An Annual Progress Report
Contract No. N00014-86-K-0245
October 1, 1988 - September 30, 1989

THE STARLITE PROJECT

Applied Math and Computer Science
Dr. James G. Smith
Program Manager, Code 1211

Computer Science Division
Dr. Andre van Tilborg
Director, Code 1133

Submitted to:

Director
Naval Research Laboratory
Washington, DC 20375

Attention: Code 2627

Submitted by:

R. P. Cook
Associate Professor

S. H. Son
Assistant Professor

Report No. UVA/525410/CS90/103
October 1989

SCHOOL OF ENGINEERING AND
APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

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.

An Annual Progress Report
Contract No. N00014-86-K-0245
October 1, 1988 - September 30, 1989

THE STARLITE PROJECT

Applied Math and Computer Science
Dr. James G. Smith
Program Manager, Code 1211

Computer Science Division
Dr. Andre van Tilborg
Director, Code 1133

Submitted to:

Director
Naval Research Laboratory
Washington, DC 20375

Attention: Code 2627

Submitted by:

R. P. Cook
Associate Professor

S. H. Son
Assistant Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/525410/CS90/103
October 1989

Copy No. 15

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1989	Annual: Oct. 1, 1988 - Sept. 30,1989

4. TITLE AND SUBTITLE

The Starlite Project

5. FUNDING NUMBERS

N00014-86-K-0245 P00002

6. AUTHOR(S)

R. P. Cook, S. H. Son

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Virginia
Department of Computer Science
Thornton Hall
Charlottesville, VA 22901

8. PERFORMING ORGANIZATION REPORT NUMBER

UVA/525410/CS90/103

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research Resident Representative
818 Connecticut Avenue, N. W. Eighth Floor
Washington, DC 20006

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution unlimited

12b. DISTRIBUTION CODE

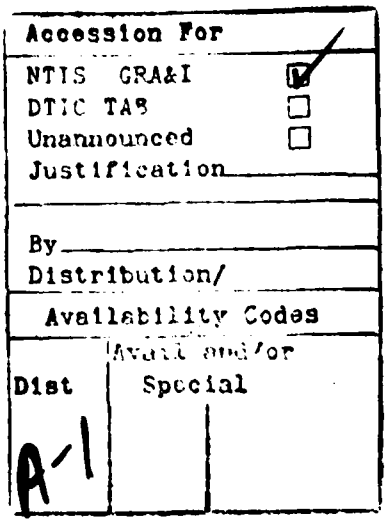
13. ABSTRACT (Maximum 200 words)

The StarLite Project has the goal of constructing a program library for real-time applications. The initial focus of the project is on operating system and database support. The project also involves the construction of a prototyping environment that supports experimentation with concurrent and distributed algorithms in a host environment before down-loading to a target system for performance testing.

The components of the project include a Modula-2 compiler, a symbolic Modula-2 debugger, an interpreter/runtime package, the Pheonix operating system, the meta-file system, a visual simulation package, a database system, and documentation.

14. SUBJECT TERMS			15. NUMBER OF PAGES
StarLite Project, Modula-2 compiler, Modula-2 debugger, Pheonix operating system			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 UL

TABLE OF CONTENTS



PROGRESS REPORT

1. Introduction

It seems improbable that a single database or operating system will suffice to solve all the application problems that are likely to arise in future real-time, embedded systems. A much more likely scenario is that future engineers, with support from a programming environment, will select and adapt modules from program libraries. The selected modules must have proven operating characteristics and the domain over which they are applicable must be well-defined.

The StarLite Project, which is supported by the Office of Naval Research, has the goal of constructing such a program library for real-time applications. The initial focus of the project is on operating system and database support.

Another goal of the StarLite project is to test the hypothesis that a host prototyping environment can be used to significantly accelerate our ability to perform experiments in the areas of operating systems, databases, and network protocols. The primary project requirement for StarLite is that software developed in the prototyping environment must be capable of being retargeted to different architectures only by recompiling and replacing a few low-level modules. The anticipated benefits are fast prototyping times, greater sharing of software in the research community, and the ability for one research group to validate the claims of another by replicating experimental conditions exactly.

As one measure of the effectiveness of the environment, it is often possible to fix errors in the operating system, compile, and reboot the StarLite virtual machine in less than twenty seconds. The compilation time on a SUN 3/280 for the 66 modules (7500 lines) that comprise the operating system is one minute (clock) or 16 seconds (user time). At the present time, all components execute on SUN workstations using the StarLite Modula-2 system.

The StarLite prototyping architecture is designed to support the simultaneous execution of multiple operating systems in a single address space. For example, to prototype a distributed operating system, we might want to initiate a file server and several clients. Each virtual machine would have its own operating system and user processes. All of the code and data for all of the virtual machines would be executed as a single UNIX process.

In order to support this requirement, we assume the existence of high-performance workstations with large local memories. Ideally, we would prefer multi-thread support, but multiprocessor workstations are not yet widely available. We also assume that hardware details can be isolated behind high-level language interfaces to the extent that the majority of a system's software remains invariant when retargeted from the host to a target architecture.

