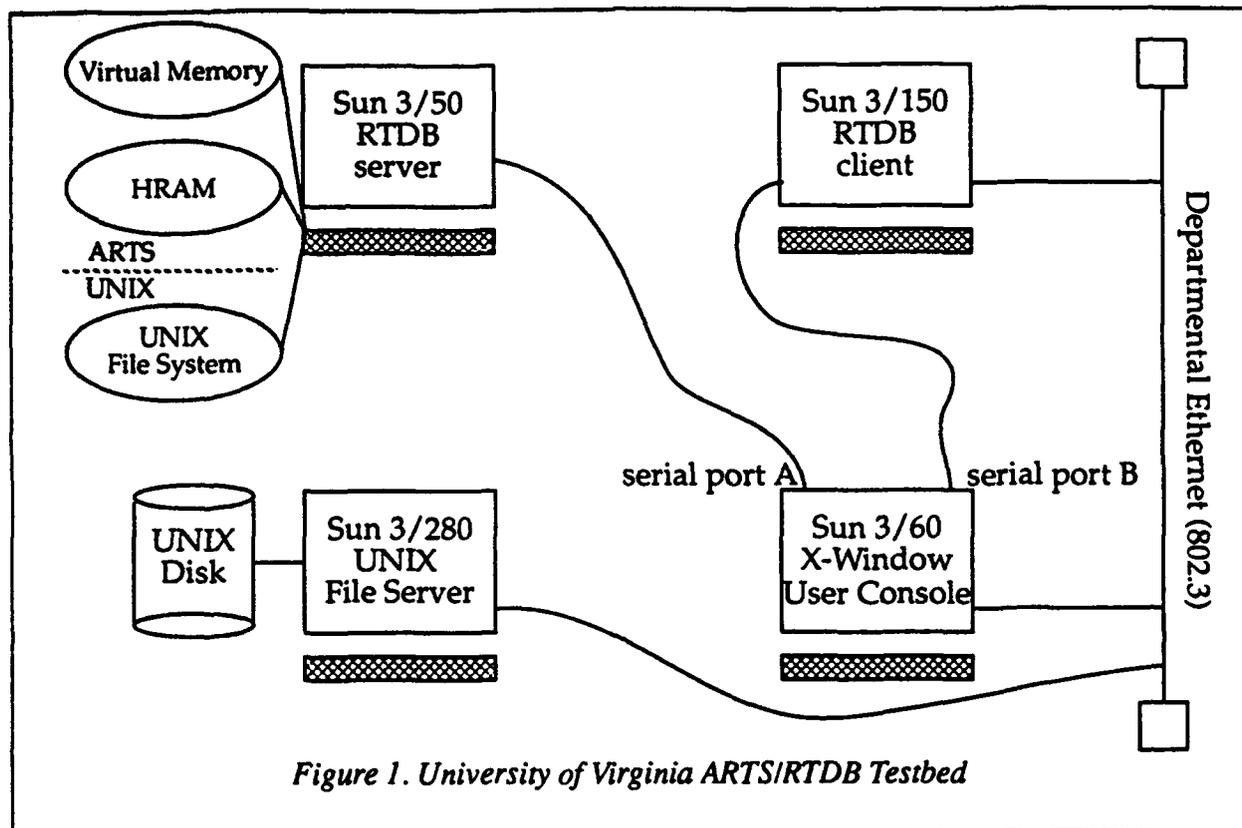


Figure 1. University of Virginia ARTS/RTDB Testbed

boundaries between these media, and to utilize several media types in an individual query for both the source and resultant relations. A detailed discussion on the performance of those media types are reported in [Son91b].

3. The ARTS Real-Time OS Kernel

Research in the area of distributed, real-time operating systems indicates that most are designed for a specific need, and as such are difficult to build, maintain, and modify; in addition, they do not afford the capability of predicting runtime behavior during application design. In fact, few non-real-time operating systems provide a functionally complete set of general purpose, real-time task and time management functions, despite the fact that the user community is expressing the desire for increasingly complex applications of this type. Since the success of applications in real-time computing is primarily contingent on a system's temporal functionality, what is needed is an environment in which the system engineer can analyze and predict, during the design stage, whether the given real-time tasks having various types of system and task interactions (i.e. memory allocation/deallocation, message communications, I/O interactions, etc.) can meet their timing requirements.

In an attempt to provide such functionality, ARTS provides the process and data encapsulation that

other distributed, object-oriented operating systems do, while at the same time including elements of temporal significance to the services it provides. This integration of data, thread, and concurrency control greatly facilitates real-time schedulability analysis. ARTS can support both hard and soft real-time tasks as well as periodic and sporadic ones [Tok89].

To support time-critical operations, the ARTS programming language interface allows designers to specify timing requirements and the chosen communication structure so that they are visible at both the language and system level; this allows the system-wide ARTS environment to make scheduling decisions based on both temporal constraints and priorities of transactions. The Integrated Time-Driven Scheduler (ITDS) model of ARTS is more effective than the common priority-based preemptive scheduling of many real-time systems. Such simple schedulers become confused during heavy system loads when they cannot decide which tasks are important and should be completed and which tasks should be aborted, causing unpredictability in the applications. The ITDS model, however, employs a time-varying "value function" which specifies both a task's time criticality and semantic importance simultaneously. A hard real-time task can be characterized by a step function where the discontinuity occurs at the deadline, while soft real-time tasks are described by continu-

ous (linear or nonlinear) decreasing function after its critical time. In addition, ARTS' designers have separated the policy and mechanism layers, so that users can implement new scheduling policies with a minimum of effort, and even dynamically changing the policy during runtime.

The issue of priority inversion is crucial to providing semantically correct system behavior in addition to addressing temporal concerns. Priority inversion occurs when a high priority activity waits for a lower priority activity to complete. Resource sharing and communication among the executing tasks can lead to priority inversion if the operating system does not manage the available resource set properly. Significant research in the construction of ARTS was done to avoid priority inversion among concurrently executing tasks; in the processor scheduling domain, low priority servers which provide service to clients of all priorities are susceptible to inversion. For example, when a low priority request is being serviced, and a high priority task requests the same service, the high priority request waits, since the server's computation is non-preemptable. Any task of higher priority than the server may preempt the server itself, however, so if a medium priority task arrives it preempts the server indefinitely, causing the high priority job to be lost in the shuffle. ARTS employs a priority inheritance mechanism to propagate information about a single computation which crosses task boundaries. That is, if a server task accepts the request of a client, the server inherits the priority of the client. Furthermore, the server should also inherit the priority of the highest priority task waiting for the service.

The notion of time encapsulation cannot be divorced from the basic structure of ARTS, in which every computational entity is represented as an object, called an *artobject*. An *artobject* is defined as either a passive or an active object. In a passive object there is no explicit declaration of a thread which accepts incoming invocation requests while an active object contains one or more threads defined by the user. In an active object, its designer is responsible for providing concurrency control among coexecuting operations. When a new instance of an active object is created, its root thread will be created and run immediately. A thread can create threads within its object.

The ARTS kernel supports the notion of real-time objects and real-time threads. A real-time object is defined with a "time fence," a timer associated with the thread which ensures that the remaining slack time is larger than the worst case execution time for the operation. A real-time thread can have a value function and timing constraints related to its execution period, worst-case execution time, phase, and delay value. When an

operation with a time fence is invoked, the operation will be executed (or accepted) if there is enough remaining computation time against the specified worst case execution time of the operation for the caller. Otherwise, it will be aborted as a time fence error. The objective of this extension to a normal object paradigm is to prevent timing errors from crossing task or module boundaries (as often happens in traditional real-time systems which use a cyclic executive) and to bind the timing error at every object invocation.

On top of the ARTS foundation we have built a relational database manager using message passing primitives and employing the client/server paradigm. The result, RTDB, currently consists of a multi-threaded server which accepts requests of several clients. Based on the temporal urgency of the request, the server determines whether it can commit the transaction or if it has to reject it.

4. The RTDB Real-Time Database Manager

RTDB is a relational database manager written in a hybrid C-based language called ARTS/C++ designed to run on ARTS. It offers not only a functionally complete set of relational operators — such as join, projection, selection, union, and set difference — but also other necessary operators such as create, insert, update, delete, rename, compress, sort, extract, import, export, and print. These operators give the user a good amount of relational power and convenience in managing the database.

We have developed two different kinds of clients for RTDB. One is an interactive command parser/request generator that makes requests to the server on behalf of the user. This client looks and behaves similarly to a single-user database manager. It is possible to run the client without knowing that any interaction between server and client is occurring. The other client is a transaction-generating "batch" client, representing a real-time process that needs to make database access requests.

The RTDB server object is the heart of the database management system. It is responsible for creating and storing the relations, receiving and acting on requests from multiple clients, and returning desired information to the clients.

The server object defines three threads. The root thread is an aperiodic thread, which is automatically executed by ARTS upon invocation of the server. The server activates one or more worker threads. The worker threads are aperiodic and each one has a different priority which will match the priority of the messages it will service. The backup thread is a low priority periodic thread responsible for periodically backing up

the relations that reside only in main memory.

The root thread of the server is responsible for binding the server's name in the ARTS name server so that the clients can find it and send requests. It is also responsible for reading the relations into its local memory, initializing the lock table and the blocked request queue, instantiating the backup thread and the server worker threads. There is usually one worker thread created for each priority level. After completing these tasks, the root thread enters an infinite loop that accepts database requests from any client. The requests come in as packets. RTDB provides two different types of packets: *call* packets and *return* packets. The call packet, created by a client, contains all the information that the server needs to carry out the desired database access operation. Since different commands require different information, the call packet has a variant field containing different information for each command. When the server completes the processing of the request, it returns a packet to the client with the information requested. This packet is called a return packet. The return packet is created by the server and also has a variant field that carries command specific information.

The communication between the server and clients is performed by the ARTS communication primitives: Request, Accept, and Reply. The communication is synchronous; when a client issues a Request, it is blocked until the server Accepts and Replies to the message. This may cause some problems, especially in a real-time environment, for two reasons: priority inversion and data sharing.

The ARTS kernel (and thus the RTDB system) supports eight message priorities. When the root thread Accepts a message, it extracts priority information from the message packet. The root thread then enqueues the request on the message queue (i.e. pending request queue) of the worker thread designated to service requests of that priority level. If inactive, the worker thread will be polling its queue; if active, the requests will be processed in FIFO order. Note that in this way we can easily exploit the scheduling merits of the underlying ARTS kernel without circumventing its priority-based scheduling mechanism. Since the worker thread's priority matches that of the messages it services, it will only be scheduled for the CPU in an interval where its priority is currently the highest in the system. This is a general case; for those instances where the scheduling technique is not priority based, or ARTS priority inheritance mechanism is employed, these decisions will naturally be reflected in the workers.

This technique of distributing requests among a pool of workers based on information contained in the request packet is called *workload mediation*. In our system workload mediation is realized by the server root

thread which accepts the messages from the clients and puts them in the appropriate worker queue according to their priority. It is intrinsic in implementing various transaction prioritizing algorithms which utilize semantic information provided by the clients and/or the database transaction requests such as user-entered runtime estimates, deadline constraints, or command-to-priority mappings. Determining the proper balance of control between ARTS primitives and RTDB explicit mediation will help us achieve the most beneficial symbiosis of the system's resources. Figure 2 illustrates the mediator mechanism incorporated within the server object.

This mechanism is unobtrusive from the view of system-wide scheduling, because it does not do any scheduling on its own. It only breaks the incoming workload up among the worker pool. Controlling what criteria are used to make this static assessment is important, and Table 1 indicates the techniques we are investigating. The ARTS OS provides eight priority levels. In the first two cases, there are as many worker threads as priority levels and the mapping is direct; in the first case,

Strategy	Parameters
direct map, object	client object priority
direct map, message	client message priority
modulo map, object	client object priority
modulo map message	client message priority
command mapping	requested command type
complexity mapping	requested command complexity rating
site/node mapping	client's host network id number
media based mapping	media type(s) housing relation(s) in query

Table 1

Workload mediation strategies and their parameters

according to message priority, and in the second case, according to the priority of the client that sent the message. In the next two cases, the priority levels are a multiple of the available workers so the priority of the incoming message (case 3) or that of the client that sent it (case 4) has to be divided by a specific number before it is put in the appropriate worker queue. In the command mapping, the message or client priority is ignored and instead the priority that has been preassigned to each one of the database operations determines the worker that will process the request. The next case (complexity mapping) is a variation of the previous one where priority is determined according to previously calculated complexity rates for each operation. Site/node mapping maps each node participating in the distributed system to one worker. Finally, media based mapping maps the request according to the media

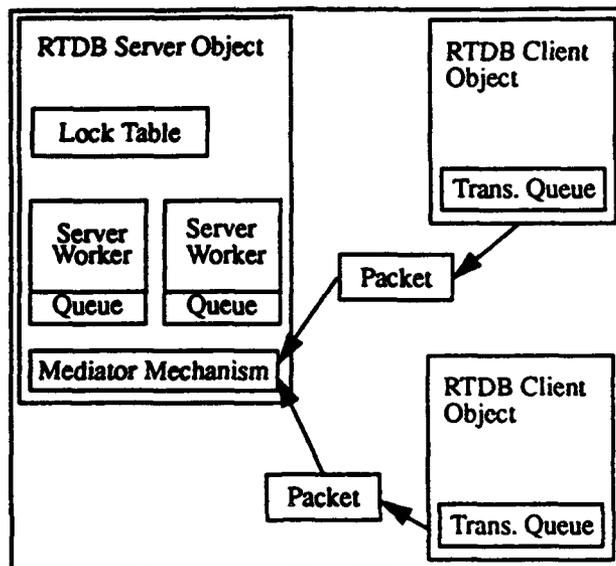


Figure 2. Mediator as internal server object

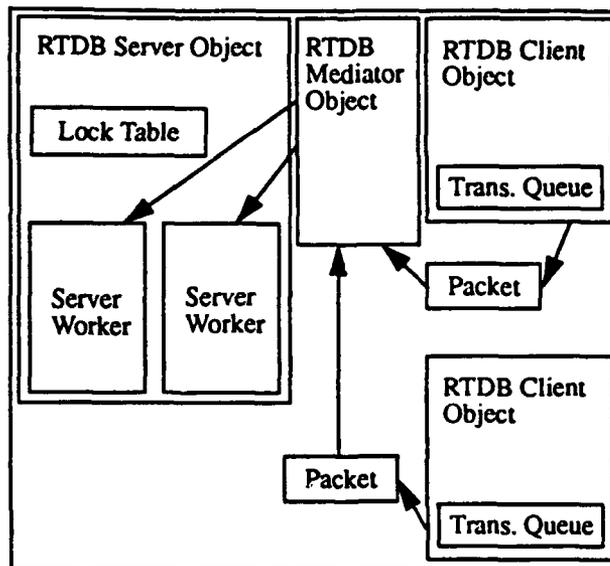


Figure 3. Mediator as separate object

type(s) (virtual memory, hram, file system) housing the relation(s) used in the request. We try as much as possible to make mediation invisible and uncontrollable from the vantage point of the clients, primarily for security reasons-- the server has the ability to ignore a query's requested priority to prevent users from improperly seeking a higher priority than they deserve. Also, since the server is in a better position to analyze the current situation than remote users, allowing the server to monitor itself minimizes the amount of spurious parameters that might enter the mediation algorithms.

Another method of implementing this technique involves creating a dedicated object which acts as an intermediary between the RTDB client and server (Figure 3). The advantages of this implementation are enhanced abstraction for the mediation algorithms, and better modeling of the components involved. Also, algorithm implementors have a centralized repository of the routines they require and a strong definition of the communication interface they are required to maintain. However, it is our contention that the disadvantages of this scheme outweigh its merits. The primary disadvantage is a severe performance decrease due to the overhead of increased inter-object message passing. In addition, many of the data structures needed by the worker threads are also needed by the mediation algorithms.

Returning to our model, the worker thread of the RTDB server performs the client's request to access the database. It checks its request message queue, calls the appropriate database function that executes the requested operation, and replies back to the client. The

worker replies to a client without completing a request when it needs to return more information than can fit in a single packet. In such a case, the worker sets a continuation flag indicating that there is more information to be sent back to the client. The client must make continuation requests to the server until it gets all the information requested. To maintain the consistency of the database, the RTDB server needs to handle conflicting requests properly. For example, a problem occurs when some request or part of a request (as in a multi-relational query) has to be blocked since it needs to lock a relation that is already locked. Our solution is to use a lock table that keeps track of which relations are in use at any given time. Currently, we use a coarse granularity for locks, where the worker locks the file which contains the relation it needs to access. If a request for file A comes in while file A is being used by another active worker, then the new request must be put on an internal queue until A and any other files it needs are available. Using coarse granules incurs low overhead due to locking, since there are fewer locks to manage. However, it also reduces the degree of concurrency, since operations are more likely to conflict. Fine granularity locks (e.g., tuple locks) improve the degree of concurrency by allowing a transaction to lock only those data items it accesses. But fine granularity involves higher locking overhead, since the number of locks requested and that to be maintained will be higher. We are investigating an appropriate granularity level for our database system, including multi-granularity locking mechanisms [Bern87].

Whenever the worker becomes free, it first checks its queue of blocked requests. If there are any

requests in the block queue that can be unblocked, it dequeues the request and processes it. If no request in the block queue is ready to be processed, the worker looks to its incoming request queue.

5. The Programming Interface

Conventional database systems often provide some interface through which they export functionality to application developers. Such programming interfaces simplify storage and retrieval tasks and provide a scheme for the creation, manipulation, and destruction of database files. For systems utilizing the client-server paradigm, communication primitives can also be accessed through such an interface, achieving further hiding of the implementation details.

Programming interfaces in real-time databases differ greatly in terms of application-developer friendliness. Some DBMS interfaces are tightly coupled to theoretical techniques such as the relational algebra. CASE-DB [Ozso90] is an example of this type of interface. While this interface satisfies the desired functionality requirements for a database, it can be awkward to use when developing large, complex applications. For these applications it is more appropriate to use an interface similar to those already in use in non-real-time systems. These application program interfaces consist of library functions.

To facilitate the construction of application clients, we have written an application programming interface (API) for the database command set which hides the implementation details of the system as much as possible. In this way, developers who are more familiar with function-call interfaces can quickly adjust to the task of constructing custom application clients instead of application programs. In addition to providing routines as in other relational databases, we can hide the details of ARTS' Request/Accept/Reply message passing sequence, by developing an appropriate programming interface for RTDB. Lack of such an interface would require the application developer to be familiar with the RTDB message passing mechanism, since it is necessary to ensure correct communication between the application program and the server. Moreover, the user would have to be concerned with scheduling and concurrency issues, because of the multi-user, multi-programming nature of RTDB.