StarLite has matured to the point that we are, in the coming year, pursuing several technology transition possibilities. We will briefly describe several of our ideas. The progress to date for each of the StarLite components is covered in later Sections.

First, we think that it might be possible to convert our whole environment over to an Ada subset. Note the word "subset"; no one in a University could do a full Ada environment with our funding level. The advantage is that we could take advantage of StarLite's portability to provide a direct technology transfer mechanism for industry. Second, we have started work with IBM Manassas to combine our database work with the CMU work on the ARTS kernel. The database group at NOSC is waiting to use the

resulting system as a testbed. Third, we are attempting to convince CLI, which has a DARPA contract to formally verify Kernelized Mach, to adopt our modular design approach so that we could work jointly on the project. This would result in a modular Kernelized Mach running under the StarLite environment. The advantage is that Mach could then be widely distributed as a research vehicle. With more researchers working on making Mach better, DOD and its other users would benefit. Furthermore, the StarLite Mach could serve as a training device for defense contractors who wished to build their own OS on top of the Mach kernel. Fourth, some of the people in the 43RSS development group have agreed to work with us to use their Pulse Detection System specification as a testbed for our operating system and database work. We have already started the implementation. The result will be a real test case that can be widely distributed to other researchers. It will also be a better testbed for our new algorithms than the random parameter ranges that we currently use.

2. Related Activities

- Cook, General Chairman, Seventh IEEE Workshop on Real-Time Software and Operating Systems, Charlottesville, VA (1990).
- Son, participation in the coordination meeting with Prof. Tokuda from CMU and Pat Watson from IBM (Sept. 1989).
- Cook, participant in IDA/ONT/ONR/SNSC Workshop on Operating Systems for Mission Critical Computing (Sept. 1989).
- Cook, invited participant in AIA/SEI Workshop on Research Advances Required for Real-Time Software Systems in the '90s (Sept. 1989).
- Son, presentation at the 18th International Conference on Parallel Processing (Aug. 1989)
- Son, participation in the IEEE Data Engineering Conference program committee meeting (Aug. 1989).
- Son, participation in the ACM SIGMOD Conference (June 1989).
- Son, invited talk at Stanford University on real-time databases (June 1989).
- Cook and Son, accepted for Tools Fair presentation of StarLite at the 11th International Conference on Software Engineering, (May 1989).
- Cook, Session Chair, Sixth IEEE Workshop on Real-Time Software and Operating Systems, (May 1989).
- Cook, invited participant at the IEEE Indialantic Workshop on Tools and Environments for Reuse, (May 1989).
- Son, participation in the Carnegie-Mellon University ARTS project meeting (May 1989).
- Son, presentation at the International Symposium on Database Systems for Advanced Applications (April 1989).
- Son, Session Chair and panelist at the International Symposium on Database Systems for Advanced Applications (April 1989).
- Son, presentation at the IEEE INFOCOM '89 (April 1989).
- Cook and Son, presentation at the ACM Conference on Hypercube Concurrent Computers and Applications (March 1989).
- Son, invited talk at the NSWC on reliable distributed database systems (March 1989).

• Son, participation in the Real-Time Systems Symposium (Dec. 1988).
• Cook and Son, presentation at the ONR Foundations of Real-Time Computing Research Initiative Workshop (Nov. 1988).

3. Student Participation

Chun-Hyon Chang (Post Doc.), priority-based contention protocols
Anthony Burrell (Ph.D. student), real-time operating system scheduling
Shi-Chin Chiang (Ph.D. Student), checkpointing in distributed database systems
Lee Hsu (Ph.D student), just getting started
Ying-Feng Oh (Ph.D. student), just getting started
Juhnyoung Lee (Ph.D. student), just getting started
Jeremiah Ratner (Ph.D. student), synchronization protocols for real-time systems
Ambar Sarkar (Ph.D. student), real-time, fault-tolerant network protocols
David Duckworth (M.S. student), Modula-2 to C compiler
Greg Fife (M.S. student), real-time, distributed, site atomic transactions
Navid Haghighi (M.S. student), multi-version database performance evaluation
Chris Koeritz(M.Sc. student), real-time operating system
Marc Poris (M.S. student), integration of a database with real-time kernel
Paul Shebalin (M.S. student), software safety in real-time systems
Alan Tuten (M.S. student), relational database extension
Prasad Wagle (M.S. student), temporal consistency issues
Richard McDaniel(B.S. student), prototyping environment

4. Publications Since September 1988

• Journal Publications

(1) Cook, R. P., "An Empirical Analysis of the Lilith Instruction Set," *IEEE Transactions on Computers 38*, 1(Jan. 1989) 156-158.

(2) Cook, R.P., "StarMod--A Language for Distributed Programming," reprinted in *Concurrent Programming*, Addison-Wesley, edited by N. Gehani and A.D. McGettrick, (1988).

(3) Son, S. H., "An Adaptive Checkpointing Scheme for Distributed Databases with Mixed Types of Transactions," *IEEE Transactions on Knowledge and Data Engineering*, (Dec. 1989), to appear.

(4) Son, S. H., "An Algorithm for Non-Interfering Checkpoints and its Practicality in Distributed Database Systems," *Information Systems*, (Dec. 1989), to appear.

(5) Son, S. H. and A. Agrawala, "Distributed Checkpointing for Globally Consistent States of Databases," *IEEE Transactions on Software Engineering 15*, 10(Oct. 1989) 1157-1167.

(6) Son, S. H., "Recovery in Main Memory Database Systems for Engineering Design Applications," *Information and Software Technology 31*, 1(March 1989) 85-90.

(7) Son, S. H., "Checkpointing and Recovery in Distributed Database Systems," *Data Engineering 12*, 1(March 1989) 44-50.

(8) Son, S. H., "An Algorithm for Efficient Decentralized Checkpointing," *Journal of Computer Systems Science and Engineering 4*, 1(Jan. 1989) 27-34.

(9) Son, S. H., "Replicated Data Management in Distributed Database Systems," *ACM SIGMOD Record 17*, 4(Dec. 1988) 62-69.

(10) Son, S. H., "Semantic Information and Consistency in Distributed Real-Time Systems," *Information and Software Technology 30*, 3(Sept. 1988) 443-449.

- **Refereed Conference Publications**

(11) Cook, R. P., "The StarLite Operating System," Workshop on Operating Systems for Mission-Critical Computing, (Sept. 1989) J1-J7.

(12) Son, S. H. and N. Haghighi, "Performance Evaluation of Multiversion Database Systems," *Sixth IEEE International Conference on Data Engineering*, Los Angeles, California, (Feb. 1990), to appear.

(13) Son, S. H., "On Priority-Based Synchronization Protocols for Distributed Real-Time Database Systems," *IFAC/IFIP Workshop on Distributed Databases in Real-Time Control* Budapest, Hungary, (Oct. 1989), to appear.

(14) Son, S. H. and Y. Kim, "A Software Prototyping Environment and Its Use in Developing a Multiversion Distributed Database System," *18th International Conference on Parallel Processing*, St. Charles, Illinois, (Aug. 1989) 81-88.

(15) Son, S. H. and R. Cook, "Scheduling and Consistency in Real-Time Database Systems," *Sixth IEEE Workshop on Real-Time Operating Systems and Software*, Pittsburgh, Pennsylvania, (May 1989) 42-45.

(16) Son, S. H. and C. Chang, "Distributed Real-Time Database Systems: Prototyping and Preformance Evaluation," *International Symposium on Database Systems for Advanced Applications*, Seoul, Korea, (April 1989) 251-258.

(17) Son, S. H. and H. Kang, "Approaches to Design of Real-Time Database Systems," *International Symposium on Database Systems for Advanced Applications*, Seoul, Korea, (April 1989) 274-281.

(18) Son, S. H., "A Resilient Replication Method in Distributed Database Systems," *IEEE INFOCOM '89*, Ottawa, Canada, (April 1989) 363-372.

(19) Son, S. H., J. Pfaltz, and J. French, "Synchronization of Replicated Data in Parallel Database Systems," *Fourth ACM Conference on Hypercube Concurrent Computers and Applications*, Monterey, California, (March 1989).

(20) Son, S. H., R. Cook and J. Ratner, "Communication Paradigms for Message-Based Multicomputer Systems," *Fourth ACM Conference on Hypercube Concurrent Computers and Applications*, Monterey, California, (March 1989).

(21) Pfaltz, J., J. French, and S. H. Son, "Parallel Set Operators," *Fourth ACM Conference on Hypercube Concurrent Computers and Applications*, Monterey, California, (March 1989).

- **Technical Reports**

(22) Son, S. H. and J. Ratner, "StarLite: An Environment for Distributed Database Prototyping," *Technical Report TR-89-05*, Dept. of Computer Science, University of Virginia, (Aug. 1989).

(23) Son, S. H. and N. Haghighi, "Performance Evaluation of Multiversion Database Systems," *Technical Report IPC-TR-89-007*, Institute for Parallel Computation, University of Virginia, (July 1989).

(24) Son, S. H. and N. Haghighi, "Multiple Data Versions in Database Systems," *Technical Report TR-89-01*, Dept. of Computer Science, University of Virginia, (June 1989).

5. The Prototyping Environment

The components of the environment include a Modula-2 compiler, a symbolic debugger, a window package, an interpreter/runtime package, the Phoenix operating system, the concurrency control algorithm testbed, a simulation package, and documentation.

During the past year, the windows package was extended to support bit-mapped graphics operations. As a result, we were able to implement a number of support tools for profiling, graphing, and visual simulation. Also, the debugger was rewritten to be window-based and mouse-driven. This also involved changing the compiler so that breakpoints worked correctly.

One of the problems with the environment is the delay introduced by using an interpreter. This problem is being addressed in two ways. First, we performed a static and dynamic analysis of instruction opcode usage as a prerequisite to improving the interpreter's architecture. Secondly, we think that we have found a way to support "mixed" execution; that is, a program that combines interpreted code and native machine code. If our design works, all of the tools will continue to work but users can mix and match machine language modules for significant performance gains. We believe that this goal can be achieved without sacrificing portability.