By providing a programming interface, the client and server appear as if the application client were the only one interacting with the server. This goal is only partially attainable, since the physical code provided by the application developer must coexist in the same source code file the as code which specifies constants and declarations necessary to construct the complete cli-

ent image (that is, certain C++ tokens which allow object creation and specification). To expedite the development process, we provide a thoroughly commented, standardized client template with which developers need only combine their source and compile. All the system specific declarations and function calls that the application developer need not be concerned about are coded in the client template. These include the data structure declarations used by the API and all the object and thread declarations and instantiation function calls. When writing an application, the user forms queries by placing operation-specific information into function parameters in a specified format and then calling the appropriate function. This way, when interaction is not needed, a number of database operations can be submitted in batch mode and intermediate results can be manipulated and acted upon in a predefined, user-specified manner, coded in the application program.

We currently support a small subset of database operations through the API, namely: *Create*, *Insert*, *Update*, *Select*. This is a minimal set of operations required to perform experiments on any relational database. We are planning to support a complete interface by providing the full set of database operations currently supported by our interactive client.

6. The RTDB Imprecise Server

Certain real-time applications require that some result of initiated computation be available at a deadline, often at the cost of absolute precision. Much research has been done in this area, aptly named *imprecise computation*. [Liu90]. The concept behind imprecise computation is most often associated with numeric computations whose precision is improved proportionally with the amount of time spent performing the calculation. However, several instances of using this technique with a database merit consideration.

With RTDB, we have created a server object capable of performing imprecise query retrievals. Basically, we provide the client a mechanism to specify a deadline by which a computation (query) must complete. This was easily accommodated by adding a deadline field to the request packet that the client sends to the server. Now the server knows, not only the operation requested, but also the time constraints upon the operation. The server then attempts to complete the query, checking repeatedly at strategic intervals whether it is within danger of missing the specified deadline. If it is not, the server continues working and returns the exact result of the query to the user. If unable to complete the entire query, the server will return imprecise data, provided the computation had proceeded to a point where the output would be meaningful and appropriate. This is

Alloted Execution Time	Number of Tuples Retrieved	Calculated Average
250 milliseconds	38	1344.4737
500 milliseconds	214	2004.9766
750 milliseconds	393	1935.1247
1 second	582	1906.7096
1 second 250 milliseconds	748	2051.6618
1 second 500 milliseconds	922	2195.1106
1 second 750 milliseconds	978	2273.7606

Table 2. Elapsed time, tuple count, and computed results for imprecise server

an important caveat to consider, as returning certain volatile data structures could have serious detrimental effects if permitted. The result is that all tuples returned from a query match the search criteria. However additional matching tuples may also exist in the relation, and they would have been retrieved had sufficient time been allotted.

The intervals at which deadline checks are made were carefully calculated. Each database request is serviced by breaking down the operation in simple primitive operations. This, in turn, requires making appropriate function calls that implement the primitive operations. The execution time of each one of those functions is variable and sometimes it is not enough to check for a missed deadline upon calling and returning from the function. The structure of several functions had been changed in order to provide real-time services, and in many cases, deadline checking had to be performed more than once in a function body. One problem that hinders the transformation of a non-real time function to a real-time one is recursion. Recursive functions are not amenable to being interrupted as easily as iterative functions. Due to these and other minor causes, we have used the state machine approach in representing the execution stages of each function, and to "pump" through the necessary actions with a measurable amount of time allotted to each stage of execution. The state machine approach proved to be very useful because it simplified the analysis of associated routines. It is our hypothesis that the state machine is a useful tool for the real-time database, since it isolates the various stages of action and allows run-time estimates to be computed more easily.

Table 2 indicates an instance where an imprecise computation may yield a satisfactory answer. Here, we issued the aggregate operator "average" on a numeric

attribute in a relation. The relation file involved contained 978 tuples. We initially set the deadline time to be 250 milliseconds from the packet transmission time at the client site and we incremented the deadline interval by 250 milliseconds until a precise answer was obtained.

7. Conclusions and Future Work

A real-time database manager is one of the critical components of a real-time system, in which tasks are associated with deadlines and a significant portion of data is highly perishable in that it has value to the mission only if used in a timely manner. To satisfy the timing requirements, transactions must be scheduled considering not only consistency constraints but also timing constraints. In addition, the system should support a predictable behavior such that the possibility of missing deadlines of critical tasks could be determined ahead of time, before the deadlines expire. Since the characteristics of a real-time database manager are distinct from conventional database managers, there are different kinds of issues to be considered in developing a real-time database manager. For example, priority-based scheduling and memory resident data are two such issues.

In this paper, we have presented an experimental database manager developed for time-constrained distributed systems. The foundation now exists for a real-time relational database manager. We have discussed our work toward providing a flexible programming interface and standard client template to allow quick prototyping and fast modeling. We also have presented our experiences in developing a server based on the notion of imprecise computing. RTDB described in this paper with its multi-threaded server model is an appropriate research vehicle for investigating new techniques

and scheduling algorithms for distributed real-time database systems.

As with any active research project, there are several technical issues associated with real-time database systems that need further investigation. For example, temporal database components are being investigated for inclusion in RTDB. They will address the desired timestamping of surveillance updates generated by radar, sonar, or similar equipment, and temporal consistency requirements of real-time transactions. Other potential improvements in efficient implementation are being examined to determine their overall value to RTDB. Indices and views are two of them. Since such features not only alter the speed and predictability of the system but also the basic file structure, they need to be examined closely on their own and then as new elements within the existing system.

We are also examining inclusion of run-time estimates for various commands within the server which will enable it to offer a choice of service to clients whose work cannot be completed in the time allotted: imprecise results or a missed deadline. Conceivably some clients might wish to simply exclude some queries which might introduce incomplete results, and terminate as quickly as possible. These execution estimates would be maintained in a table in the server and will be based on several factors such as relation file size, query type, media types involved, and current resource utilization. Properly implementing such a heuristic mechanism will require carefully controlled execution timing, and some consideration of the temporal impact of held data locks.

References

- [Abb89] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *VLDB Conference*, August 1989.
- [Buc89] Buchmann, A. et al., "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control," *Fifth Data Engineering Conference*, Feb. 1989, 470-480.
- [Comp91] *IEEE Computer*, Special Issue on Real-Time Systems, vol. 24, no. 5, May 1991.
- [IEEE91] *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, Georgia, May 1991.
- [Kor90] Korth, H., "Triggered Real-Time Databases with Consistency Constraints," *16th VLDB Conference*, Brisbane, Australia, Aug. 1990.
- [Lin90] Lin, Y. and S. H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *11th IEEE Real-Time Systems Symposium*, Orlando, Florida, Dec. 1990, to appear.
- [Liu90] Liu, J. et al., "Algorithms for Scheduling Imprecise Computations," *ONR Annual Workshop on Foundations of Real-Time Computing*, Washington, DC, Oct. 1990.
- [ONR91] *ONR Annual Workshop on Foundations of Real-Time Computing*, Washington, DC, Oct. 1991.
- [Ozso90] Ozsoyoglu, G., et al., "CASE-DB--A Real-Time Database Management System," *Tech. Rep.* Case Western Reserve University, 1990.
- [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.
- [Sha91] Sha, L., R. Rajkumar, S. H. Son, and C. Chang, "A Real-Time Locking Protocol," *IEEE Transactions on Computers*, vol. 40, no. 7, July 1991, 793-800.
- [Son88] Son, S. H., guest editor, *ACM SIGMOD Record* 17, 1, *Special Issue on Real-Time Database Systems*, March 1988.
- [Son90] Son, S. H. and C. Chang, "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment," *10th International Conference on Distributed Computing Systems*, Paris, France, June 1990, 124-131.
- [Son90b] Son, S. H. and J. Lee, "Scheduling Real-Time Transactions in Distributed Database Systems," *7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, Virginia, May 1990, 39-43.
- [Son91] Son, S. H., P. Wagle, and S. Park, "Real-Time Database Scheduling: Design, Implementation, and Performance Evaluation," *International Symposium on Database Systems for Advanced Applications (DASFAA '91)*, Tokyo, Japan, April 1991, 146-155.
- [Son91b] Son, S. H., M. Poris, and C. Iannacone, "Implementing a Distributed Real-Time Database Manager," *The Second International Symposium on Database Systems for Advanced Applications (DASFAA '91)*, Tokyo, Japan, April 1991, 51-60.
- [Ston81] Stonebraker, M., Operating System Support for Database Management, *Commun. of ACM* 24, 7 (July 1981), 412-418.
- [Tok89] Tokuda, H. and C. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM Operating Systems Review*, 23 (3), July 1989.

Replication Control for Distributed Real-Time Database Systems

Sang H. Son and Spiros Kouloumbis

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

ABSTRACT

Schedulers for real-time distributed replicated databases must satisfy two requirements: transactions should meet their timing constraints, and mutual consistency of replicated data should be preserved. In this paper, we propose a new replication control algorithm, which integrates real-time scheduling and replication control. The algorithm adopts a token-based scheme for replication control and attempts to balance the urgency of real-time transactions with the conflict resolution policies. In addition, the algorithm employs epsilon-serializability (ESR), new correctness criterion which is less stringent than conventional one-copy-serializability. The algorithm is flexible and very practical, since no prior knowledge of data requirements or execution time of each transaction is required.

1. Introduction

In *Real-time Distributed Database Systems* (RTD-DBS), transactions must be scheduled to meet the *timing constraints* and to ensure that the replicas remain *mutually consistent* [Son90]. Real-time task scheduling can be used to enforce timing constraints on transactions, while concurrency control is employed to maintain data consistency. Unfortunately, the integration of the two mechanisms is non trivial because of the trade-offs involved. Serializability may be too strong as a correctness criterion for concurrency control in database systems with timing constraints, for serializability limits concurrency. As a consequence, data consistency might be compromised to satisfy timing constraints.

In real-time scheduling, tasks are assumed to be independent, and the time spent synchronizing their access to shared data is assumed to be negligible compared with execution time. Knowledge of resource and data requirements of tasks is also assumed to be available in advance.

In replication control methods, on the other hand, the objective is to provide a high degree of concurrency

and thus faster average response time without violating data consistency [Son87]. Two different policies can be employed in order to synchronize concurrent data access of transactions and to ensure identical replica values: *blocking* transactions or *aborting* transactions. However, blocking may cause priority inversion when a high priority transaction is blocked by lower priority transactions. Aborting lower priority transactions, though, wastes the work done by them. Thus, both policies have negative effects on time-critical scheduling.

Conventional replication control algorithms are synchronous, in the sense that they require the atomic updating of some number of copies. This leads to reduced system availability and decreased throughput as the size of the system increases. On the other hand, asynchronous replication control methods that would allow more transactions to meet their deadlines suffer from a basic problem: the system enters an inconsistent state in which replicas of a given data object may not share the same value. Standard correctness criteria for coherency control such as the *1-copy serializability* (1SR) [Ber87] are thus hard to attain with asynchronous consistency control.

A less stringent, general-purpose consistency criterion is necessary. The new criterion should allow more real-time transactions to satisfy their timing constraints by temporarily sacrificing database consistency to some small degree. *Epsilon-serializability* (ESR) is such a correctness criterion, offering the possibility of maintaining mutual consistency of replicated data asynchronously [Pu91]. Inconsistent data may be seen by certain query transactions, but data will eventually converge to a consistent (1SR) state. Additionally, the degree of inconsistency can be controlled within a specified threshold.

The goal of our work is to design a replication control algorithm that allows as many transactions as possible to meet their deadlines and at the same time maintains the consistency of replicated data in the absence of any *a priori* information. Our algorithm is based on a token-based synchronization scheme for replicated data in conventional distributed databases. Real-time scheduling features are developed on top of this platform. Epsilon-serializability is employed as the cor-

rectness criterion that guarantees the robustness of the scheme.

2. Database Model

Before presenting our real-time replication scheme, we first present the model of the underlying distributed database system and transaction processing.

2.1 Distributed System Environment

A distributed system consists of multiple autonomous computer systems (sites) connected via a communication network. Each site maintains a local database system. The smallest unit of data accessible to the user is called *data object*. A data object is an abstraction that does not correspond directly to a real database item. In distributed database systems with replicated data objects, a logical data object is represented by a set of one or more replicated physical data objects. We assume that the database is fully replicated at all sites. *Read* and *Write* are the two fundamental types of logical operations that are implemented by executing the respective physical operation on one or more copies of the physical data object in question. A *token* designates a read-write copy. Each logical data object has a predetermined number of tokens, and each token copy is the latest version of the data object. The site which has a token-copy of a logical data object is called a *token site*, with respect to the logical data object. In order to control the access to data objects, the system uses timestamps. When a write operation is successfully performed and the transaction is committed, a new version is created which replaces the previous version of the token copy.

When a transaction performs a write operation to a physical data object, there are two values that are associated with the data object: the *after-value* (the new version) and the *before-value* (the old version). Because the before-value is available during the transaction processing, it is natural to ask if concurrency can be improved by giving out this value [Bay80]. For example, if the transaction T_1 has been given a permission to write the new value of a data object and the transaction T_2 requests to read the same data object, then it is possible to give T_2 the before-value of the data object, instead of making T_2 wait until T_1 is finished. However, an appropriate control must be exercised in doing so, otherwise the database consistency might be violated. In the example above, assume that T_1 has written a new value for two data objects X and Y, and T_2 has read the before-value of X. T_2 wants to read Y also. If T_2 gets the after-value of Y created by T_1 , there is no serial execution of T_1 and T_2 having the same effect because in reading the before-

value of X, T_2 sees the database in a state before the execution of T_1 , and in reading the after-value of Y, T_2 sees the database in a state after the execution of T_1 .

2.2 Transactions

A *transaction* is a sequence of operations that takes the database from a consistent state to another consistent state. Two types of transactions are allowed in our environment: *query* transactions and *update* transactions. Query transactions consist only of read operations that access data objects and return their values to the user. Update transactions consist of both read and write operations. They execute a sequence of local computations and update the values of all replicas of each associated data object.

Transactions arriving at the system are assumed to be non-periodic. A globally *unique timestamp* is generated for each transaction [Lam78]. Each time a transaction is aborted and resubmitted, a new timestamp value is assigned to it. If a transaction T_1 has a smaller timestamp than another transaction T_2 , we say that T_1 is the older transaction and T_2 is the younger one.

We assume no *a priori* knowledge of which or how many data objects are going to be accessed by each individual transaction. However, we assume that the average length of query and update transactions are known in order to control the level of inconsistency. Transactions that miss their deadlines are immediately aborted.

Read or write operations of the same transaction are executed one by one in a serial fashion. Each read and write carries the timestamp of the transaction that issued it, and each copy carries the timestamp of the transaction that wrote it. A conflict occurs when a transaction issues a request to access a data object for which other transaction has previously issued a request to access, and furthermore at least one of these requests is a write request. There are three kinds of conflicts: read-write (RW), write-read (RW), and write-write (WW) conflicts [Ber87]. In each case, we say that the transaction requesting the new access has caused a conflict.

2.3 Token-Based Conflict Resolution

Let T_1 be the transaction which already issued an access request, and T_2 cause the conflict. For each token copy of X, conflicts are resolved as the following [Son89]:

(1) RW conflict: If T_2 is younger than T_1 , then it waits for the termination of T_1 . If T_2 is older than T_1 , then it reads before-value of X.

(2) WR conflict: If T_2 is younger than T_1 , then its write request is granted with the condition that T_2 cannot

commit before the termination of T_1 . If T_2 is older than T_1 , then T_2 is rejected.

(3) WW conflict: If T_2 is younger than T_1 , then it waits for the termination of T_1 . If T_2 is older than T_1 , then T_2 is rejected.

The coordinator of an update transaction maintains the *before-list* (BL), a list of transactions which read the before-value of any data object in its write set, and the *after-list* (AL), a list of transactions which write the after-value of any data object in its read set. The BL and AL are used during the commitment phase of every update transaction.

When a transaction T_2 reads the before-value of a data object locked by T_1 , the token-site which gives the before-value, conveys the identifier of T_2 to the coordinator of T_1 . Hence, the identifier of T_2 is inserted in the before-list of T_1 , which stores all the transactions that read the before-values of any data object in T_1 's write-set. The transaction manager at the read-only site of T_2 also conveys the identifier of T_1 to the coordinator of T_2 . Actually, the identifier of T_1 is inserted in the after-list of T_2 , which stores all the transactions that write the after-value of any data object in T_2 's read-set.

When a transaction terminates (either commits or aborts), the coordinator of the terminating transaction must inform the coordinator of each transaction in its AL about the termination by sending Termination Messages (TM). On receiving a TM from the coordinator of a transaction in its BL, the coordinator of the active transaction removes the identifier of the terminating transaction (sender of the TM) from the BL. A transaction can commit only when its BL is empty. By this way, we prevent non-serializable execution sequences to occur.

Update transactions have their own *private workspace* where they initially apply their write operations. Update transactions commit by employing a two-phase protocol. In the first phase (*vote-phase*), an update transaction sends an update message to each token-site of every data object in its write-set. The transaction waits until it gets a response from all the token-sites for each data object. If all token-sites vote YES, then the transaction enters the second phase (*commit phase*). It sends the actual value of each data object to be written to the respective token-sites. Update messages to non token-sites can be scheduled after commitment. Therefore, a temporary and limited difference among object replicas is permitted; these replicas are required to converge to the standard ISR consistency as soon as all the update messages arrive and are processed. An update transaction that executes its commit phase can never be aborted, even if it potentially conflicts with another transaction.

Query transactions fall into three different categories as far as the correctness of their response is concerned:

- *Required consistent queries.* Queries are specified as such when they are first submitted by the user, and they are always guaranteed to return consistent data;

- *Consistent queries.* Their final output is correct regardless of any requirement by the user;

- *Possibly inconsistent queries.* In case of such a query, there exists a small possibility that returned values of a replicated data object might reflect an inconsistent state of the database.

Consider a read operation of transaction T_i on a data object X. If the local copy of X has *timestamp* > *timestamp* (T_i) then the local value is returned. Otherwise, an *Actualization Request Message* (ARM) is sent to any available token-site to actualize the read-only copy. At the token-site, an ARM is treated the same as a read request, and the current version of the data object will be returned. However, depending on their categories, query transactions are not always guaranteed to return accurate results.

3. Epsilon-Serializability

Epsilon-serializability (ESR) is a correctness criterion that enables asynchronous maintenance of mutual consistency of replicated data [Pu91]. A transaction with ESR as its correctness criterion is called an *epsilon-transaction* (ET). An ET is a query ET if it consists of only reads. An ET containing at least one write is an update ET. Query ETs may see an inconsistent data state produced by update ETs. The metric to control the level of inconsistency a query may return is called the *overlap*. It is defined as the set of all update ETs that are active and affecting data objects that the query seeks to access. If a query ET's overlap is empty, then the query is serializable.