As the system has grown larger, it has become more difficult to synchronize changes that propagate through multiple modules. To address this problem, we implemented a simple "make" utility that automatically compiles dependent modules. It is simpler to use than UNIX "make" and avoids unnecessary compilations.

In summary, the environment is designed to maximize productivity. Therefore, it accelerates a researcher's ability to conduct experiments, which advances the state-of-the-art. While the initial version of the environment executes as a single UNIX process, future versions could take excellent advantage of both load balancing to distribute a running prototype across a number of machines and of multiprocessor support, such as is found in Mach or Taos.

6. Operating System

During the past year, the operating system implementation was modified to execute on the multiprocessor machine model as well as the distributed nodes. Quite a bit of effort was invested in the efficient use o. spin locks. As a result, we have invented a new method for handling synchronization within the operating system. The new method should decrease the cost of lock overhead dramatically.

We also experimented with techniques to minimize interrupt latency in the operating system. This effort was successful and resulted from the isolation of the use of DISABLE to only two modules.

We also rewrote the SDB relational database system provided to Professor Son by Pat Watson from IBM Manassas. We call our system RDB for Real-time Database. Our version corrects a number of defects in SDB. It is reentrant, can be preempted, supports more flexible query processing, and it has more data types than SDB.

We experimented with a new dynamic binding mechanism for operating system services. The intent is to make it easy for application engineers to adapt the operating system to meet the requirements imposed by hard real-time tasks. For example, they might want a file system without naming to improve performance and predictability.

We experimented with and implemented a Volume Standard Format. The purpose of a VSF is to make it possible for multiple operating systems to share files but without sacrificing their own disk layouts or naming conventions. When VSF is perfected, it will be suitable for VLSI implementation as a national standard candidate.

7. Database Systems

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 c sidered in real-time database systems. Transactions must be scheduled in such a way th... they can be completed before their corresponding deadlines expire. For example, both the update and query operations on the tracking data of a missile must be processed within the given deadlines: otherwise, the information provided could be of little value. State-of-the-art database systems are typically not used in real-time applications due to two inadequacies: poor performance and lack of predictability. Current database systems 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 that are compatible with time-driven scheduling and provide system response predictability need to be investigated.

Our research effort during October 1988 to September 1989 was concentrated in three areas: investigating new techniques for real-time database systems, integrating a relational database system with the real-time operating system kernel ARTS, and developing a message-based database prototyping environment for empirical study. In addition, we have evaluated the performance of real-time database systems developed using the prototyping environment.

7.1. New Approaches

We have investigated two approaches in designing real-time database systems. The first approach is to use advanced database techniques to improve the availability and responsiveness of real-time database systems. Specifically, we have studied techniques for database checkpointing and synchronization using priorities and multiple versions of data. The second approach is to exploit semantic information about transactions and data for intelligent scheduling. This approach, combined with effective use of data replication, may improve responsiveness and reliability.

The need for having checkpoint mechanisms in distributed database systems is well known. Checkpoints are performed in database systems to save a consistent state of the database on a separate secure device. In case of a failure, the stored data can be used to restore the database. Since checkpointing is performed during the normal operation of the system, interference with transaction processing must be kept to a minimum. It is highly desirable that users are allowed to submit transactions while checkpointing is in progress and that transactions are executed in the system concurrently with the checkpointing process. In distributed systems, this non-interference requirement makes checkpointing complicated because we need to consider coordination among autonomous sites of the system. A quick recovery from failure is also desirable in real-time applications of database systems that require high availability. To achieve quick recovery, each checkpoint needs to be globally consistent so that a simple restoration of the latest checkpoint can bring the database to a consistent state. To make each checkpoint globally consistent, updates of a transaction must be either included completely in one checkpoint, or not included at all.

Recently, the possibility of non-interfering checkpointing mechanisms, which do not interfere with transaction processing and achieve global consistency, have been proposed. They are very promising for real-time database systems. We have investigated and extended the use of non-interfering and adaptive checkpointing techniques for distributed real-time database systems. Our research effort has resulted in a feasible solution for achieving the goals of checkpointing. Currently, we are implementing a non-interfering checkpointing algorithm and are using the prototyping environment to evaluate the performance of our solution.

Performance of real-time database systems can be enhanced by synchronization using priorities and multiple versions of data. 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 the 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*.

Priority inversion is said to occur when a high priority process is forced to wait for an indefinite period of time for the execution of a lower priority process to complete. Priority inversion is inevitable in transaction-based systems. However, to achieve a high degree of schedulability in real-time applications, priority inversion must be minimized.

We have implemented priority-based scheduling algorithms in our prototyping environment and investigated technical issues associated with them. One of the issues we studied was the use of the priority ceiling approach as a basis for a real-time locking protocol in a distributed environment. The priority ceiling protocol might be implemented in a distributed environment by using the global ceiling manager at a specific site.

In this approach, all decisions for ceiling blocking are performed by the global ceiling manager. Therefore, all the information for the ceiling protocol is stored at the site of the global ceiling manager. The advantage of this approach is that the temporal consistency of the database is guaranteed since every data object maintains its most up-to-date value. While this approach ensures consistency, holding locks across the network is not very attractive. Due to communication delay, locking across the network will only force the processing of a transaction using local data objects to be delayed until access requests to the remote data objects are granted. This delay for synchronization, combined with the low degree of concurrency due to the strong restrictions of the priority ceiling protocol, is counter-productive in real-time database systems.

An alternative to the global ceiling manager approach is to have replicated copies of data objects. An up-to-date local copy is used as the primary copy, and remote copies are used as the secondary read-only copies. In this approach, we assume a single writer and multiple readers model for distributed data objects. This is a simple model of applications such as distributed tracking in which each radar station maintains its view and makes it available to other sites in the network. Currently, we are investigating the trade-offs between these two approaches for distributed real-time database systems and their performance.

Maintaining multiple versions of data objects is another approach to improve system responsiveness by increasing the degree of concurrency. The objective of using multiple versions is to reduce the conflict probability among transactions and the possibility of rejection of transactions by providing a succession of views of data objects. One of the reasons for rejecting a transaction is that its operations cannot be provided 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 transactions. 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 is able to select an appropriate version to read by flexibly controlling the order of read and write operations. We have investigated several problems that must be solved to effectively use multiple versions of data in real-time applications. For example, selection of old versions for a given read-only transaction must ensure the consistency of the state seen by the transaction. In addition, the need to save old versions for read-only transactions introduces a storage management problem, i.e., methods to determine which version is no longer needed so that it can be discarded.

Since multiversion database systems maintain timing information associated with data objects, they can be used to satisfy temporal requirements of real-time transactions. The temporal consistency requirement is specified in terms of the desired accuracy of the value of data objects to be read by the transaction. Temporal consistency provides a time interval, relative to the start time of a transaction, during which accurate states of data items may be accessed. For example, the temporal consistency requirement of 15 indicates that the data items accessed by the transaction cannot be *older* than 15 time units relative to the start time of the transaction. An attempt to read an inaccurate data item (i.e. one whose write timestamp is outside of this 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. We have developed a real-time transaction model that can be used for multiversion data objects, and are currently investigating the scheduling options for multiversion real-time databases.

7.2. Integration of a Relational Database with ARTS

ARTS is the real-time operating system kernel being developed by the researchers at the Carnegie-Mellon University. The goal of the ARTS OS is to provide a predictable, analyzable, and reliable distributed real-time computing environment. We have been working closely with the ARTS developers and Pat Watson at the IBM Federal Systems Division to integrate a relational database system with ARTS. Our goal is to provide a fully functional distributed relational database manager for real-time systems. At present, a relational database server and client objects are running on top of ARTS. We are investigating methods to selectively apply consistency management techniques and to develop a multi-thread server for this real-time database manager. In addition, we are expanding the functionalities that can be provided by the real-time relational database manager.

7.3. Development of A Database Prototyping Tool

One of the primary reasons for the difficulty in successfully developing and evaluating new techniques for distributed database systems is that it takes a long time to develop a system, and evaluation is complicated because it involves analyzing a large number of system parameters that may change dynamically. Prototyping methods can be applied effectively to the evaluation of new techniques for implementing distributed database systems. By investigating design alternatives and performance/reliability

characteristics of new database techniques, we can provide a clear understanding of design alternatives with their costs and benefits in quantitative measures. Furthermore, database technology can be implemented in a modular reusable form to enhance experimentation. Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation, especially for distributed real-time database systems.

A prototyping tool to implement database technology should be flexible and organized in a modular fashion to provide enhanced experimentation capability. A user should be able to specify system configurations such as the number of sites, network topology, the number and locations of processes, the number and locations of resources, and the interaction among processes. We use the client/server paradigm for process interaction in our prototyping tool. 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 the clients of the system,. where a client can request a service by sending a request message to the corresponding server.

We have enhanced the previous version of the prototyping tool running on a Sun workstation. The current prototyping tool provides concurrent transaction execution facilities, including two-phase locking and timestamp ordering as underlying synchronization mechanisms. A series of experiments have been performed to evaluate the performance of multiversion database systems and priority-based synchronization algorithms. Using the prototyping environment, we found that for specific workload, multiversion database systems offer performance improvements despite the additional CPU and I/O costs involved in accessing the old versions of data. We have also found that transaction size is one of the most critical parameters that affects system performance. Some of our findings have been presented at the International Conference on Parallel Processing (August 1989), and will be presented at the International Conference on Data Engineering (February 1990).

We have implemented the priority-ceiling protocol and compared its performance with other design alternatives such as the two-phase locking protocol. We found that as the transaction size increases, there is little impact on the throughput of priority-ceiling protocol over a range of transaction sizes and over the workload type. Furthermore, the percentage of deadline missing transactions increases sharply for the two-phase locking protocol as the transaction size increases. A sharp rise was expected, since the probability of deadlocks would go up with the fourth power of the transaction size. The percentage of deadline missing transactions increases much more slowly as the transaction size increases in the priority-ceiling protocol, since there is no deadlock in priority-ceiling protocol and the response time is proportional to the transaction size and the priority ranking.