3.1 Query Overlap Considerations

The overlap of an active query transaction Q can be used as an *upper bound of error* on the degree of inconsistency that Q may accumulate. Given that we are interested in how many update transactions overlap with Q more than which transactions those are, the term overlap, in its further usage, will reflect the cardinality of the set of update transactions that conflict with the query ET Q . More formally, query Q 's overlap is described as follows:

$$\text{Overlap}[Q] = \{ \{ U_i \mid U_i \text{ update trans} \wedge U_i \text{ active during } Q \wedge \text{write-set}(U_i) \cap \text{read-set}(Q) \neq \emptyset \} \}$$

Suppose we have a database of A distinct data objects, and that query transactions read m data objects

on the average, and possibly conflict with update transactions that update n data objects on the average. The exact value for the overlap number can be computed as follows. We compute the maximum allowable overlap of the query Q for a given degree of query inconsistency p . The probability that U_i accesses n objects different from any of the m objects of Q 's read set is:

$$p_i = \frac{A-m}{A} \times \frac{A-m-1}{A-1} \times \dots \times \frac{A-m-n}{A-n}$$

So the probability that U_i has common elements with Q (i.e. Q overlapping with U_i) is: $1 - p_i$.

The probability for a query transaction to overlap with an arbitrarily chosen update transaction is:

$$\begin{aligned} l_i &= 1 - p_i = 1 - \frac{A-m}{A} \times \dots \times \frac{A-m-n}{A-n} \\ &= 1 - \frac{(A-m)! (A-n-1)!}{A! (A-m-n-1)!} \end{aligned}$$

Variable l_i essentially represents the inconsistency probability that a query overlaps with exactly one update transaction ($k = 1$). If k is the maximum overlap, then the equation $\sum_{i=1}^k l_i = p$ must hold. We emphasize that we have k distinct update transactions that potentially conflict with the query.

Since we assume that read/update sets are uniformly distributed within the database, we have $l_i = l \forall i \leq k$, and thus $k \times l = p$. Solving the equation for k after substituting the value for l , the overlap bound k is:

$$k = \left\lceil \frac{p}{1 - \frac{(A-m)! (A-n-1)!}{A! (A-m-n-1)!}} \right\rceil$$

Even though this choice of the overlap bound k is reasonable, it is not unique or critical. The algorithm to be presented in Section 5 will work with other choices of k , or even in its absence.

3.2 E-Transaction Compatibility

Among several replica control methods based on ESR, we have chosen the ordered updates approach [Pu91]. The ordered updates approach allows more con-

currency than ISR in two ways. First, query ETs can be processed in any order because they are allowed to see intermediate, inconsistent results. Second, update ETs may update different replicas of the same object asynchronously, but in the same order. In this way, update ETs produce results equivalent to a serial schedule; these results are therefore consistent.

There are two categories of transaction conflicts that we examine: conflicts between update transactions and conflicts between update and query transactions.

Conflicts between update transactions can be either RW conflicts or WW conflicts. Both types must be strictly resolved. No correctness criteria can be relaxed here, since execution of update transactions must remain 1SR in order for replicas of data objects to remain identical.

Conflicts between update and query transactions are of RW type. Each time a query conflicts with an update, we say that the query overlaps with this update, and the overlap counter is incremented by one. If the counter is still less than a specified upper bound (i.e. the value of k derived above), then both operation requests are processed normally, the conflict is ignored, and no transaction is aborted. Otherwise, RW conflict must be resolved by using the conventional 1SR correctness criteria of the accommodating algorithm.

The performance gains of the above conflict resolution policies are numerous. Update transactions are rarely blocked or aborted in favor of query transactions. They may be delayed on behalf of other update transactions in order to preserve internal database consistency. On the other hand, query transactions are almost never blocked provided that their overlap upper bound is not exceeded. Finally, update transactions attain the flexibility to write replicas in an asynchronous manner.

4. Real-Time Issues

In real-time databases, transactions are characterized by their timing constraints and their data and computation requirements. Timing constraints are expressed through the *release time* and the *deadline*. Computation requirements for transactions are unknown, and no run-time estimate is available for every transaction that enters the system. Neither are data requirements known beforehand, but they are discovered dynamically as the transaction executes. Our goal is to minimize the number of transactions that miss their deadlines [Abb88].

The real-time scheduling part of our scheme has three components: a policy to determine which transactions are eligible for service, a policy for assigning priorities to transactions, and a policy for resolving conflicts between two transactions that want to lock the same data object. None of these policies needs any more informa-

tion about transactions than the deadline and the name of the data object currently being accessed.

All transactions which are currently *not tardy* are eligible for service. Transactions that have already missed their deadlines are immediately aborted. When a transaction is accepted for service at the local site where it was originally submitted, it is assigned a priority according to its deadline. The transaction with the *earliest deadline* has the highest priority. This policy meshes efficiently with the "not tardy" eligibility policy adopted above, so that transactions that have already missed their deadlines are automatically screened out before any priority is assigned to them. *High priority* is the policy that is employed for resolving transaction conflicts. Transactions with the highest priorities are always favored. The favored transaction, i.e. the winner of the conflict, gets the resources that it needs to proceed (e.g., data locks and the processor [Car89]). The loser of the conflict relinquishes control of any resources that are needed by the winner. The loser transaction will either be aborted or blocked depending on the relative age of the two conflicting transactions and the special provisions made by the replication control scheme.

5. Replication Control Scheme

In this section, we present the token-based replication control scheme in detail, along with the embedded ESR correctness criteria and real-time constraints.

5.1 Controlling Inconsistency of Queries

Queries are only involved in RW/WR conflicts. When a query transaction is submitted to the system, the user may quantify it with the restriction "*required to be consistent*." Such a characterization means that all possible future RW/WR conflicts between this query and update transactions will have to be resolved in a strict (ISR) way. In other words, *consistent queries (CQs)* are treated in the same fashion as update transactions. Values returned by CQs are always correct, reflecting the up-to-date state of the respective data objects.

If no consistency constraints are specified explicitly by the user on a submitted query, then the *ESR correctness criterion* is employed to maintain the query's consistency. The overlap upper bound is computed, and an overlap counter is initialized to zero. Each time the query conflicts with an update transaction over the same data object and the counter is less than the overlap upper bound, the conflict is ignored, the counter is incremented, the query reads the value of the data object in question and proceeds to read the next object. When the overlap counter is found to be equal to the upper bound,

current and all subsequent conflicts must be resolved in a strict manner, so that no more inconsistency will be accumulated on the query.

When a query transaction eventually commits, the user is able to determine the degree of correctness of the data values returned. If the query was qualified as a CQ, then the user can be confident that the values returned are coherent. For regular query transactions, the private overlap counter is checked. If the counter is still zero, this means that no conflict has occurred throughout the entire execution of the query and the results must again be perfectly accurate. Such a query falls into the CQ class. An overlap counter greater than zero indicates that a certain number of conflicts with update transactions remained unresolved; the query had seen some possibly inconsistent intermediate states, and might yield some inaccurate data. This last type of query falls into the "*possibly inconsistent*" queries class. The probability that such a query outputs inconsistent data is bounded by the probability p which was used in the calculation of the overlap limit k . Data values can then be referenced with $[(1 - p) \times 100]\%$ confidence in their correctness.

Since arbitrary queries may produce results beyond allowed inconsistency even within its overlap limit, it is important to restrict ET queries to have certain properties that permit tight inconsistency bounds. A first attempt in this approach is proposed in [Ram91]. It is beyond the scope of this paper to deal with such strategies. In the remainder of the paper, we assume that inconsistency bounds can be enforced by the system if necessary.

For each query transaction T, we can also provide the number of possibly incorrect values read by T by checking the overlap counter of T and the number of data objects read by T. Let m_{ex} be the exact number of data objects read by T and k_{ex} be the value of the overlap counter of T after T is terminated. The number of possibly incorrect data values read by T is: $m_{ex} \times p_{ex}$, where

$$p_{ex} = k_{ex} \times \left(1 - \frac{(A - m_{ex})! (A - n - 1)!}{A! (A - m_{ex} - n - 1)!} \right)$$

is the exact probability that T is inconsistent.

5.2 Conflict Resolution

Mechanisms for conflict resolution between update transactions comprise the core of our scheme. Query transactions need not be considered separately because queries that are forced to resolve their RW conflicts with update transactions can be treated as update transactions. Therefore, in the rest of the section, we use the general term *transaction* when we refer either to a normal update

Priority Age	T_2 higher priority	T_2 lower priority
T_2 younger	<ul style="list-style-type: none"> • T_1 reads before value. • T_2 writes. • T_2 allows T_1 to commit before it commits. • If T_2 requests to commit T_1 is (cond) aborted, T_2 commits. 	<ul style="list-style-type: none"> • T_1 reads before value. • T_2 writes. • T_2 waits for T_1 to commit before it commits.
T_2 older	<ul style="list-style-type: none"> • T_1 is aborted (cond). • T_2 writes. 	<ul style="list-style-type: none"> • T_1 reads. • T_2 is aborted.

Table 3: W-R Conflicts

lower priority. In the case that T_2 has a higher priority, aborting T_2 would violate the real-time constraints. Therefore, we let T_2 proceed and write a new value for X while T_1 is aborted, since it has seen a value of X that has already become obsolete.

(3) W - W Conflict.

Transaction T_2 requests to write data object X for which transaction T_1 has already issued a write request. Table 4 shows the various resolution policies.

If T_2 is younger than T_1 , then T_2 should wait for the termination of T_1 before it writes a new value for data object X . Such conflict resolution favors T_1 and is compatible with the situation where T_2 has lower priority than T_1 . However, when T_2 has a higher priority, it is not required to wait for the lower priority transaction T_1 . Hence, T_2 will proceed, and T_1 will be conditionally aborted in order for the database to remain internally consistent.

If T_2 is older than T_1 , then T_2 should be aborted. Note that we are interested only in the most recent value of X , i.e. the value written by the younger T_1 transaction. In the case that T_2 has a lower priority, the above resolu-

tion is acceptable, since the higher priority T_1 is favored to proceed. On the contrary, when T_2 has the higher priority, T_2 must be allowed to write its own new value of X , and T_1 must be conditionally aborted for the database to remain consistent with respect to the data object X .

5.3 Commitment

The coordinator of a transaction decides to commit when the following conditions are satisfied:

- The transaction must not have missed its deadline;
- Each data-object in the read-set of the transaction is read;
 - All the token-sites of each data object in the write-set of the transaction have precommitted (this only applies to update transactions);
 - There is no active transaction that has seen before-value of any data object in the transaction's write-set. In other words, the *before-list* of the transaction must be empty (this only applies to update transactions).

Priority Age	T_2 higher priority	T_2 lower priority
T_2 younger	<ul style="list-style-type: none"> • T_1 is aborted (cond). • T_2 writes. 	<ul style="list-style-type: none"> • T_1 writes. • T_2 waits for T_1 to terminate
T_2 older	<ul style="list-style-type: none"> • T_1 is aborted (cond). • T_2 writes. 	<ul style="list-style-type: none"> • T_1 writes. • T_2 is rejected.

Table 4: W-W Conflicts

6. Concluding Remarks

In this paper we have presented a synchronization scheme for real-time distributed database systems. The algorithm is based on a *token-based approach*, in which two additional components are built. The first is a set of *real-time constraints* that each transaction has to meet. A separate priority scheme is employed to reflect the demand of a transaction to finish before its deadline. The second component is the *ESR correctness criterion* with which query transactions have to comply. Instead of applying ISR to all transactions, ISR is applied only to updates, and queries are left free to be interleaved with updates in a more flexible way.

By relaxing the consistency criteria for query transactions, queries and updates hardly ever have to abort or block each other due to conflicts between them. As an immediate consequence of this, more transactions may terminate successfully before their deadlines expire. Additionally, the second mechanism further improves performance; updating the different replicas of the same data object is done asynchronously, but in the same order. Thus, logical write operations become disjoint from the corresponding physical write operations, and update transactions are free to proceed to the next step of their execution or even to commit. Internal database consistency is preserved strictly. Data returned by certain queries are allowed to exhibit limited inconsistency, under user control.

Another advantage of our scheme lies in the fact that there is very little information the user has to provide to achieve efficient system operation. No *a priori* knowledge of the kind or the number of the data objects that are included in the read-set or the write-set of a transaction is needed. The only information required is the kind of each submitted transaction (query or update), and the expected average number of objects accessed by each transaction. Moreover, no execution time estimate is required for each submitted transaction. It would be extremely difficult to compute a run-time estimate, especially in the distributed environments for which our scheme is designed.

There is a price to pay for relaxing correctness criteria and meeting more deadlines. Although the user can control the maximum permissible inconsistency of queries, one cannot know exactly which one transaction out of the set of all possibly inconsistent queries will return incorrect data, unless a tight inconsistency bound is provided. Note that an overlap counter greater than zero does not necessarily mean that the respective query transaction is inconsistent. It simply indicates that certain RW/WR conflicts were passed unresolved, and inconsistency might be present among the data values returned.

REFERENCES

- [Abb88] R. Abbott, and H. Garcia-Molina, "Scheduling Real-time Transactions: a Performance Evaluation," Proceedings of the 14th VLDB Conference, Los Angeles, California 1988.
- [Bay80] R. Bayer, H. Heller, and A. Reiser, "Parallelism and Recovery in Database Systems," ACM Trans. Database Syst. 5, 2, June 1980.
- [Ber87] P.A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems," Addison-Wesley Publishing Co., 1987.
- [Car89] M.J. Carey, R. Jauhari, and M. Livny, "Priority in DBMS Resource Scheduling," Proceedings of the 15th VLDB Conference, Amsterdam, 1989.
- [Lam78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Comm. ACM, vol. 21, no. 7, pp. 558 - 565, July 1978.
- [Pu91] C. Pu, and A. Leff, "Replica Control in Distributed Systems: An Asynchronous Approach," ACM SIGMOD Conference, May 1991.
- [Ram91] K. Ramamritham and C. Pu, "A Formal Characterization of Epsilon Serializability," Tech. Rep. 91-91, Dept. of Computer Science, Univ. of Massachusetts, Dec. 1991.
- [Son87] S.H. Son, "Synchronization of Replicated Data in Distributed Systems," Information Systems, vol. 12, no. 2, pp. 191 - 202, 1987.
- [Son89] S.H. Son, "A Resilient Replication Method in Distributed Database Systems," Proceedings of IEEE INFOCOM '89, Ottawa, Canada, April 1989.
- [Son90] S.H. Son, "Real-Time Database Systems: A New Challenge," Data Engineering, vol. 13, no. 4, Special Issue on Future Directions on Database Research, December 1990.

Scheduling real-time transactions using priority

S H Son

Compared with traditional databases, real-time database systems have a distinct feature: they must satisfy timing constraints associated with transactions. This requires that transactions in real-time database systems should be scheduled to consider both data consistency and timing constraints. The paper addresses the issues associated with transaction scheduling and concurrency control in real-time database systems. As a specific example of real-time transaction scheduling, a priority-based scheduling algorithm is discussed, together with a performance study using a database prototyping environment.

real-time systems, databases, prototyping, synchronization, transaction, priority

As computers are becoming an essential part of real-time systems, real-time computing is emerging as an important discipline in computer science and engineering¹. The growing importance of real-time computing in a large number of applications, such as aerospace and defence systems, industrial automation and robotics, and nuclear power plants, has resulted in increased research in this area. Researchers working in the real-time systems area have found that traditional data models are not adequate for real-time systems. In recent workshops, the need for more active research in database systems that satisfy timing constraints in collecting, updating, and retrieving shared data has been pointed out^{2,3}. Most database systems are not designed for real-time applications and lack the features required to support real-time transactions. Few conventional database systems allow users to specify timing constraints or ensure that the system meets those set by the user. Interest in this new application domain is also growing in the database community. Recently, a number of research results has appeared in the literature⁴⁻¹².

Real-time database systems have (at least some) transactions with explicit timing constraints. Typically, a timing constraint is expressed in the form of a deadline, a certain time in the future by which a transaction needs to be completed. A deadline is said to be 'hard' if it cannot be missed or else the result is useless. If a deadline can be missed, it is a 'soft' deadline. With soft deadlines, the usefulness of a result may decrease after the deadline is

missed. In real-time database systems, the correctness of transaction processing depends not only on maintaining consistency constraints and producing correct results, but also on the time at which a transaction is completed. Transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query on the tracking data for a missile must be processed within given deadlines, satisfying not only database consistency constraints, but also timing constraints.

Conventional database systems are typically not used in real-time applications due to the inadequacies of poor performance and lack of predictability. They are designed to provide good average performance, while possibly yielding unacceptable worst-case response times. In addition, conventional database systems do not schedule their transactions to meet response requirements and they commonly lock data tables to assure only the consistency of the database. Locks and time-driven scheduling are basically incompatible, resulting in response requirement failures when low-priority transactions block higher-priority transactions. New techniques are required to manage the consistency of real-time databases, and they should be compatible with time-driven scheduling and meet both the required temporal constraints and data consistency.

To address the inadequacies of current database systems, the transaction scheduler needs to be able to take advantage of the semantic and timing information associated with data objects and transactions. A model of real-time transactions needs to be developed that characterizes distinctive features of real-time databases and can contribute to the improved responsiveness of the system. The semantic information of the transactions investigated in the modelling study can be used to develop efficient transaction schedulers^{12,13}.

The satisfying of timing constraints while preserving data consistency requires the concurrency control protocol to accommodate timeliness of transactions as well as data consistency requirements. In real-time database systems, timeliness of a transaction is usually combined with its criticality to take the form of the priority of the transaction. Therefore, proper management of priorities and conflict resolution in real-time transaction scheduling are essential for predictability and responsiveness of real-time database systems.

This paper addresses the issues associated with tran-

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

saction scheduling and concurrency control in real-time database systems. First, a priority-based scheduling approach for real-time database systems is introduced. As a specific example of real-time transaction scheduling, the priority-ceiling protocol is discussed, together with a performance study using a database prototyping environment. Other issues in scheduling real-time transactions are also discussed.

PRIORITY-BASED SCHEDULING

Real-time databases are often used in applications such as tracking. Tasks in such applications consist of both computing (signal processing) and database accessing (transactions). A task can have multiple transactions, which consist of a sequence of read and write operations that operate on the database. Each transaction will follow the two-phase locking protocol¹⁴, 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 a real-time database system, scheduling protocols must not only maintain the consistency constraints of the database, but also satisfy the timing requirements of the transactions that access the database. To satisfy both the consistency and real-time constraints, it is necessary to integrate synchronization protocols with 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 inversion¹⁵. Priority inversion is said to occur when a higher-priority task is forced to wait for the execution of a lower-priority task for an indefinite period. When two transactions attempt to access the same data object, the access must be serialized to maintain consistency. If the transaction of the higher-priority task gains access first, then the proper priority order is maintained; however, if the lower-priority transaction gains access first and then the higher-priority transaction requests access to the data object, this higher-priority task will be blocked until the lower-priority transaction completes its access to the data object. Priority inversion is inevitable in transaction systems. To achieve a high degree of schedulability in real-time applications, however, priority inversion must be minimized. This is illustrated by the following example.