APPENDIX

The StarLite Operating System

Robert P. Cook*
cook@cs.virginia.edu
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
(804) 979-9943

1.0 Introduction

The StarLite project [1,2,3] has four research components in the areas of prototyping, operating systems, database, and computer network technology. The prototyping environment, which executes on Sun workstations, supports the development and execution of software for uni- or multi-processors, as well as distributed systems.

Figure 1 illustrates the use of the prototyping environment during a test session for the StarLite operating system. The figure illustrates our proprietary UNIX* implementation "booting up" on a six-node virtual network. Once the virtual network has booted, the system designer can execute test programs, collect statistics, or examine the system state using the builtin debugger, which is illustrated in Figure 2. We have invested a good deal of effort in building the prototyping system to create what we feel is the best possible research environment.

The purpose of this paper is to describe our approach to designing a new operating system for mission-critical computing and to review some of the technology issues being explored as part of the StarLite project.

*This work is supported by by ONR under contract N00014-86-K0245 and ARO under contract DAAL03-87-K0090.
*UNIX is a registered trademark of AT&T Bell Laboratories.

2.0 Operating System Interfaces

In this Section, we describe the interface requirements that we feel would be most appropriate for a mission-critical operating system solution. Interfaces are important because they can be standardized and because they are designed to outlive implementations and machine architectures.

It is now widely accepted that the use of a procedural interface, such as the C library for UNIX, is the most advantageous method for presenting an operating system's functionality to an end user. Such an interface can be machine and language invariant. These are desirable properties given the diversity of hardware/software used by today's defense contractors.

There are two design options to choose from as the basis for an interface standard: **flat** and **layered**. An operating system with a flat interface, such as UNIX, is essentially closed; that is none of the interfaces used in the implementation can be accessed. Flat interfaces are inflexible and typically trade performance and control for generality.

A layered interface specification, such as the ISO OSI definition for computer networks, overcomes the deficiencies of the traditional, flat

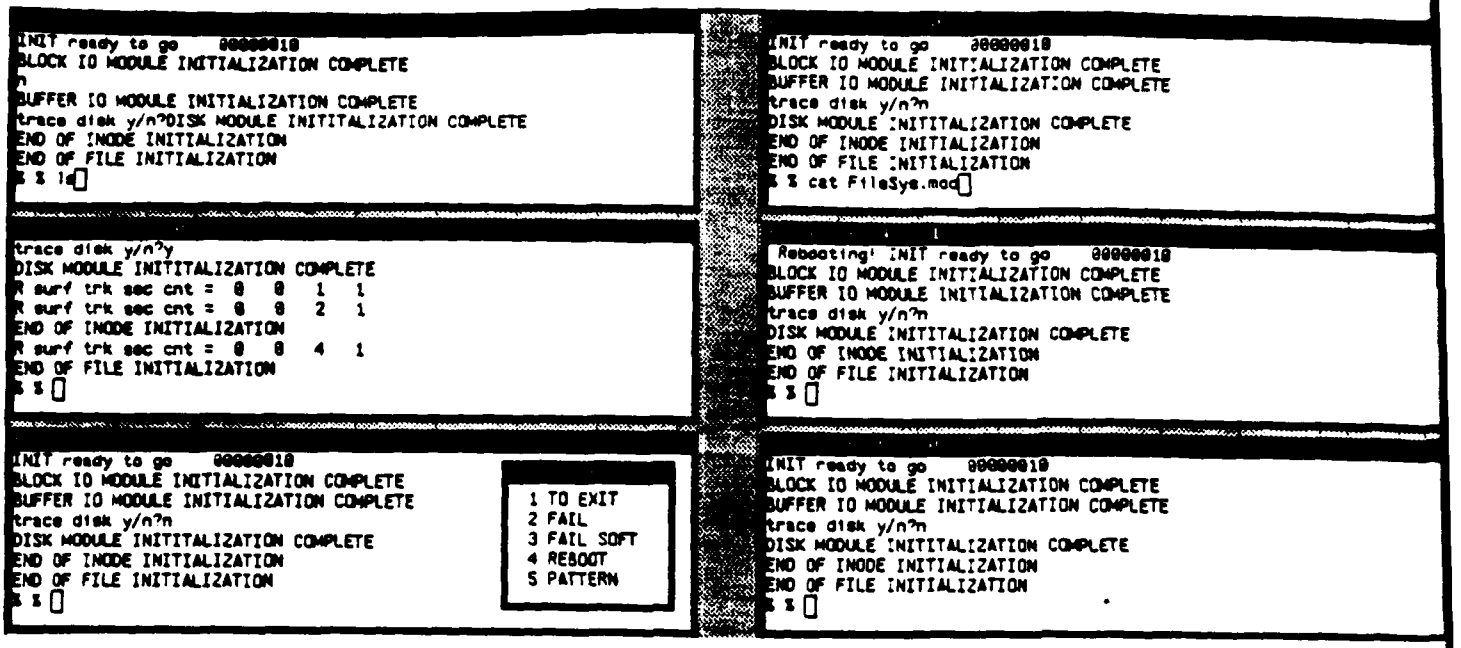


Figure 1. A Six-Node StarLite System

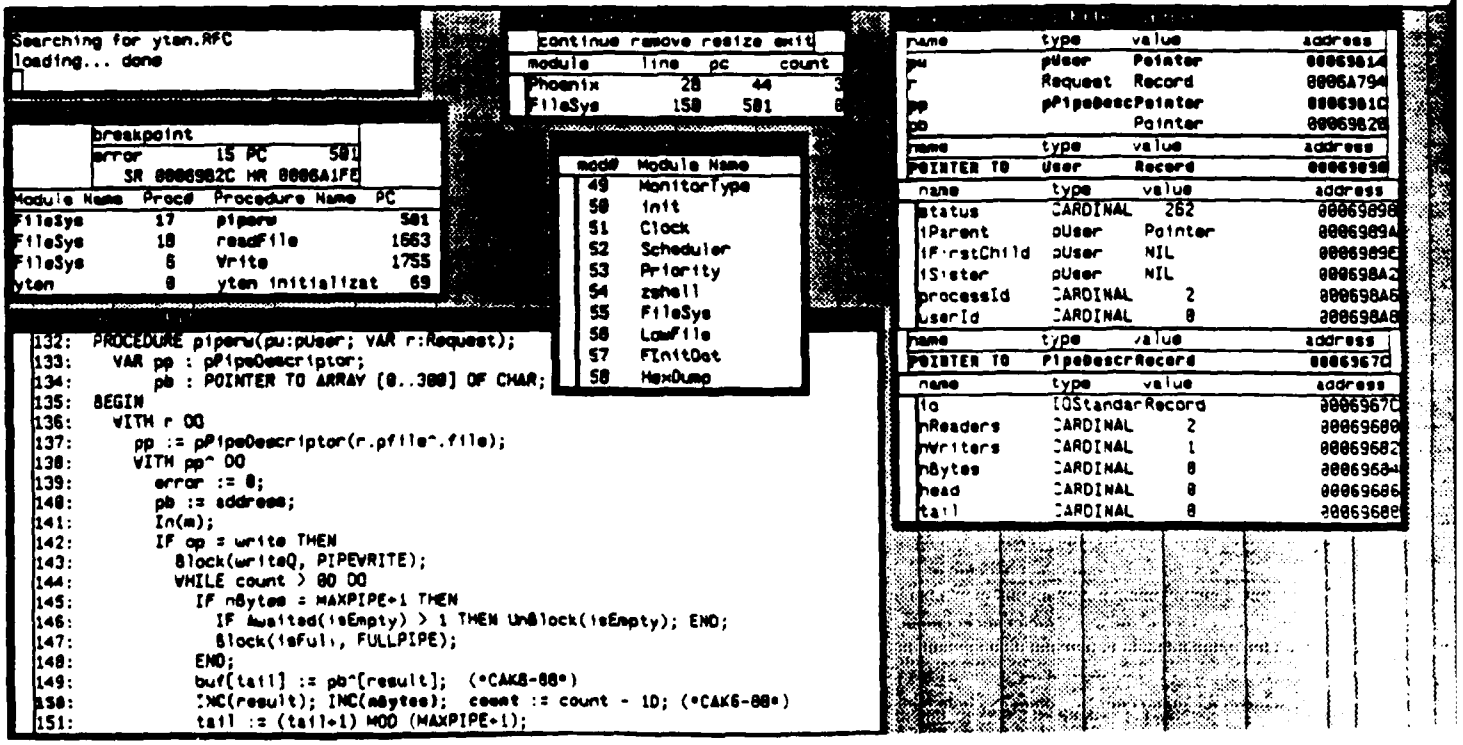


Figure 2. The StarLite Symbolic Debugger

operating system interface designs by allowing the application engineer to choose an interface layer that most closely fits the problem to be solved. For example, if UNIX were a layered design, it would be possible for a database system to manipulate the operating system's buffer cache in a manner that has long been requested by implementors[4].

Access to low-level interfaces can address the performance requirements of mission-critical software. Another advantage of a layered design is that layers can be omitted to save space. For example, if an application does not use files, the file system could be omitted. It is also possible to implement layers in hardware to improve performance.

The StarLite operating system is based on a layered design with standard interfaces. Two of the research issues are how to partition the layers and how to define the interfaces at each layer.

To experiment with different options, we designed and implemented a UNIX-compatible operating system according to the layering principles defined by ISO[5]. The StarLite UNIX is proprietary in that it is not based upon nor does it contain any code from other UNIX implementations. We have rewritten the system several times to try different layering and implementation strategies.

We have found interface specification to be a more demanding task than doing the implementation. In other words, writing a monolithic piece of code to solve a problem is much easier than creating a layered design in which the layers are intended to form functionally complete and useful subsets.

We have found two other problems with a layered design that we are addressing as part of our research. The first problem is the overhead of procedure calls through multiple layers of software. The second problem results from application requirements, such as protection features, which can affect interfaces and implementations at a number of layers. Even if the requirement is removed at a higher layer, there may be unused procedures and data structures at lower layers that affect performance. Both problems are being solved by improving compiler technology.