Example 1

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, as O_i is already locked. It would be expected 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 , which 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 short transactions, because it creates unnecessary blocking. For instance, once a long low-priority transaction starts execution, a high-priority transaction that does not require 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 proposed¹⁵. The basic idea of priority inheritance is that when a transaction T of a task blocks higher-priority tasks, 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, by solving the unbounded priority inversion problem. In the context of preemptive scheduling, a higher-priority transaction T can preempt the execution of lower-priority transactions unless T is blocked by the locking protocol. The priority inheritance rule states that when a transaction blocks the execution of higher-priority transactions, it executes at the highest priority of all the transactions blocked by its locks. For example, suppose transaction T_1 is blocked by T_3 . Then the priority-inheritance protocol ensures that T_3 will execute at T_1 's priority until it releases the lock on the data object T_1 is blocked for.

The priority inheritance alone, however, 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 access sequentially 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 locked by the lower-priority transactions T_2 and T_3 , respectively. 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 transactions⁹. 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: either executing or being preempted in the middle of its execution. The idea of total priority ordering is that the real-time locking protocol ensures that each active transaction is executed at some priority level, taking priority inheritance and read/write semantics into consideration.

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 the 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 equal to the absolute priority ceiling. When it is read-locked, the rw-priority ceiling of this data object is equal to the write-priority ceiling.

The reason for specifying the rw-priority ceiling differently depending on the lock type set on the data object is lock compatibility. When a data object is write-locked, it cannot be read or written by another transaction. To ensure this, the rw-priority ceiling of the data object is set to its absolute-priority ceiling. As the absolute-priority ceiling of a data object is equal to the priority of the highest-priority transaction that may either read or write it, it prevents another task from reading or writing until the lock is released. Similarly, if it is read-locked, it cannot be written by another transaction. To ensure this, when a data object is read-locked by a transaction, its rw-priority ceiling is set to its write-priority ceiling. As the write-priority ceiling equals the priority of the highest-priority transaction that may write it, it prevents another transaction from writing the data object. According to the rw-priority-ceiling rule, the systems can guarantee that a data object can be locked by a transaction T only if T 's priority is higher than the priority ceiling of all data objects currently locked by transactions other than T in an incompatible mode.

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

When a transaction attempts to lock a data object, the transaction's priority is compared with the highest rw-priority ceiling of all data objects currently locked by other transactions. If the priority of the transaction is not higher than the rw-priority ceiling, the access request will be denied, and the transaction will be blocked. In this case, the transaction is said to be blocked by the transaction that holds the lock on the data object of the highest rw-priority ceiling. Otherwise, it is granted the lock. In the denied case, the priority inheritance is performed to overcome the problem of uncontrolled priority inversion. For example, if transaction T blocks higher transactions, T inherits P_H , the highest priority of the transactions blocked by T . Priority inheritance is transitive. The next example shows how transactions are scheduled under the priority-ceiling protocol.

Example 2

Consider the same situation as in example 1. 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 . As T_3 blocks T_2 , its priority is promoted to that of T_2 . When T_1 requests O_i , it will be blocked, and the priority of T_3 will be promoted to that of T_1 . When T_3 unblocks O_i , the priority of T_3 resumes its original priority. At that point, T_1 will preempt T_3 and will lock 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.

Using the priority-ceiling protocol, mutual deadlock of transactions cannot occur and each transaction can be blocked by at most one lower-priority transaction until it completes or suspends itself. A high-priority transaction can be blocked by a low-priority transaction in one of three cases.

- The first case occurs when a high-priority transaction attempts to lock a data object already locked by a low-priority transaction.
- The second case occurs when a medium-priority transaction is blocked by a low-priority transaction that has promoted its priority by inheriting that of a high-priority transaction. This type of blocking is necessary to avoid a situation in which a high-priority transaction is indirectly blocked by a medium-priority transaction.
- The third type of blocking is called ceiling blocking, which occurs when a transaction cannot start the execution because its priority is not higher than the priority ceiling of the data objects locked by other active transactions. Ceiling blocking is necessary to avoid deadlock and chained blocking.

The total priority ordering of active transactions leads to some interesting behaviour. As shown in example 2, 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.

PERFORMANCE EVALUATION

The issues associated with the idea of total ordering in priority-based scheduling protocols have been investigated using a database prototyping environment¹⁶. 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. The results indicate that the ceiling protocol offers performance improvement over the two-phase locking protocol (2PL).

In the author's experiments, transactions are generated and put into the start-up queue. When a transaction

is started, it leaves the start-up queue and enters the ready queue. Transactions in the ready queue are ordered from the highest priority to the lowest priority. The transaction with the highest priority is always selected to run. The current running transaction sends requests to the concurrency controller. The transaction may be blocked and placed in the block queue. It may also be aborted and restarted. In such a case, it is first delayed for a certain amount of time and then put in the ready queue again. When a transaction in the block queue is unblocked, it leaves the block queue and is placed in the ready queue. Whenever a transaction enters the ready queue and its priority is higher than the current running transaction, it preempts the current running transaction.

When a transaction enters the start-up queue, it has the arrival time, the deadline, the priority, the read set, and the write set associated with it. The transaction inter-arrival time is a random variable with exponential distribution. The data objects in the read set and the write set are uniformly distributed across the entire database. A transaction consists of a sequence of read and write operations. A read operation involves a concurrency-control request to get access permission, followed by a disc input/output (I/O) to read the data object, followed by a period of central processing unit (CPU) use for processing the data object. Write operations are handled similarly, except for their disc I/O. A transaction can be discarded at any time if its deadline is missed. Therefore, the model employs a hard deadline policy.

Transaction size (the number of data objects a transaction needs to access) has been used 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. The arrival rate was chosen so that protocols are tested in a heavily loaded rather than lightly loaded system. For the design of real-time systems, high-load situations must be considered. Even though they may not arise frequently, it is desirable 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 important⁴. The following summarizes the findings briefly to illustrate the performance of the algorithms.

In Figure 1, the throughput of the ceiling protocol (C), 2PL with priority mode (P), and 2PL without priority mode (L), is shown for transactions of different sizes. The two-phase locking protocol with priority mode is also called the high-priority protocol⁴. In that protocol, all data conflicts are resolved in favour of the transaction with higher priority. When a transaction requests a lock on a data object held by other transactions in an incompatible mode, if the requester's priority is higher than that of all the lock holders, the holders are restarted and the requester is granted the lock; if the requester's priority is lower, it waits for the lock holders to release the lock.

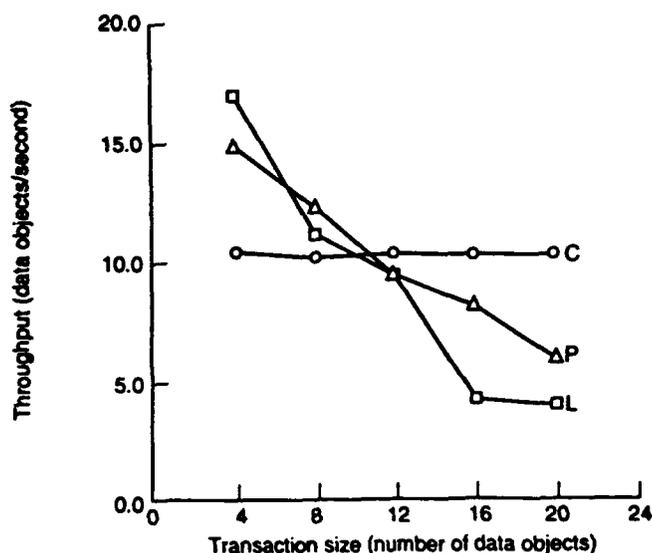


Figure 1. Transaction throughput

When the transaction size is small, there is little locking conflict and the problem, such as deadlock and priority inversion, has little effect on the overall performance of a locking protocol. On the other hand, when transaction size becomes large, the probability of locking conflict rises rapidly. Hence it would be expected that the performance of protocols will be dominated by their abilities to handle locking conflicts when the transaction size is large.

As illustrated in Figure 1, the performance of the 2PL with or without priority assignments degrades very fast when transaction size increases. On the other hand, the ceiling protocol handles locking conflicts well. The protocol is free from deadlocks and exhibits the block-at-most-once property. Hence it performs much better than 2PL when transaction size is large. This is because in the priority-ceiling protocol the conflict rate is determined by ceiling blocking rather than by direct blocking, and the frequency of ceiling blocking is not sensitive to the transaction size.

Another important performance statistic is the percentage of deadlines missed by transactions, as the synchronization protocol in real-time database systems must satisfy the timing constraints of individual transactions. In the 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 2, the percentage of deadlines missed by transactions increases sharply for the 2PL as the transaction size increases due to its inability to deal with deadlock and to give preference to transactions with shorter deadlines. Two-phase lock with priority assignment performs somewhat better, because the timing constraints of transactions are considered, although the deadlock and priority-inversion problems still handicap its performance. The ceiling protocol has the best relative performance because it addresses both the deadlock and priority-inversion problems.

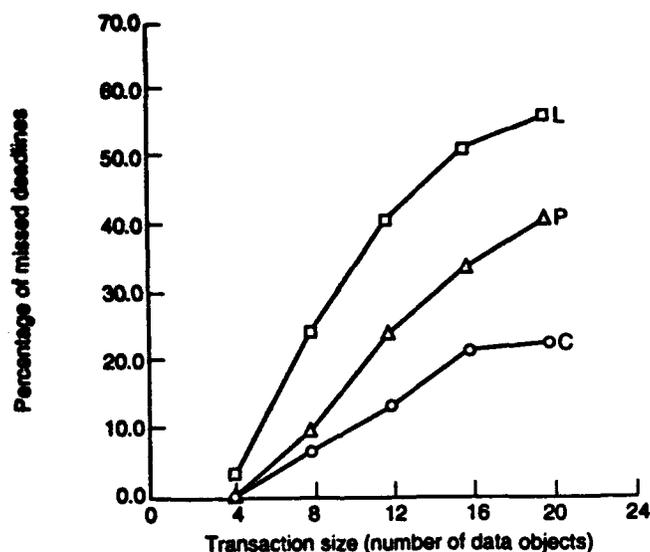


Figure 2. Percentage of deadline-missing transactions

ISSUES IN SCHEDULING REAL-TIME TRANSACTIONS

Deadlines are timing constraints associated with transactions. In real-time database systems, scheduling decisions are often directly related to whether the transactions meet or miss their deadlines. Scheduling decisions must be made when the scheduler has to select from among a collection of transactions that are ready to be started or when a choice has to be made between two or more transactions that are competing for the same resources (i.e., data objects). A decision to abort a transaction for a later restart may result in the transaction missing its deadline.

In addition to deadlines, other kinds of timing constraints are associated with data as well as transactions in real-time database systems. For example, each sensor input could be indexed by the time at which the sample was taken. Furthermore, once entered into the database, data may get old or become out of date if they are not updated within a certain period. To quantify this notion of 'age', each datum is associated with a valid interval. Data out of the valid interval do not represent the current state. The time associated with the data is the time at which the value is currently believed to have been true. The valid interval indicates the time interval after the most recent updating of a data object during which a transaction may access a data object with 100% degree of accuracy. What occurs when a transaction attempts to access a data object outside of its valid interval is dependent on 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 data values. In general, each data object is 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 it is given 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 transactions. For example, if the temporal consistency requirement is 15, it indicates that data objects accessed by the transaction cannot be older than 15 time units relative to the start time of the transaction. This temporal consistency requirement can be specified as either hard or soft, just like deadlines. If it is hard, an attempt to read an invalid data object (out of its valid interval) will cause the transaction to abort.

While a deadline can be thought of as providing a time interval as a constraint in the future, the 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 must be able to provide an answer using available (may be not up-to-date) information. The answer may change as valid intervals change with time. In a distributed database system, sensor readings may not be applied to the database at the same time and may not be reflected consistently at the console due to the different delay 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. It should distinguish adequately between 'information not available at time t ' and 'information out of valid interval at time t '.

Real-time systems require a scheduler that should incorporate timing constraints associated with transactions and data objects in its scheduling decisions. The goals of such a scheduler in real-time database systems are:

- to minimize the number of transactions that miss their deadlines
- to ensure meeting the timing requirements of highly critical transactions
- to maximize the overall transaction accuracy within the system

A simplistic approach would be solely to consider the minimization of transaction loss due to missing deadlines. This goal by itself, however, is not sufficient as the transactions due to their temporal requirements may have different degrees of criticalness associated with them. A scheduler may be able to maximize the overall number of transactions that meet their deadlines by successfully scheduling the less critical transactions and causing the highly critical transactions to miss their deadlines. The failure of these highly critical transactions may be too costly in terms of endangering the safety of the entire system. Therefore, it would be desirable for the scheduler to make an effort to ensure that the deadlines of the highly critical transactions are met. In addition, as transactions may require different degrees of accuracy based on their temporal consistency requirements and the validity intervals of data objects, the scheduler must consider the overall transaction accuracy within the system before making a decision for or against a transaction.

An intuitive approach to achieve only the first goal

would be to assign the highest priority in the system to the transaction with the smallest deadline. As this transaction is at the highest risk of missing its deadline, it would be favoured in the scheduling decision. This approach is not acceptable, however, because it fails to consider other important factors. For example, it is possible that the transaction is so close to its deadline that it is almost certain that it would miss its deadline anyway. Making a decision in favour of this transaction may lead to the competing transaction missing its deadline, resulting in a poor performance. Therefore, the scheduler must consider the feasibility of meeting the deadline of the transaction in making a decision.

It is proposed that the goals mentioned above may be achieved by having the scheduler consider the deadlines, the criticalness, and the temporal consistency levels that are associated with transactions. A set of scheduling algorithms that consider each one of these elements in the scheduling decisions has been developed. It has been shown that using these algorithms could reduce the number of deadline-missing transactions and meet the temporal consistency requirements¹². For real-time transactions, it is necessary to define an appropriate notion of correctness, and investigate new techniques to guarantee the desired level of correctness while increasing the performance of the system by using the semantic knowledge of transactions and a temporal data model. A multiversion data object is one approach for exploiting the semantic information of real-time transactions and temporal data models. In a system with multiple versions of data, each write operation on a data object produces a new version instead of overwriting the old version. Hence, for each read operation, the system selects an appropriate version to read, enjoying the flexibility of controlling the order of read and write operations. One of the issues that needs further study is methods to specify appropriate correctness requirements of real-time transactions by their timing constraints and the data objects they need to access.

CONCLUSIONS

In real-time database systems, transactions must be scheduled to meet their timing constraints. In addition, the system should support a predictable behaviour such that the possibility of missing deadlines of critical tasks could be informed ahead of time, before their deadlines expire. The priority-ceiling protocol is one approach to achieve a high degree of schedulability and system predictability. It has been discussed that this protocol might be appropriate for real-time transaction scheduling as it is stable over the wide range of transaction sizes and, compared with the two-phase locking protocol, it reduces the number of deadline-missing transactions.

There are many technical issues associated with real-time transaction scheduling that need further investigation. In the priority-ceiling protocol and many other database scheduling algorithms, preemption in locking 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 carefully, and it should not necessarily be based only on relative deadlines¹⁷. This is so as preemption implies not only that the work done by the preempted transaction must be undone, but also that later on, if restarted, it must redo the work. The resultant delay and the wasted execution may cause one or both of these transactions, as well as other transactions, to miss deadlines.

Even though data objects out of their valid interval do not represent the current state, they might be used for approximation. Methods to specify the temporal-consistency requirement of transactions, and to use valid intervals of data objects in determining the degree of consistency of transactions that access them, are relatively unexplored and are an important problem. Several approaches to designing scheduling algorithms for real-time transactions have been proposed^{4,5,8,12}, but their performance in distributed environments has not been studied. The author is currently working on implementing scheduling algorithms for distributed real-time transactions, using his prototyping environment.

ACKNOWLEDGEMENT

This work was supported in part by ONR under contract N00014-91-J-1102, by DOE, and by NOSC.

REFERENCES

- 1 'Special issue on real-time systems' *Computer* Vol 24 No 5 (May 1991)
- 2 *Proc. Eighth IEEE Workshop on Real-Time Operating Systems and Software* Atlanta, GA, USA (May 1991)
- 3 *ONR Annual Workshop on Foundations of Real-Time Computing* Washington, DC, USA (October 1990)
- 4 Abbott, R and Garcia-Molina, H 'Scheduling real-time transactions: a performance study' in *Proc. 14th VLDB Conf.* (September 1988) pp 1-12
- 5 Abbott, R and Garcia-Molina, H 'Scheduling real-time transactions with disk resident data' in *Proc. 15th VLDB Conf.* (August 1989)
- 6 Buchmann, A *et al.* 'Time-critical database scheduling: a framework for integrating real-time scheduling and concurrency control' in *Proc. Fifth Data Engineering Conf.* (February 1989) pp 470-480
- 7 Korth, H 'Triggered real-time databases with consistency constraints' in *Proc. 16th VLDB Conf.* Brisbane, Australia (August 1990)
- 8 Lin, Y and Son, S H 'Concurrency control in real-time databases by dynamic adjustment of serialization order' in *Proc. 11th IEEE Real-Time Systems Symp.* Orlando, FL, USA (December 1990)
- 9 Sha, L, Rajkumar, R and Lehoczky, J 'Concurrency control for distributed real-time databases' *ACM SIGMOD Record* Vol 17 No 1 (March 1988) pp 82-98
- 10 Sha, L, Rajkumar, R, Son, S H and Chang, C 'A real-time locking protocol' *IEEE Trans. Comput.* Vol 40 No 7 (July 1991) pp 793-800
- 11 Son, S H (guest ed) 'Special issue on real-time database systems' *ACM SIGMOD Record* Vol 17 No 1 (March 1988)
- 12 Son, S H, Wagle, P and Park, S 'Real-time database scheduling: design, implementation, and performance evaluation' in *Proc. Int. Symp. Database Systems for Advanced Applications (DASFAA '91)* Tokyo, Japan (April 1991) pp 146-155

- 13 **Son, S H and Lee, J** 'Scheduling real-time transactions in distributed database systems' in *Proc. 7th IEEE Workshop on Real-Time Operating Systems and Software* Charlottesville, VA, USA (May 1990) pp 39-43
- 14 **Bernstein, P, Hadzilacos, V and Goodman, N** *Concurrency control and recovery in database systems* Addison-Wesley (1987)
- 15 **Sha, L, Rajkumar, R and Lehoczky, J** 'Priority inheritance protocol: an approach to real-time synchronization' *Technical report* Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, USA (1987)
- 16 **Son, S H** 'An environment for prototyping real-time distributed databases' in *Proc. Int. Conf. Systems Integration* Morristown, NJ, USA (April 1990) pp 358-367
- 17 **Stankovic, J** 'Misconceptions about real-time computing' *Computer* Vol 21 No 10 (October 1988) pp 10-19

Uneven text on CASE for design and practice

CASE: computer-aided software engineering

T G Lewis

Van Nostrand Reinhold (1991) 593pp
£33.50 ISBN 0-442-00361-7

This book includes much information about some rather unusual computer-aided software engineering (CASE) tools, embedded in what would appear to be Lewis' university course in software engineering generally.

First, the coverage of CASE tools particularly: all CASE tools discussed run on and, for the most part, only on Apple Macintoshes. Some are commercial products; some are local products of Oregon State University at Corvallis, USA, the results of student projects and apparently available from the author. A few tools are uncredited, but obviously exist because they, like most others in the text, are discussed to the accompaniment of endless illustrations of their many menus and output screens.

In several cases this merely underlines the difficulty of using necessarily linear text descriptions of highly nonlinear concepts. In all cases the amount of detail devoted to 'which mouse button to push' seems out of place in what is not a user manual but a textbook.

Second, the coverage of software engineering in general: the best I can say is 'uneven', or perhaps 'idiosyncratic'; important issues are left out or glossed over, others are laboured to death. In the 'left-out' category I would put at the top of the list careful discussion of the difference between specification and design models. There is no discussion of 'implementation-free' specification models, of the concept of domain modelling, or of how to move from a formal problem description to a design.

The rather cursory discussion of dataflow modelling merely reinforces this lack of distinction. For instance, what is meant to be a specification dataflow diagram (for a problem that is about cost estimation) includes stores with labels such as 'RAM', 'file', and 'printer'.

Even more fundamentally, Lewis seems to feel that a method without a CASE tool is a fish without a bicycle — that it will not get very far. I was surprised. Many times I have clarified my thinking about a complex subject matter by drawing a simple data model on the back of a fish. I've used dataflow diagrams to help discuss problems with fish. I make sense of communications

protocols by turning them into state models. I have borrowed 'how to start' heuristics from Yourdon, Ward, or Shlaer and Mellor when I am beginning a model of a problem. Methods, and their component notations, are tools for thought. Many companies have bought CASE tools, but not taught their employees how to think with the ideas behind them. Result: disaster. For years, electronics engineers successfully used finite-state machine notations to model complex digital problems without the help of computer-aided engineering. They may have occasionally made mistakes, but their digital design work could not have been done without the discipline of paper-and-pencil work with decision tables and state models. 'Computer-aided' is not a synonym for 'useful'.

Other highly contentious and I think wrong claims are made. For instance, 'There are no guide-lines for combining methods when appropriate'. Or 'None of the methods guarantee incremental correctness of design' ('guarantee', maybe not, but 'help verify' or 'assist in', certainly).

There is some confusion between methods, notations advocated by them, and tools used to expedite the use of either. A naive student might assume for instance that data dictionaries and the use of a data composition notation is a feature of Anatoool, as opposed to a realisation of a concept common to many methods.

The coverage of many major methods and notations is half-hearted. Any book that includes dataflow diagrams with no arrows on any of the data flows is probably not taking the concept behind the diagrams very seriously. There is virtually no coverage of data-modelling/entity-relationship-diagram notations and the methods that use them. The few pages devoted to them use the eccentric term 'entity category relation', which (notation? method?) is introduced without discussion or definition. The reader is given the impression that such notations and the models built with them are only, or mostly, to do with the physical design of databases.

'Object oriented' is interpreted as encapsulation, and as a design issue. There is no coverage or mention of object-oriented or domain analysis. Inheritance is only mentioned once or twice, and then not helpfully; 'subtype/supertype' appears to be equated with

'instance/class'.

Real-time issues are relegated to one two-paragraph programmatic sidebar. There is little coverage of methods or notations used for real-time analysis and design. State models are not discussed or taught, but only briefly mentioned, and then only as a notation for communicating with a user-interface design tool. Jackson Structured Programming (JSP) is discussed, but not Jackson Structured Design (JSD). Just to keep things interesting, JSP is called JSD in the text.

Although the preface claims a readership of 'practitioners who manage, design, code, test and market ...' software, most of them will probably fall asleep over the chapters on mathematical verification and software metrics. The former is fuzzily written and too shallow to teach a beginner anything useful. The latter leads the eager practitioner — as software metrics often seem to — nowhere much. Nor do the pages on the mathematics of reliability statistics and cost estimates. The pages 'adapted from the IEEE Standard for Configuration Management Plan' teach nothing and are likely to induce terminal coma.

Lewis' heart would seem to be in design and implementation; his coverage of lower-CASE tools and the concepts behind them is excellent. He makes good use of quantitative data (I had not before run into Card's wonderful result suggesting that small modules are more expensive to maintain!). Testing, programming style and complexity, and coding standards are covered well.

The text occasionally gets bogged down in programming details of no general consequence. There are pages devoted to Macintosh system programming, down even to hex listings of icon and mask resources.

Coverage of project management issues is deeper than what usually gets into an introductory software engineering text, and much more than one would expect from a book purportedly about CASE. My main protest would be the use made of Boehm's spiral model. Lewis leaves out all mention of risk, and uses the spiral model as a mechanism for discussing rapid prototyping using very-high-level code generators. Boehm's risk-driven model is easy to understand, and I believe really should be discussed whenever the concept of prototyping comes up.

The book could have benefited from better production. Graphics are derived

Hybrid Protocols Using Dynamic Adjustment of Serialization Order for Real-Time Concurrency Control

SANG H. SON, JUHNYOUNG LEE AND YI LIN

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

1. Introduction

A *real-time database system (RTDBS)* differs from a conventional database system because in addition to the consistency constraints of the database, timing constraints of individual transaction need to be satisfied. In order to provide a timely response for queries and updates while maintaining the consistency of data, *real-time concurrency control* should involve efficient integration of ideas from both database concurrency control and real-time scheduling. Various real-time concurrency control protocols have been proposed which employ either a pessimistic or an optimistic approach to concurrency control.

In this paper, we present two hybrid real-time concurrency control protocols which combine pessimistic and optimistic approaches to concurrency control in order to control blocking and aborting in a more effective manner. One protocol is a combination of optimistic concurrency control and locking, and the other is a combination of optimistic concurrency control and timestamp ordering.

2. Integrated Real-Time Locking Protocol

Concurrency control protocols induce a serialization order among conflicting transactions. For a concurrency control protocol to accommodate timing constraints of transactions, the serialization order it produces should reflect the priority of transactions. However, this is often hindered by the past execution history of transactions. A higher priority transaction may have no way to precede a lower priority transaction in the serialization order due to previous conflicts. For example, let T_H and T_L be two transactions with T_H having a higher priority. If T_L writes a data object x before T_H reads it, then the serialization order between T_H and T_L is determined as $T_L \rightarrow T_H$. T_H can never precede T_L in the serialization order as long as both reside in the execution history. Most of the current (real-time) concurrency control protocols resolve this conflict either by blocking T_H until T_L releases the writelock or by aborting T_L in favor of the higher priority transaction T_H . Blocking of a higher priority transaction due to a lower priority transaction is contrary to the requirement of real-time scheduling. Aborting is also not desirable because it degrades the system performance and may lead to violations of timing constraints. Furthermore, some aborts can be wasteful

when the transaction which caused the abort is aborted due to another conflict. The objective of our first protocol is to avoid such unnecessary blocking and aborting.

In this protocol called *Integrated Real-Time Locking*, a priority-dependent locking protocol is used to adjust the serialization order of active transactions dynamically. Its goal is to execute high priority transactions first so that they are not blocked by uncommitted lower priority transactions, while keeping lower priority transactions from being aborted even in the face of a conflict. This adjustment of the serialization order can be considered as a mechanism to support real-time scheduling.

This protocol is an integrated protocol because it uses different solutions for read/write (rw) and write/write (ww) synchronization, and integrates the solutions of the two subproblems to yield a solution to the entire problem (Bernstein, Hadzilacos and Goodman 1987).

The protocol is similar to optimistic concurrency control (OCC) in the sense that each transaction has three phases, but unlike the optimistic method, there is no validation phase. This protocol's three phases are read, wait, and write. The read phase is similar to that of OCC wherein a transaction reads from the database and writes to its local workspace. In this phase, however, conflicts are also resolved by using transaction priority. While other optimistic realtime concurrency control protocols resolve conflicts in the validation phase, this protocol resolves them in the read phase. In the wait phase, a transaction waits for its chance to commit. Finally, in the write phase, updates are made permanent to the database.

2.1. Read Phase

The read phase is the normal execution of a transaction except that all writes are on private data copies in the local workspace of the transaction instead of on data objects in the database. Such write operations are called *prewrites*. The prewrites are useful when a transaction is aborted, in which case the data in the local workspace is simply discarded. No roll-back is required.

In this phase read-prewrite and prewrite-read conflicts are resolved using a priority based locking protocol. A transaction must obtain the corresponding lock before it reads or prewrites. According to the priority locking protocol, higher priority transactions must complete before a high-priority transaction, it is required to wait until it is sure that its commitment will not lead to the higher priority transaction being aborted.

Suppose T_H and T_L are two active transactions and T_H has higher priority than T_L , there are four possible conflicts as follows.

- (1) $r_{T_H}[x]$ followed by $pw_{T_L}[x]$. The resulting serialization order is $T_H \rightarrow T_L$, hence satisfies the priority order, and does not need to adjust the serialization order.
- (2) $pw_{T_H}[x]$ followed by $r_{T_L}[x]$. Two different serialization orders can be induced with this conflict; $T_L \rightarrow T_H$ with immediate reading, and $T_H \rightarrow T_L$ with delayed reading. Certainly, the latter should be chosen for priority scheduling. The delayed reading in this protocol means blocking of $r_{T_L}[x]$ by the writelock of T_H on x .
- (3) $r_{T_L}[x]$ followed by $pw_{T_H}[x]$. The resulting serialization order is $T_L \rightarrow T_H$, which violates the priority order. If T_L is in read phase, abort T_L . If T_L is in its wait phase, avoid aborting T_L until T_H commits in the hope that T_L gets a chance to commit before

T_H does. If T_H commits, T_L is aborted. But if T_H is aborted by some other conflicting transaction, then T_L is committed. With this policy, we can avoid unnecessary and useless aborts, while satisfying priority scheduling.

- (4) $pw_{T_L}[x]$ followed by $r_{T_H}[x]$. Two different serialization orders can be induced from this conflict; $T_H \rightarrow T_L$ with immediate reading, and $T_L \rightarrow T_H$ with delayed reading. If T_L is in its write phase, delaying T_H is the only choice. This blocking is not a serious problem for T_H because T_L is expected to finish writing x soon. T_H can read x as soon as T_L finishes writing x in the database, not necessarily after T_L completes the whole write phase. If T_L is in its read or wait phase, choose immediate reading.

As transactions are being executed and conflicting operations occur, all the information pertaining to the induced dependencies in the serialization order needs to be retained. In order to maintain this information, we associate the following with each transaction; two sets, *before_trset* and *after_trset*, and a count, *before_cnt*. The *before_trset* (respectively, *after_trset*) of a transaction contains all the active lower priority transactions that must precede (respectively, follow) this transaction in the serialization order. The *before_cnt* of a transaction is the number of higher priority transactions that precede this transaction in the serialization order. When a conflict occurs between two transactions, their dependency is determined and the values of their *before_trset*, *after_trset*, and *before_cnt* are changed accordingly.

By summarizing what we discussed above, we define the locking protocol as follows:

LP1. Transaction T requests a read lock on data object x .

```

for all transactions  $t$  with write_lock( $t,x$ ) do
  if (priority ( $t$ ) > priority ( $T$ ) or  $t$  is in write phase) /* Case 2, 4*/
  then deny the lock and exit;
  endif
enddo

for all transactions  $t$  with write lock ( $t,x$ ) do /*Case 4*/
  if  $t$  is in before_trset $_T$  then abort  $t$ ;
  else if ( $t$  is not in after_trset $_T$ )
    then
      include  $t$  in after_trset $_T$ 
      before_cnt $_t$  := before_cnt $_t$  + 1;
    endif
  endif
enddo
grant the lock;

```

LP2. Transaction T requests a write lock on data object x .

```

for all transactions  $t$  with read lock ( $t,x$ ) do
  if priority ( $t$ ) > priority ( $T$ )
  then /* Case 1 */
    if ( $T$  is not in after_trset $_t$ )

```

```

    then
        include t in after_trset;
        before_cntT := before_cntT + 1;
    endif
else
    if t is in wait phase /* Case 3 */
    then
        if (t is in after_trsetT)
        then abort t;
        else
            include t in before_trsetT;
        endif
    else if t is in read phase
    then abort t;
    endif
    endif
endif
enddo
grant the lock;

```

2.2. Wait Phase

The wait phase allows a transaction to wait until it can commit. A transaction in the wait phase can commit if all transactions with higher priority that must precede it in the serialization order, are either committed or aborted. Since the *before_cnt* of a transaction keeps track of the number of such transactions, the transaction can commit only if its *before_cnt* becomes zero. A transaction in the wait phase may be aborted due to two reasons; if a higher priority transaction requests a conflicting lock or if a higher priority transaction that must follow this transaction in the serialization order commits. Once a transaction in its phase finds a chance to commit, it commits, switches to its write phase and releases all readlocks. The transaction is assigned a final timestamp which is the absolute serialization order.

2.3. Write Phase

Once a transaction is in the write phase, it is considered to be committed. All committed transactions can be serialized by their final-timestamp order. In the write phase, the only work of a transaction is making all its updates permanent in the database. Data items in local workspaces are copied into the database. The write requests of each transaction are sent to the data manager, which carries out the write operations in the database. Transactions submit write requests along with their final timestamps. After each write operation, the corresponding write lock is released. In order to resolve write-write conflicts here, we apply Thomas' Write Rule (TWR) (Bernstein et al. 1987), which just ignores late write requests rather than aborting them.

3. Hybrid Timestamp Interval Protocol

3.1. Key Ideas

This protocol is a combination of OCC and timestamp ordering. One serious problem of OCC is that of *wasted resources*. Because data conflicts are detected and resolved only during the validation phase, transactions can end up aborting after having used resources and time for most of the transaction's execution. The situation becomes even worse because previously performed work has to be redone when the transaction is restarted. The problem of the wasted resources and time becomes even more serious for real-time transaction scheduling, because it reduces the chances of meeting the deadlines of transactions.

Another problem of OCC is *unnecessary aborts*. When a transaction is ready to commit, it is checked whether this transaction is involved in any nonserializable execution. This validation test is usually conducted based on the read sets and write sets of transactions, rather than on actual execution order. Hence sometimes the validation process using the read sets and write sets erroneously concludes that a nonserializable execution has occurred, even though it has not in actual execution. The problem of unnecessary aborts is serious because it results in a waste of resources and time.

The problem of wasted resources is partly remedied with forward validation scheme, because the validation test is conducted against active transactions in their read phase (Haritsa, Carey and Livny 1990; Huang, Stankovic, Ramamritham and Towsley 1991). Early detection and resolution of conflicts can reduce the wasted resources and time. Our protocol presented here also utilizes OCC with forward validation to take the advantage of the early detection and resolution of nonserializable executions. Furthermore, this protocol employs the notion of dynamic timestamp allocation (Bayer, Elhardt, Heigert and Reiser 1982) and dynamic adjustment of serialization order using timestamp interval (Boksenbaum, Cart, Ferrie and Pons 1987). With these, the ability of early detection and resolution of nonserializable execution is improved, and unnecessary aborts are avoided.

3.1.1. OCC with forward validation. The execution of each transaction in this protocol consists of three phases; read, validation, and write, as in other OCC protocols. This protocol uses a forward validation scheme, rather than a backward validation scheme. As mentioned earlier, in forward validation, the validation test is conducted against active transactions in their read phase. When a conflict is detected, either the validating transaction or the conflicting active transaction can be aborted. It is this property that makes OCC with forward validation flexible and allows it to be easily combined with the priority mechanism. The phase-dependent control of OCC and the property of forward validation scheme provide a framework for the following components of the protocol.

3.1.2. Categories of conflicting transactions. Since this protocol uses forward validation conducted against active transactions, when a validation test is performed for a transaction, say T_v , active transactions in the system can be divided into several sets according to their execution history (with respect to that of T_v). First, the set of the active transactions are divided into two sets; a *conflicting set*, which contains transactions in conflict with T_v , and a *nonconflicting set*, which contains transactions not in conflict with T_v . The conflicting

set can be further divided into two sets; a *Reconcilably Conflicting (RC) set* and an *Irreconcilably Conflicting (IC) set*. Transactions in the RC set are in conflict with T_v , but the conflicts are reconcilable, i.e., serializable. However, transactions in the IC set are in conflict with T_v , and the conflicts are irreconcilable, i.e., nonserializable. The formal description of the conditions to categorize these sets of active transactions and the definitions of the terms such as reconcilable conflict and irreconcilable conflict can be found in (Son, Lee and Lin 1992).

The RC transactions do not have to be aborted, but their execution histories have to be adjusted with the timestamp interval facility of this protocol. The IC transactions should be handled with priority-based real-time conflict resolution schemes.

3.1.3. Dynamic timestamp allocation. Another important aspect of this protocol is dynamic timestamp allocation. Most timestamp-based concurrency control protocols use a static timestamp allocation scheme, i.e., each transaction is assigned a timestamp value at its startup time, and a total ordering instead of a partial ordering is built up. This total ordering does not reflect any actual conflict. Hence, it is possible that a transaction is aborted when it requests its first data access (Bayer et al. 1982). Besides the total ordering of all transactions is too restrictive, and degrades the degree of concurrency considerably. With dynamic timestamp allocation, serialization order among transactions are dynamically constructed on demand whenever actual conflicts occur. Only the necessary partial ordering among transactions is constructed instead of a total ordering from the static timestamp allocation.

This dynamic timestamp allocation scheme is possible, because OCC provides a phase-dependent structure of transaction execution. During the read phase, a transaction gradually builds its serialization order with respect to committed transactions on demand whenever a conflict with such transactions occurs. Only when the transaction commits (after passing the validation test), is its permanent timestamp order (i.e., the final serialization order) determined.

3.1.4. Dynamic adjustment of serialization order with timestamp intervals. The dynamic timestamp allocation scheme is made more efficient with a timestamp interval facility (Boksenbaum et al. 1987). More flexibility to adjust serialization order can be obtained using a timestamp interval (initially, the entire range of the timestamp space) assigned to each transaction instead of single value for the timestamp. The timestamp intervals of active transactions preserve the partial ordering constructed by serialization execution. The timestamp interval of each transaction is adjusted (shrunk) whenever the transaction reads or writes a data object to preserve the serialization order induced by committed transactions. When the timestamp interval of a transaction shuts out, it means the transaction has been involved in a nonserializable execution, and the transaction should be restarted. With this facility, it is possible to detect and resolve nonserializable execution early in read phase.

When a transaction, say T_v , commits after its validation phase, the timestamp intervals of those transactions categorized as *reconcilably conflicting* are adjusted, i.e., the serialization order between the validating transaction T_v and its RC transactions are determined. Since the permanent serialization order (final timestamp) of these active transactions is not determined, all we have to do is determine the partial ordering between T_v and these active

transactions by adjusting their timestamp intervals. Therefore these transactions do not have to be aborted even though they are in conflict with the committed transaction, i.e., unnecessary aborts are avoided, unlike other OCC protocols.

3.1.5. Real-Time Conflict Resolution. In order to resolve an irreconcilable conflict means a nonserializable execution. As mentioned, since this protocol is based on OCC with a forward validation scheme, either the validating transaction or the conflicting active transaction can be aborted. To determine which transaction to abort, we can employ the following priority-based conflict resolution schemes (Haritsa et al. 1990; Huang et al. 1991).

- **commit:** When a transaction reaches the validation phase, it commits and notifies all the IC transactions. These IC transactions are immediately restarted.
- **priority abort:** When a transaction reaches its validation phase, it is aborted if its priority is less than that of all the IC transactions. If not, it commits and all the IC transactions are restarted immediately as with the commit scheme.
- **priority sacrifice:** When a transaction reaches its validation phase, it is aborted if at least one IC transaction has a higher priority than the validating transaction; otherwise it commits and all the IC transactions are restarted immediately.
- **priority wait:** When a transaction reaches its validation phase, if its priority is not the highest among the IC transactions, it waits for the IC transactions with higher priority to complete.

3.2. Procedural Description

To execute the proposed protocol, the system maintains an *object table* and a *transaction table*. The object table entries maintain the following information:

RTS: the largest timestamp of the committed transactions that read the data object; and
WTS: the largest timestamp of the committed transactions that wrote the data object.

The transaction table entries maintain the following information:

RS(T): read set of transaction T ;
WS(T): write set of transaction T ; and
TI(T): timestamp interval of transaction T .

We assume that the write set of a transaction is a subset of its read set and there is no blind write. In addition to the timestamp interval assigned to each active transaction, a final timestamp, denoted as $TS(T)$, is assigned to each committed transaction, T , that has passed the validation test.

The read, validation and write phase of transaction execution with the proposed protocol can be summarized as follows:

If T is not aborted during the real-time conflict resolution (if any), then it is validated and committed. The execution of T should be reflected in the serialization order of committed

transactions. Thus a final timestamp for T should be reflected in the serialization order of committed transactions. Thus a final timestamp for T should be chosen such that the order induced by the final timestamp does not destroy the serialization order constructed by the already committed transactions. In fact, any timestamp in the range of $TI(T)$ satisfies this condition because $TI(T)$ preserves the order induced by all committed transactions. Hence any timestamp from $TI(T)$ can be chosen for the final timestamp. Then RTS and WTS for all data objects that T accessed should be updated, if necessary, and finally, the timestamp intervals of all the RC transactions should be adjusted.

4. Conclusions

Time-critical scheduling in real-time database systems has two components: real-time scheduling and concurrency control. While both concurrency control and real-time scheduling are well-developed and well-understood, there is only limited knowledge about the integration of concurrency control and real-time scheduling. Though recently the problem has been studied actively, the proposed solutions are still at an initial stage. A major source of problems in integrating the two is the lack of coordination in the development. They are developed on different objectives and incompatible assumptions (Buchmann 1989).

Most of the proposed work for real-time concurrency control employ a simple method to utilize one concurrency control scheme such as 2PL, TO and OCC, and to consider the priority of operations inherited from the timing constraints of transactions in operation scheduling. This method has an inherent disadvantage of being limited by the concurrency control method used as the base. Since neither of pessimistic nor optimistic concurrency control is satisfactory by itself for real-time scheduling, this simple method using only one control can hardly satisfy the timing requirements of RTDBS. Problems such as excessive blocking, wasted restarts, and priority inversion are serious in RTDBS.

In this paper, we proposed two real-time transaction scheduling protocols which employ a hybrid approach, i.e., a combination of both pessimistic and optimistic approaches. These protocols make use of a new conflict resolution scheme called dynamic adjustment of serialization order, which supports priority-driven scheduling, and avoids unnecessary aborts.

References

- Bayer, R., Elhardt, K., Heigert, J. and Reiser, A. 1982. Dynamic timestamp allocation for transactions in database systems, *Proc. 2nd Int. Symp. Distributed Data Bases*, September, pp 9-20.
- Bernstein, P.A., Hadzilacos, V. and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*, Reading, Mass. Addison-Wesley.
- Boksenbaum, C., Cart, M., Ferrie, J. and Pons, J. 1987. Concurrent certifications by intervals of timestamps in distributed database systems, *IEEE Transactions on Software Engineering*, SE-13, (4), April pp. 409-419.
- Buchmann, A. et al., 1989. Time-critical database scheduling: a framework for integrating real-time scheduling and concurrency control, *Fifth Data Engineering Conference*, February 1989.
- Haritsa, J.R., Carey, M.J. and Livny, M. 1990. Dynamic real-time optimistic concurrency control, *IEEE Real-Time Systems Symposium*, Orlando, Florida, December pp. 94-103.
- Huang, J., Stankovic, J.A., Ramamritham, K. and Towsley, D. 1991. Experimental evaluation of real-time optimistic concurrency control schemes, *VLDB Conference*, Barcelona, Spain, September.
- Son, S.H., Lee, J. and Lin, Y. 1992. Hybrid protocols using dynamic adjustment of serialization order, *Technical Report TR-92-07*, Department of Computer Science, University of Virginia, March.

An Environment for Integrated Development and Evaluation of Real-Time Distributed Database Systems

SANG H. SON

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

(Received June 26, 1990; Revised April 26, 1991)

Abstract. Real-time database systems must maintain consistency while minimizing the number of transactions 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 an environment for investigating distributed real-time database systems. The environment is based on a concurrent programming kernel that supports the creation, blocking, and termination of processes, as well as scheduling and interprocess communication. The contribution of the paper is the introduction of a new approach to system development that utilizes a module library of reusable components to satisfy three major goals: modularity, flexibility, and extensibility. In addition, experiments for real-time concurrency control techniques are presented to illustrate the effectiveness of the environment.

Key Words: *Distributed database, prototyping, synchronization, transaction, real-time.*

1. Introduction

In this paper, we report our experiences with a new approach to integrated development and evaluation of real-time distributed database systems, and present experimental results of various real-time synchronization techniques. The goal of the project is to test the hypothesis that a host environment can be used to significantly accelerate the rate at which we can perform experiments in the areas of operating systems, databases, and network protocols for real-time systems. A tool for developing components of real-time distributed systems and integrating them to evaluate design alternatives is essential for the advance of real-time computing technology. To the best of our knowledge, this is the first successful attempt to develop such a tool as an environment consisting of a hybrid of actual implementation and simulation.

As computers are becoming an essential part of real-time systems, *real-time computing* is emerging as an important discipline in computer science and engineering [1]. 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. Researchers working on developing real-time

This work was supported in part by ONR contract # N00014-88-K-0245, by DOE contract # DEFG05-88-ER25063, by CIT contract # CIT-INF-90-011, and by IBM Federal Systems Division.

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, 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 [2, 3]. 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 [4].

In addition to providing relational access capabilities, distributed real-time database systems offer a means of loosely coupling software processes, making it easier to rapidly update software, at least from a functional perspective. However, with respect to time-driven scheduling and system-timing predictability, they present new problems. One of the characteristics of current database managers is that they do not schedule their transactions to meet response 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 that is 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. In this paper, we report our experiences with a new database prototyping environment. It is constructed to support research in distributed database and operating system technology for real-time applications. 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 20-node distributed database system can be executed in a software environment, it is not necessary to purchase a 20-node distributed system thereby 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 whereas 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. In addition, a prototyping environment provides a friendlier development environment than a target hardware system. The bare-machine environment is the worst possible place in which to explore new software concepts. For example, even the recovery of the event history leading up to an error in a distributed system can be a difficult and, in some cases, impossible, task. Debugging is greatly facilitated in a prototyping environment. The symbolic debugger

of our environment supports the examination of an arbitrary number of execution threads. As a result, the state of a distributed computation can be examined as a whole.

Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation, especially for distributed real-time database systems. Recently, simulators have been developed for investigating performance of several concurrency control algorithms for real-time applications [5, 6]. However, they do not provide a module hierarchy composed from reusable components as in our prototyping environment. Software developed in our prototyping environment will execute in a given target machine without modification of any layer except the hardware interface. In addition, because our environment is a hybrid of prototyping and simulation (i.e., partially implemented and partially simulated), we can easily capture important timing features of the system, whereas it is very hard using simulation only.

A database system must operate in the context of available operating system services. In other words, database operations need to be coherent with the operating system, because correct functioning and timing behavior of database control algorithms depend on the services of the underlying operating system. Unless you have a control over the operating system, investigating timing behavior of a database system does not provide much information. An environment for database systems development must, therefore, provide facilities to support operating system functions and integrate them with database systems for experimentation.

Another important use of a prototyping environment is to analyze the reliability of database control mechanisms and techniques. Because 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 checkpointing for distributed databases have been developed recently [7-11], experimentation to verify their properties and to evaluate their performance has not been performed due to the lack of appropriate test tools.

When a database system is developed, functional completeness and performance of the system are of primary concern. The resulting systems are often not layered or modular in their implementation. However, for experimentation, a layered implementation approach facilitates the rapid evaluation of new techniques. Such a facility improves significantly the capability of the system designer in comparing design alternatives in a uniform environment. In this regard, the concept of developing a methodology for layered implementation of the system and building a library of modules with different performance/reliability characteristics for operating system and database system functions seems promising. The prototyping environment we have developed follows this approach [12, 13].

The rest of the paper is organized as follows. Section 2 presents an informal description of a message-based simulation. Section 3 describes the design principles and the current implementation of the prototyping environment. Section 4 presents experimentations of priority-based synchronization algorithms and multiversion data objects using the prototyping environment. Section 5 concludes the paper.

2. Message-Based Simulation

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 [14] and GPSS [15]. 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, whereas 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.

To prototype a distributed database system on a single-host machine, it is necessary to provide virtual machines for each node of the system being simulated. For that, the process view of a system has been adopted. A distributed system being simulated consists of a number of *processes* that interact with others at discrete instants of time. Processes are basic building blocks of a simulation program. A process is an independent, dynamic entity that manipulates *resources* to achieve its objectives. A resource is a passive object and may be represented by a simple variable or a complex data structure. A simulation program models the dynamic behavior of processes, resources, and their interactions as they evolve in time. Each physical operation of the system is simulated by a process, and the process interactions are called *events*.

In the literature, the notion of a process has been given numerous definitions. The definition used in our model is much the same as that given in [16]: A process is the execution of an interruptible sequential program and represents the unit of resource allocation, such as the allocation of CPU time, main memory, and I/O devices.

We use the client/server paradigm for process interaction in the prototyping environment. The system consists of a set of clients and servers, which are processes that cooperate for the purpose of transaction processing. Each server provides a service to its clients, where a client can request a service by sending a request message (a message of type *request*) to the corresponding server. The computation structure of the system to be modeled can be characterized by the way clients and servers are mapped into processes. For example, a server might consist of a fixed number of processes, each of which may execute requests from every transaction, or it might consist of a varying number of processes, each of which executes on behalf of exactly one transaction.

Internal actions of a process, i.e., actions that do not involve interactions with other processes in the system, are modeled either by the passage of simulation time or by the execution of sequential statements within the process. We use a simulator clock to represent the passage of time in a simulation. The simulator clock advances in discrete steps where each step simulates the passage of time between two events in the system.

In a physical system, each process makes independent progress in time if the resources they need are available, and many processes execute in parallel. In its simulation, the multiple processes of a physical system must be executed simultaneously on one processor. This

simultaneity is achieved in the prototyping environment by supporting a simultaneous execution of multiple processes in a single address space.

A message-based prototyping environment can be of enormous benefit in designing and testing emerging systems, such as real-time systems, and in comparing and improving algorithms that are applicable to many different systems. One such benefit is that the software to be used in an actual system can be developed using the environment. The prototyping environment can support a simulated environment, actual hardware, or a "hybrid" mode in which some of the modules are implemented in hardware and some are simulated. In this way, it is irrelevant to the software developer using the environment whether or not all or part of the software is running on hardware. When the system is running in a hybrid mode, the virtual clock used for performance measurement is updated by the actual time used for direct execution, making performance measurements correct.

3. Structure of the Prototyping Environment

The prototyping environment is designed to facilitate easy extensions and modifications. Server processes can be created, relocated, and new implementations of server processes can be dynamically substituted. The prototyping environment efficiently supports a spectrum of real-time database functions at the operating level and facilitates the construction of multiple database systems with different characteristics. For experimentation, system functionality can be adjusted according to application-dependent requirements without much overhead for a new system setup. Because one of the design goals of the prototyping environment is to conduct an empirical evaluation of the design and implementation of real-time distributed database systems, it has built-in support for performance measurement of both elapsed time and blocked time for each transaction.

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. The StarLite kernel supports process control to create, ready, block, and terminate processes. It also supports the semaphore abstraction to be used by higher-level modules in resource control, critical section implementation, and synchronous message passing. The internal structure of the kernel follows the well-known *client-server model* [17], in which most of the operating system operates as server processes in the same address space as client processes, with the kernel merely handling message communication between various processes. Figure 2 shows an instance of this model. This structure is particularly useful for extensible systems such as our prototyping environment, as additional or alternative functionality

can easily be provided by creating a new server, instead of changing and recompiling the kernel.

Scheduler in the kernel maintains a virtual clock and provides the **hold** primitive to control the passage of time. The benefit of a virtual clock is that any number of performance monitoring operations may be performed at an instant of virtual time. If a physical clock were embedded, the monitoring activities themselves would interfere with other system activities and add to the execution time, resulting in incorrect performance measures.

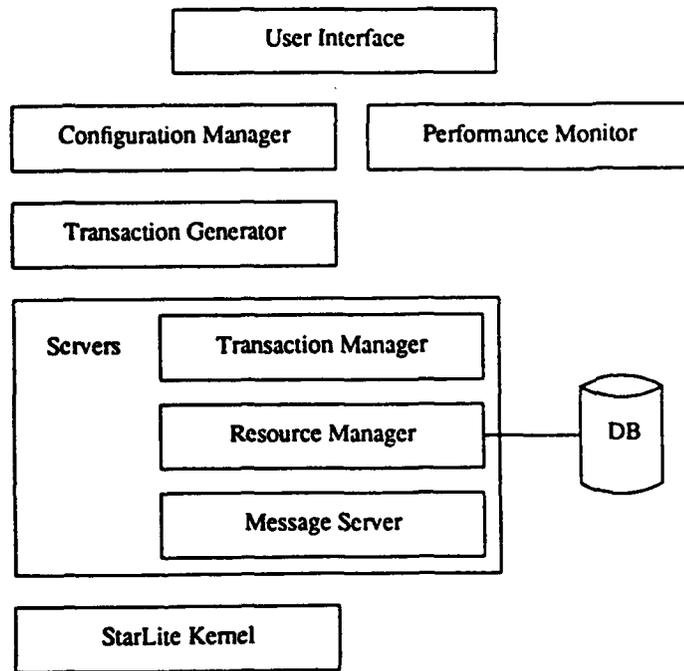


Figure 1. Structure of the prototyping environment.

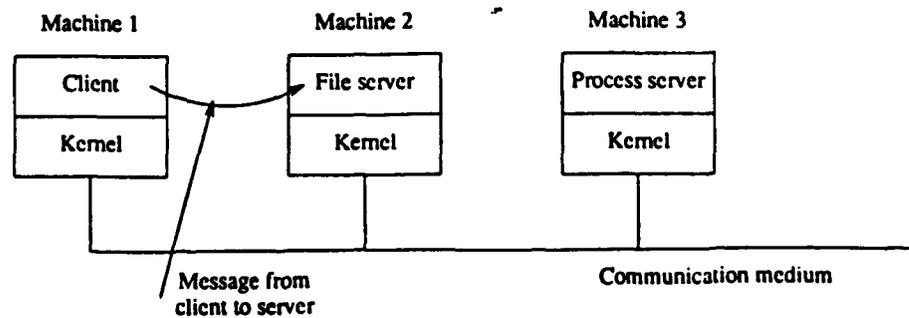


Figure 2. Client-server model.

The kernel also provides the capability of isolating overhead imposed by each system component. For instance, total time at each node can be divided into CPU time and I/O time, to determine the computation-intensive and I/O-intensive functions and investigate the distribution of tasks around the system so as to maximize parallelism. The 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:

1. System configuration: number of sites and the number of server processes at each site, topology and communication costs
2. Database configuration: database at each site with user-defined structure, size, granularity, and levels of replication
3. 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
4. Concurrency control: locking, timestamp ordering, and priority based.

The UI initiates the configuration manager (CM), which initializes necessary data structures for transaction processing based on user specification. The database at each site consists of different number of files, and each file consists of different number of records. The database structure can be made complicated if necessary. However, we use a simple file access because investigating synchronization problems does not require complex database structures.

The CM invokes the transaction generator at an appropriate time interval to generate the next transaction to form a Poisson process of transaction arrival. The environment is flexible enough to generate any number of transactions with different characteristics. The user can specify his or her own procedure for transactions. At initialization time, the user-specified procedure is converted into a transaction process. Furthermore, the prototyping environment supports the facility that allows mixing system generated transactions with user-specified ones. It is very desirable to have such a capability as the user can setup any workload that represents the situation to be simulated, with or without system-generated background workload.

A transaction is distinguished from the other processes in the system by its behavior. To the system, the only distinction between transactions and server processes is the Port-Tags on which each receives messages. When a transaction is generated, it is assigned an identifier that is unique among all transactions in the system. Each transaction is also assigned a globally unique timestamp hidden within a single module. The advantage of extracting the definition and assignment of the timestamp from its use is that it provides a means of uniquely assigning timestamps that are independent from any specific implementation.

The timestamp assignment is closely related to the clocks in the system. In a sequential simulation, a single clock suffices to order events in the system. An event is taken off the event queue, and the global clock is advanced to the time required for the event to occur. Events are related in time by their relation to the global clock. In prototyping distributed environments, no such global clock is available. Time is referred to by local clocks, which is maintained at each site and visible only to processes at that site. Ordering of events in

terms of the global time, therefore, depends on the proper synchronization of local clocks. In our environment, clocks are synchronized by intersite communication. An intersite message includes the clock value of the sender site at the time the message is sent. If the sum of this clock value and the propagation delay between the sites is greater than the clock value at the receiver site, the receiver increments its clocks by the difference between the sum and its clock value. In this way, all succeeding events at the receiver site can be said to occur *after* the sending of the message. This satisfies our intuitive notion of "happens before" relationship [18].

Transaction execution consists of read and write operations. Each read or write operation is preceded by an access request sent to the resource manager, which maintains the local database at each site. Each transaction is assigned to the transaction manager (TM). The TM issues service requests on behalf of the transaction and reacts appropriately to the request replies. For instance, if a transaction requests access to a 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 a deadlock, TM receives 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. Figure 3 illustrates a queueing model adopted for transaction processing.

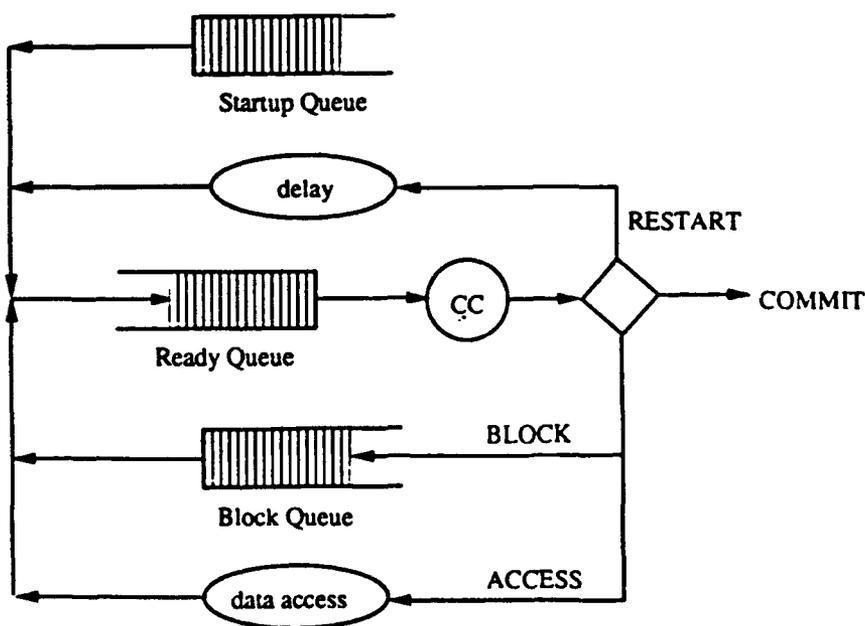


Figure 3. Simulation model.

Transactions are generated and put into the start-up queue. When a transaction is started, it leaves the start-up and enters the ready queue. The transaction at the top of the queue is selected to run. The current running transaction sends requests to the concurrency controller (CC) implemented in the resource manager. The transaction may be blocked and placed in the block queue. It may also be aborted and restarted. In such a case, it is first delayed for a certain amount of time and then put in the ready queue again. When a transaction in the block queue is unblocked, it leaves the block queue and is placed in the ready queue again.

In prototyping distributed database systems, a communication network is an important component to be simulated because the system performance depends heavily on the topology and communication protocols used. However, in many database simulators, the communication subsystem is either ignored or simplified by adding communication cost to the transaction processing time. Our prototyping environment uses a different approach by providing a virtual communication network that actually runs a layered communication protocol on a network topology specified by the user. Because the communication module is a separate building block in the prototyping environment, the user can change it to simulate different requirements of the application.

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 supports both Ada-style rendezvous (synchronous) as well as asynchronous message passing. Interprocess communication within a site does not go through the message server; processes send and receive messages directly through their associated ports. The interprocess communication structure is designed to provide a simple and flexible interface to the client processes of the application software independent of the low-level hardware configurations. It is split into three levels of hierarchy: transport, network and physical layers.

The transport layer is the interface to the application software, thus it is designed to be as abstract as possible in order to support different port structures and various message types. In addition, application level processes need not know the details of the destination device. The invariant built into the design of the interprocess communication interface is that the application level sender allocates the space for a message, and the receiver deallocates it. Thus, it is irrelevant whether or not the sender and receiver share memory space, i.e., whether or not the physical layer on the sender's side copies the message into a buffer and deallocates it at the sender's site, and the physical layer at the receiver's site allocates space for the message. This enables prototyping distributed systems or multiprocessors with no shared memory, as well as multiprocesses with shared memory space. When the latter is prototyped, only addresses need to be passed in messages without intermediate allocation and deallocation.

The physical layer of message passing simulates the physical sending and receiving of bits over a communication medium, i.e., it is for intersite message passing. The device number in the interface is simply a cardinal number, this enables the implementation to be simple and extensible enough to support any application. To simulate sending or to

actually send over an Ethernet in the target system, for example, a module could map network addresses onto the cardinal numbers. To send from one processor to another in a distributed system, the cardinals can represent processor numbers.

Messages are passed to specific processes at specific sites in the network layer of the communication interface. This layer separates the transport and the physical layers so that the transport-layer interface can be processor and process independent and the physical layer interface need be concerned only with the sending of bits from one site to another. The transport layer interface of the communication subsystem is implemented in the transport module. A transport-level Send is made to an abstraction called a *PortTag*. This abstraction is advantageous because the implementation (i.e., what a PortTag represents) is hidden in the Ports module. Thus the PortTag can be mapped onto any port structure or the reception point of any other message passing system. The transport-level Send operation builds a packet consisting of the sender's PortTag, used for replies, the destination PortTag, and the address of the message. It then retrieves from the destination PortTag the destination device number. If this number is the same as the sender's, the Send is an intrasite message communication, and hence the network-level Send is performed. Otherwise the send requires the physical module for intersite communication. Note that accesses to the implementation details of the PortTag are restricted to the module that actually implements it; this enables changing the implementation without recompiling the rest of the system.

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 the number of aborts.

Because 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.

4. Prototyping Real-Time Database Systems

Section 3 described the structure of the prototyping environment with some of its advanced features. In this section, we present real-time database systems implemented using the prototyping environment. The objectives of our study using the prototyping environment are (1) to evaluate the prototyping environment itself in terms of correctness, functionality, and modularity, (2) to compare performance between two-phase locking and priority-based synchronization algorithms and between a multiversion database and its corresponding single-version database, through the sensitivity study of key parameters that affect performance.

Compared with traditional databases, real-time database systems have a distinct feature: they must satisfy the timing constraints associated with transactions. In other words, "time" is one of the key factors to be considered in real-time database systems. The timing constraints of a transaction typically include its ready time and deadline, as well as temporal

consistency of the data accessed by it. Transactions must be scheduled in such a way that they can be completed before their corresponding deadlines expire. For example, both the update and query on a tracking data of a missile must be processed within the given deadlines otherwise the information provided could be of little value. In such a system, transaction processing must satisfy not only the database consistency constraints but also the timing constraints.

The prototyping environment we have developed is especially useful for investigating timing behavior of real-time transactions as we can control all the system components. An alternative to the prototyping approach is to develop a system on a bare machine, based on a specialized real-time kernel. The ARTS [19] and the RT-CARAT [20] systems take this approach. Difficulties with such an approach are that (1) it takes much more effort to develop, (2) the system is strongly coupled with its hardware and hence hard to change its timing characteristics when needed, and (3) the system is not portable as it is implemented in the target environment.

4.1. Steady-State Estimation

In order to show that the results we get from experiments represent the performance of the system in steady states, we have performed experiments to check if the system were allowed to run for any length of time greater than certain threshold value, the variation in results would be within some tolerable interval. We have implemented a well-known synchronization protocol, two-phase locking (2PL), for the following system and workload configuration:

- 8 sites with fully interconnected network
- multiprogramming level of 10
- 75% read-only and 25% update transactions
- read-only transactions access 3% of the database
- update transactions access 1% of the database
- database consists of 500 unreplicated objects
- Poisson distribution of transaction arrivals

Figure 4 shows the average response time of transactions using the 2PL. It shows that the average response time begins to stabilize at 3000 simulation time units and varies only slightly from then on. The lower response time up to 3000 time units are due to the first set of transactions that benefits from a lower initial multiprogramming level and potential conflicts. In addition, because transactions requiring longer execution time will increase the average response time when they complete, they do not contribute to the average response time during the early stage of transaction execution if they were in the initial group of transaction. These initial characteristics are gradually erased from the average performance.

In addition, as we increase the time for experiments, the average response time is determined from an increasing number of transactions. For example, at 100 time units, the number of transactions contributing to the mean is approximately 12. At 4000, it is approximately 60. Thus the overall behavior of the system becomes less and less subject to the behavior

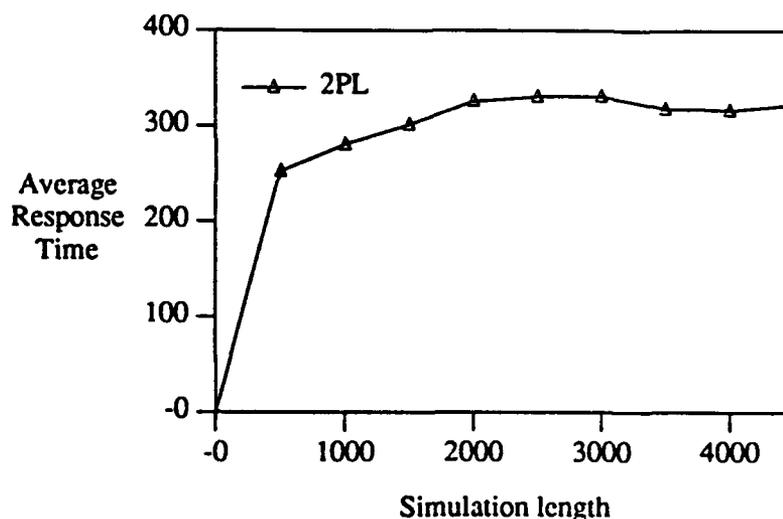


Figure 4. Response-time stability.

of individual transactions. From the graph and characteristics of our environment, we concluded that an experiment must run at least 3500 time units before it starts to capture the steady state behavior of the system.

4.2. Priority-Based Synchronization

Real-time databases are often used by applications such as tracking. Tasks in such applications consist of both computing (signal processing) and database accessing (transactions). A task can have multiple transactions, which consist of a sequence of read and write operations operating on the database. Each transaction will follow the two-phase locking protocol, 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 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, there is a need 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* [6]. 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 that 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 because 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 [21]. 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, although 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 [22]. A transaction is said to be *active* if it has started but not yet completed its execution. A transaction can be active in one of two states: executing or being preempted in the middle of its execution. The idea of total priority ordering is that the real-time locking protocol ensures that each active transaction is executed at some priority level, taking priority inheritance and read/write semantics into consideration.

4.3. Total Ordering by Priority Ceiling

To ensure the total priority ordering of active transactions, three priority ceilings are defined for each data object in the database: the write-priority ceiling, the absolute-priority ceiling, and the rw-priority ceiling. The write-priority ceiling of a data object is defined as the priority of the highest-priority transaction that may write into this object, and absolute-priority ceiling is defined as the priority of the highest-priority transaction that may read or write the data object. The rw-priority ceiling is set dynamically. When a data object is write locked, the rw-priority ceiling of this data object is defined to be equal to the absolute priority ceiling. When it is read locked, the rw-priority ceiling of this data object is defined to be equal to the write-priority ceiling. The priority ceiling protocol is premised on systems with a fixed priority scheme. The protocol consists of two mechanisms: *priority inheritance* and *priority ceiling*. With the combination of these two mechanisms, we get the properties of freedom from deadlock and a worst case blocking of at most a single lower priority transaction.

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

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

4.4. Performance Evaluation

Various statistics have been collected for comparing the performance of the priority-ceiling protocol with other synchronization control algorithms. 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 that 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 do not present all our results but have selected the graphs that 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: $\%missed = 100 * (\text{number of deadline-missing transactions} / \text{number of transactions processed})$. A transaction is processed if either it executes completely or it is aborted. We assume that all the transactions are *hard* in the sense that there will be no value for completing the transaction after its deadline. Transactions that miss the deadline are aborted and disappear from the system immediately with some abort cost. We have used the 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 conflict would occur frequently. The high conflict rate allows synchronization protocols to play a significant role in the system performance. We choose the arrival rate so that protocols are tested in heavily loaded rather than lightly loaded system. In order to design 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 [5].

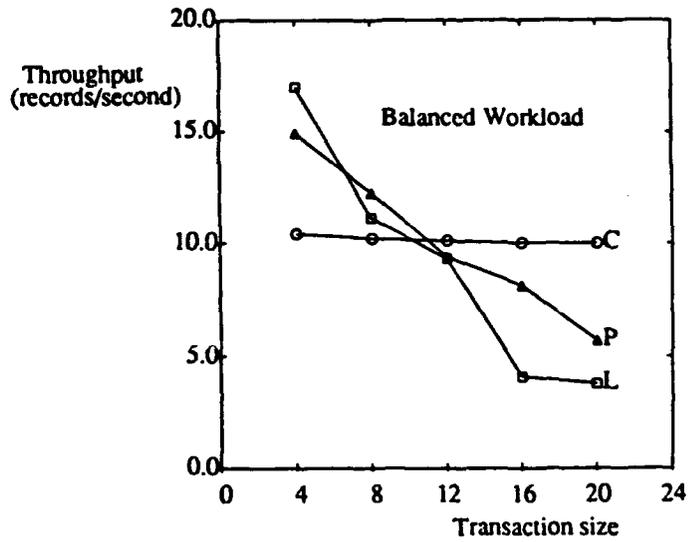
We normalize the transaction throughput in records accessed per second for successful transactions, not in transactions per second, in order to account for the fact that bigger transactions need more database processing. The normalization rate is obtained by multiplying the transaction completion rate (transactions/second) by the transaction size (database records accessed/transaction).

In Figure 5, 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 different sizes with balanced workload and I/O bound workload. The two important factors affecting the performance of locking protocols are their abilities to resolve the locking conflicts and to perform I/O and transactions in parallel. When the transaction size is small, there is little locking conflict and the problem such as deadlock and priority inversion has little effect on the overall performance of a locking protocol. On the other hand, when the transaction size becomes large, the probability of locking conflicts rises rapidly. In fact, the probability of deadlocks goes up with the fourth power of the transaction size [23]. Hence, we would expect that the performance of protocols will be dominated by their abilities to handle locking conflicts when transaction size is large.

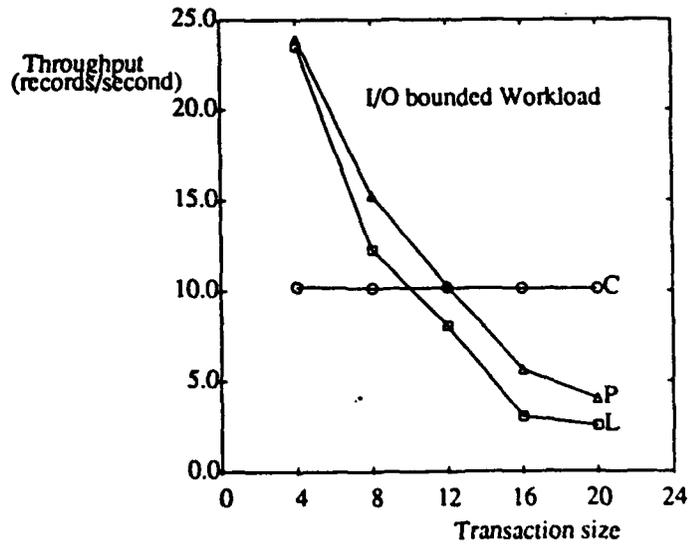
As illustrated in Figure 5, the performance of the two-phase locking protocol, with or without priority assignments to transactions, degrades very fast when transaction size increases. This can be attributed to the inability of this protocol to prevent deadlock and priority inversions. On the other hand, the priority-ceiling protocol handles locking conflicts very well. The protocol performs much better than the two-phase locking protocol when the transaction size is large. The main weakness of the priority-ceiling protocol is its inability to perform I/O and transactions in parallel. For example, suppose that transaction T has lock on O_1 and it now wants to lock data object O_2 . Unfortunately, O_2 is not in the main memory. As a result, T is suspended. However, neither are transactions with priorities lower than the rw-priority ceiling of O_1 allowed to execute. This could lead to the idling of the processor until either O_2 is transferred to the main memory or a transaction whose priority is higher than the rw-priority ceiling arrives. We refer this type of blocking as I/O blocking. When the transaction size is small, the locking conflict rate is small. Hence, the two-phase locking protocol performs well. However, due to I/O blocking the throughput of the priority ceiling protocol is not as good as that of the two-phase locking protocol, especially when the workload is I/O bounded.

Because I/O cost is one of the key parameters in determining performance, we have investigated an approach to improve system performance by performing I/O operation before locking called the *intention I/O*. In the intention mode of I/O operation, the system pre-fetches data objects that are in the access lists of transactions submitted without locking them. This approach will reduce the locking time of data objects, resulting in higher throughput. As shown in Figure 6, intention I/O improves throughput of both the two-phase locking and the ceiling protocol. However, improvement in the ceiling protocol is much more significant. This is because intention I/O effectively solves the I/O blocking problem of the priority ceiling protocol.

Another important performance statistics is the percentage of deadline missing transactions, since the synchronization protocol in real-time database systems must satisfy the timing constraint of individual transaction. In our experiments, each transaction's deadline is set to proportional 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 7, the percentage of deadline missing transactions increases sharply for the two-phase locking protocol as the transaction size increases due to its inability to deal with deadlock and to give preference to transactions with shorter deadlines. Two-phase locking with priority



a) balanced workload transaction



b) I/O bounded workload transaction

Figure 5. Transaction throughput. C: priority_ceiling protocol. P: 2-phase locking protocol with priority mode. L: 2-phase locking protocol without priority mode.

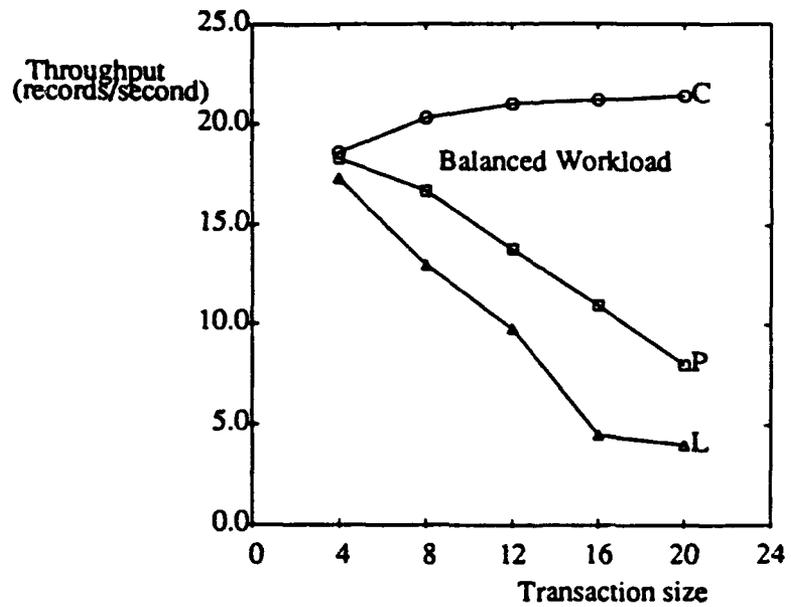


Figure 6. Transaction throughput with intention I/O.

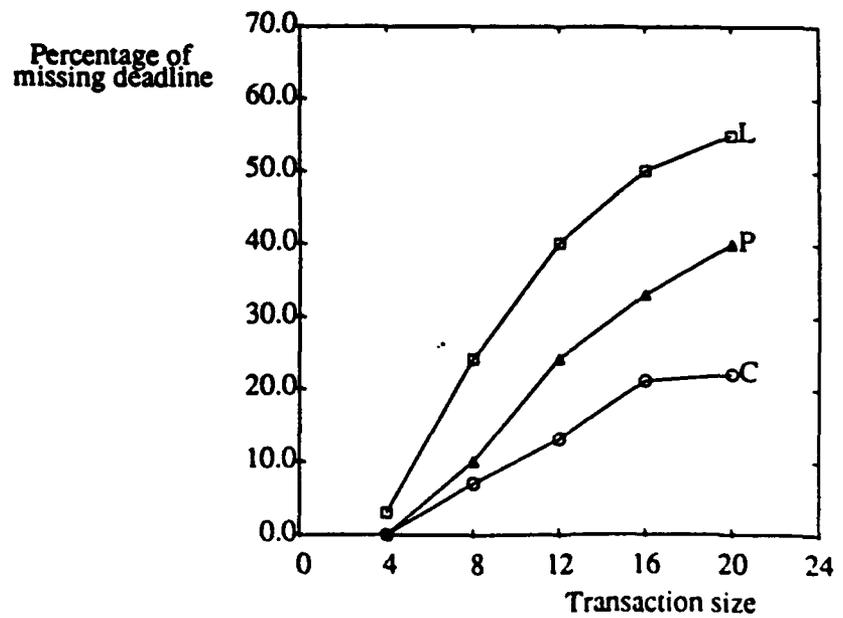


Figure 7. Percentage of missing deadline.

assignment performs somewhat better because the timing constraints of transactions are considered, although the deadlock and priority inversion problems still handicap its performance. The priority-ceiling protocol has the best relative performance because it addresses both the deadlock and priority inversion problem. A drawback of the priority-ceiling protocol from a practical viewpoint is that it needs knowledge of all transactions that will be executed in the future. This may be a very strong requirement to satisfy in some applications.

The priority-ceiling protocol takes a conservative approach. It is based on two-phase locking and employs only blocking, but not rollback, to solve conflicts. For conventional database systems, it has been shown that optimal performance may be achieved by compromising blocking and rollback [24]. For real-time database systems, we may expect similar results. Aborting a few low priority transactions and starting them later may allow high priority transactions to meet their deadlines, resulting in improved system performance. Several concurrency control protocols based on optimistic approach have been proposed [9, 11, 25]. They incorporate priority-based conflict resolution mechanisms, such as *priority wait*, that makes low-priority transactions wait for conflicting high-priority transactions to complete. However, this approach of detecting conflicts during validation phase degrades system predictability. A transaction is detected as being late when it actually misses its deadline as the transaction is only aborted in the validation phase.

4.5. Multiversion Database System

To illustrate the effectiveness of the prototyping environment, we have investigated the performance of a multiversion database system. There is no correlation between the priority-ceiling protocol study and the multiversion database study.

In a multiversion database system, each data object consists of a number of consecutive versions. The objective of using multiple versions in real-time database systems is to increase the degree of concurrency and to reduce the possibility of rejecting user requests by providing a succession of views of data objects. One of the reasons for rejecting a user request is that its operations cannot be serviced by the system. For example, a read operation has to be rejected if the value of data object it was supposed to read has already been overwritten by some other user request. Such rejections can be avoided by keeping old versions of each data object so that an appropriate old value can be given to a tardy read operation. In a system with multiple versions of data, each write operation on a data object produces a new version instead of overwriting it. Hence, for each read operation, the system selects an appropriate version to read, enjoying the flexibility in controlling the order of read and write operations. When a new version is created, it is *uncertified*. Uncertified versions are prohibited from being read by other transactions to guarantee cascaded-abort free [26]. A version is *certified* at the commit time of the transaction that generated the version.

The multiversion database system we have implemented is based on timestamp ordering. Each data object is represented as a list of versions, and each version is associated with timestamps for its creation and the latest read, and a valid bit to specify whether the version is certified. The multiversion concurrency control scheme we have implemented is called the "multiversion timestamp ordering method" and is proved to satisfy the serializability [26].

Each transaction consists of read and write requests for data objects. Read requests are never rejected in a multiversion database system if all the versions are retained. A read operation does not necessarily read the latest committed version of a data object. A read request is transformed to a version-read operation by selecting an appropriate version to read. The timestamp of a read request is compared with the write-timestamp of the highest available version. When a read request with timestamp T is sent to the resource manager, the version of a data object with the largest timestamp less than T is selected as the value to be returned. Figure 8 shows an example of a read operation with a timestamp "11".

The timestamp of a write request is compared with the read timestamp of the highest version of the data object. A new version with the timestamp greater than the read-timestamp of the highest certified version is built on the upper level, with the valid bit reset to indicate that the new version is not certified yet. In order to simplify the concurrency control mechanism, we allow only one temporary version for each data object. Inserting a new version in the middle of existing valid versions is not allowed.

The experiment was conducted to measure the average response time and the number of aborts for a group of transactions running on a multiversion database system and its corresponding single-version system. Two groups of transactions with different characteristics (e.g., type and number of access to data objects) were executed concurrently. The objective was to study the sensitivity of key parameters on those two performance measures. Here we present our findings briefly.

Performance is highly dependent on the set size of transactions. As shown in Figure 9, a multiversion database system outperforms the corresponding single-version system for the type of workload under which they are expected to be beneficial: a mix of small update transactions and larger read-only transactions. The reason for this is that, in a multiversion database system, read requests have higher priority than the write requests, whereas the priority for read requests is not provided in a single-version system. Therefore, in a single-version system, the probability of rejecting a read request is equal to that of a write request. The experiment shows that a single-version database system outperforms its multiversion counterpart for a different transaction mix.

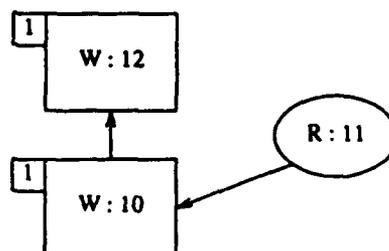


Figure 8 A read operation with two certified versions of a data object.

It was observed that the performance of a multiversion system in terms of the number of aborts is better than its single-version counterpart for a mix of small update transactions and larger read-only transactions. Similar experiments have been performed by changing the database size and the mean interarrival time of transactions. It was found, however, that the main result remains the same. From these experiments, it becomes clear that among the four variables we studied, the set size of transactions is the most sensitive parameter for determining the performance of a multiversion database system. This experiment demonstrates the expressive power and performance evaluation capability of the prototyping environment.

5. Conclusions

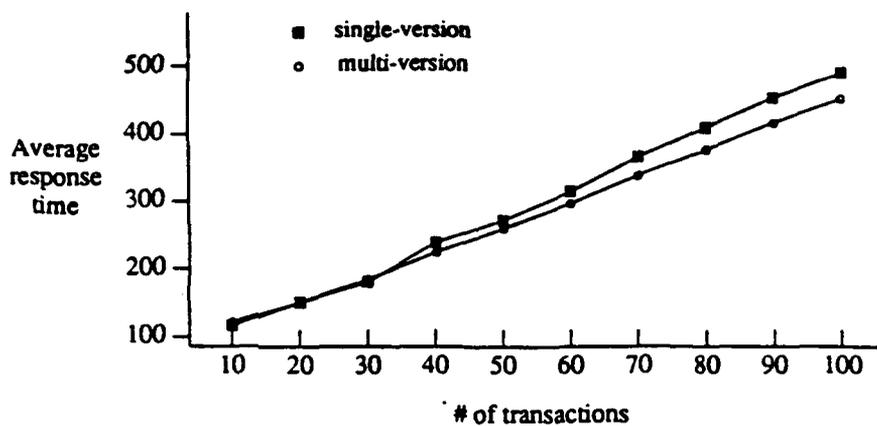
Prototyping large software systems is not a new approach. However, methodologies for developing a prototyping environment for real-time database systems have not been investigated in depth in spite of its potential benefits. In this paper, we have presented a prototyping environment that has been developed 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 controls techniques, and even an integrated evaluation of database systems.

There are three main goals to be achieved in developing a prototyping environment for real-time database systems: modularity, flexibility, and extensibility. Modularity enables the environment to be easily reconfigured as any subset of the available modules can be combined to produce a new testing environment.

An additional benefit of the "right" modularity is that actual system software can be developed in the prototyping environment and then ported to the target machine. This is enabled by the use of technology-independent interfaces that are general enough to support any target system architecture. In addition to the portability, programs may be run in a "hybrid" mode, that is, not all service calls need be simulated. For example, file system calls in the application program can be intercepted by the interpreter and directed to the existing host file system. Then, as a file system is developed, the file system calls can be directed to it. If the file system is not necessary or is not the focus of the current research, it need not be developed. This feature of the prototyping environment allows the developer to focus on only pertinent design issues.

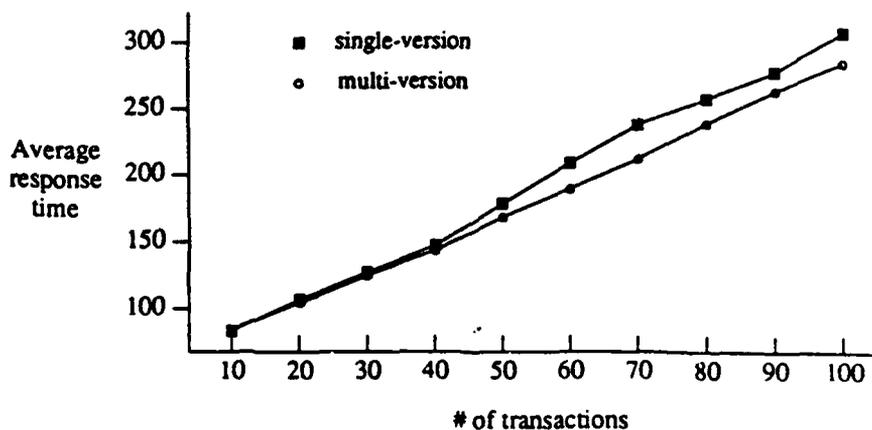
Flexibility enables the prototyping environment to be applicable over a wide range of configurations and system parameters. One of the keys to achieving this goal is to design interfaces whose operations are independent both of the implementation technology and the context in which they are used. For example, the user-level Send operation sends an array of bytes to an abstract data type, the PortTag. Thus this operation can be used to send any packet type to any destination, be it local or distant.

The third goal is that the prototyping environment be extensible enough to model additional features of particular systems by adding modules without affecting the operation of or requiring the recompilation of existing modules. For instance, the implementation can be extended to model the operation of different types of I/O devices of different speeds by modifying the implementation module that performs the read and write operations. One way to modify the implementation would be to delay for a period depending on the address

**PARAMETERS**

Group 1 : Setsize = 10, Type = READ-only, Transaction Ratio = 80%

Group 2 : Setsize = 2, Type = WRITE-only, Transaction Ratio = 20%

**PARAMETERS**

Group 1 : Setsize = 10, Type = READ-only, Transaction Ratio = 50%

Group 2 : Setsize = 2, Type = WRITE-only, Transaction Ratio = 50%

Figure 9. Average transaction-response time.

passed to the read or write operation. Reading from a disk might be indicated by one range of addresses and take some time while reading from a tape drive might be indicated by another range and presumably take longer. However, because the interface of this module is device independent, changing the implementation to process I/O requests at different speed will not affect any of the modules that request I/O operations. Therefore, time and effort for system reconfiguration can be reduced.

Expressive power and performance evaluation capability of our prototyping environment has been demonstrated by implementing real-time database systems and investigating the performance characteristics of the priority-ceiling protocol and multiversion databases.

In real-time database systems, transactions must be scheduled to meet their timing constraints. In addition, the system should support a predictable behavior such that the possibility of missing deadlines of critical tasks could be informed ahead of time, before their deadlines expire. Priority-ceiling protocol is one approach to achieve a high degree of 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 compared with two-phase locking protocols, it reduces the number of deadline-missing transactions.

Using the prototyping environment, we have shown that in general, a database system with a multiversion concurrency control algorithm performs better for processing read requests. Read requests that would be aborted in a single-version database system due to conflicts may be successfully processed in a multiversion system using older versions. Therefore, when the read requests dominate the transaction load, and there is a high probability for abort of read-only transactions due to conflicts, a multiversion system outperforms its corresponding single-version system. The relative size of the read and write sets of transactions is an important factor affecting the performance. Although the actual performance figures will vary depending on workload and implementation details, we believe that our results provide a good picture of the costs and benefits associated with the multiversion approach to concurrency control.

Real-time distributed database systems 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 [27], it should not be necessarily based only on relative deadlines because 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 [5, 7, 26] 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 real-time database systems.

References

1. K.G. Shin, "Introduction to the special issue on real-time systems." *IEEE Trans. Computers*, vol. 36, no. 8, pp. 901-902, August 1987.
2. *Seventh IEEE Workshop on Real-Time Operating Systems and Software*, University of Virginia, Charlottesville, May 1990.
3. *ONR Workshop on Foundations of Real-Time Computing*, Washington, D.C., October 1990.
4. S.H. Son, ed. *ACM SIGMOD Record* vol. 17, no. 1, *Special Issue on Real-Time Database Systems*, March 1988.
5. R. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: A performance study," *VLDB Conf.*, pp.1-12, September 1988.
6. R. Rajkumar, "Task synchronization in real-time systems." Ph.D dissertation, Carnegie-Mellon University, Pittsburgh, PA, August 1989.
7. J.W.S. Liu, K.J. Lin, and S. Natarajan, "Scheduling real-time periodic jobs using imprecise results," *Real-Time Systems Symp.* San Jose, CA, December 1987.
8. H. Korth, "Triggered real-time databases with consistency constraints," *16th VLDB Conf.*, Brisbane, Australia, August 1990.
9. Y. Lin and S.H. Son, "Concurrency control in real-time databases by dynamic adjustment of serialization order," *11th IEEE Real-Time Systems Symp.*, Orlando FL, December 1990.
10. S.H. Son and A. Agrawala, "Distributed checkpointing for globally consistent states of databases," *IEEE Trans. Software Eng.*, vol. 15, no. 10, pp. 1157-1167, October 1989.
11. S.H. Son and J. Lee, "Scheduling real-time transactions in distributed database systems," *7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, VA, pp. 39-43, May 1990.
12. R. Cook and S.H. Son, "The StarLite Project," *4th IEEE Workshop on Real-Time Operating Systems*, Cambridge, MA, pp. 139-141, July 1987.
13. S.H. Son, "A message-based approach to distributed database prototyping," *5th IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, pp. 71-74, May 1988.
14. P. Kiviat, R. Villareau, and H. Markowitz, *The SIMSCRIPT II Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1969.
15. T. Schriber, *Simulation Using GPSS*, New York: Wiley, 1974.
16. P. Hansen Brinch, "Distributed processes: A concurrent programming concept," *Comm. the ACM*, vol. 21, no. 11, pp. 934-941, November 1978.
17. A. Tanenbaum, *Operating Systems Design and Implementation*, Englewood Cliffs, NJ: Prentice-Hall, 1987.
18. L. Lamport, "Time, clocks and ordering of events in distributed systems," *Commun. ACM*, vol. 21, no. 7, pp. 558-565, July 1978.
19. H. Tokuda and C. Mercer, "ARTS: A distributed real-time kernel," *ACM Operating Syst. Rev.*, vol. 23, no. 3, pp. 29-53, July 1989.
20. J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, "Real-time transaction processing: Design, implementation and performance evaluation," *Tech. Rep. TR-90-43*, Dept. of Computer and Information Science, University of Massachusetts, May 1990.
21. L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocol: An approach to real-time synchronization," *Technical Report*, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA, 1987.
22. L. Sha, R. Rajkumar, and J. Lehoczky, "Concurrency control for distributed real-time databases," *ACM SIGMOD Record*, vol. 17, no. 1, *Special Issue on Real-Time Database Systems*, pp. 82-98, March 1988.
23. J. Gray, et al., "A straw man analysis of probability of waiting and deadlock," *IBM Research Report*, RJ 3066, 1981.
24. P. Yu and D. Dias, "Concurrency control using locking with deferred blocking," *6th Int. Conf. Data Engineering*, Los Angeles, pp. 30-36, February 1990.
25. J. Haritsa, M. Carey, and M. Livny, "On being optimistic on real-time constraints," *ACM PODS Symp.*, April 1990.
26. P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and recovery in Database Systems*, City: Addison, Wesley, 1987.
27. J. Stankovic, "Misconceptions about real-time computing," *IEEE Computer*, vol. 21, no. 10, pp. 10-19, October 1988.

DISTRIBUTION LIST

- 1 - 2 Office of Naval Research
 Chief of Naval Research
 Code 1267/Annual Report
 Ballston Tower One
 Room 528
 800 N. Quincy Street
 Arlington, VA 22217-5660
- Attention: Dr. James G. Smith, Program Manager
 Information Systems
- 3 - 8 Director
 Naval Research Laboratory
 Washington, DC 20375
- Attention: Code 2627
- 9 - 20 Defense Technical Information Center, S47031
 Building 5, Cameron Station
 Alexandria, VA 22314
- 21 Ms. Charlotte Luedeke
 Administrative Contracting Officer
 Office of Naval Research Resident Representative
 2135 Wisconsin Avenue, N. W.
 Suite 102
 Washington, DC 20007
- 22 - 23 S. H. Son
- 24 A. K. Jones
- 25 - 26 E. H. Pancake, Clark Hall
- * SEAS Postaward Administration
- 27 SEAS Preaward Administration Files

JO#4745:ph