3.0 Interface Implementations

Most operating system implementations are **closed**; that is, the user cannot and probably should not modify them. The StarLite operating system is designed to support an arbitrary number of different, validated implementations for a given interface. As a result, the operating system as a whole follows an **open** systems architecture that can be tuned to meet application requirements. Examples of different implementation options for the same interface specification include CPU and disk scheduling algorithms or hierarchical versus flat-file name interpretation.

The long-term goal of the StarLite project is to create an operating system generator that could automatically select implementations from a module library based on specified application requirements and a given target architecture. The first step toward achieving this goal is to create a library of implementation modules suitable for mission critical applications. The current phase of the StarLite project is concerned with creating such a library.

4.0 A Software Backplane

One of the prerequisites for experimenting with a library of operating system components is having the ability to add and delete modules or services. Also, we felt that some composition mechanism would be necessary to achieve the goal of creating an operating system generator.

This section discusses the two components of the StarLite operating system that make up what we call a **software backplane**. The two components are a composition strategy for process

objects and a dynamic binding option for system services. The first component is used to create the internal structure of an operating system; the second is used to connect various services to that underlying structure.

In a traditional operating system implementation, such as UNIX, the properties of a process are stored as a single record. Any changes in one module of UNIX require a change in the ".h" file for the shared record. The result is that all the modules in the system must then be recompiled.

The StarLite composition mechanism eliminates unnecessary recompilations by binding properties to processes dynamically. The method is object based but does not support inheritance. Thus, the support code is small and fast.

In StarLite, there is only one class of object, a process. Each process object can be composed of a limited number of properties that can be connected to it in any order and at any time.

When the operating system boots up, each module has been statically linked to the code of the modules that it depends on. However, each module dynamically connects its data type to the process object using a low-level **creat** system call. For example, 'creat(">process/TimingInfo")' would append a set of timing properties of a certain size to every process object. The property fields are created only once when the system boots up. Also at boot time, the modules that use a particular property retrieve the location of its fields with an **open** system call, such as 'open(">process/TimingInfo")'. Again, this only occurs once. Note that the net effect is the same as being able to declare a RECORD structure with the field location bound at runtime.

In order to use the TimingInfo property, a module must execute a **read** system call to retrieve a pointer to or copy of the desired field, depending on the semantics. The contents of the field can then be manipulated with the same efficiency as the traditional UNIX process structure. For example, "p^.nextTimeOut" would retrieve a field from the TimingInfo record.

It is also possible to associate managers with properties. When a process object is closed, the managers are notified one at a time so that the individual fields may be closed. For example, the **exit** system call's implementation is unaware that there is a file system associated with a process. When the manager of the file-system property is invoked as the result of a process exit, it does its own cleanup by closing all open files.

Managers can also be used to monitor the actions on fields for debugging purposes. This is somewhat equivalent to the probe points used on hardware backplanes.

The second component of StarLite's software backplane design is its user services interface. The operating system acts as an agent between user interfaces and modules that provide services. The connection between the two is by means of messages in which the operations requested can be **open**, **share**, **read**, **write**, **rcontrol**, **wcontrol**, and **close**. However, the interpretation of the message is strictly up to the service modules. Thus, the system implementor can create an arbitrary number of user interfaces and an arbitrary number of implementations of those interfaces.

For example, assume that a user **opens** "/dev/pipe". The result is that an action procedure is dynamically associated with the IO field in the user's process object. Next, an open message is constructed and sent to the Pipe module. The return value, which represents two file descriptor tags for the read/write ends of the pipe, is sent to the user's process as a result. The applications engineer can choose from a variety of pipe implementations by using different names. Note that dynamic binding need not entail demand loading; the implementation modules can be loaded with the boot image if desired.

The user services interface has one other aspect, the notion of **context**, that we feel is important for mission critical computing. A context defines the mechanism by which names are interpreted. In the StarLite implementation, all name resolution is accomplished by messages sent to context services by means of action procedure calls.

As a result, any path name syntax and any effect can be realized. For example, the dynamic service binding is implemented by a context module. Contexts can also be used for performance enhancement. For instance, the standard UNIX implementation of path name resolution can result in lengthy and unpredictable disk accesses. Critical read-only file names could be resolved by a context so that their index and data blocks were locked in memory thereby achieving unit access times.

We feel that adaptability and extensibility are desirable properties for operating systems to support mission critical computing. The traditional methods of changing interfaces as new application and technological requirements arise are unacceptable. StarLite achieves flexibility without sacrificing performance.

5.0 Technology Issues

In this section, we discuss some of the StarLite project's research in operating system implementation techniques. The areas discussed include a Volume Storage Format standard (VSF), synchronization, and resource allocation.

5.1 A VSF standard

We feel that one of the key aspects of a support stategy for mission critical computing is a standard format for disk volumes. The advantages are that this standard could be implemented in hardware for high-performance and that files stored on any volume could be accessed by any operating system.

One way to achieve this goal would be to dictate the use of a standard file system for all critical computing. This may not be feasible so we have investigated the lesser goal of standardizing file manipulation, indexing, and disk space allocation. Each vendor's operating system is then presented with a standard interface to a volume.

At the current time, the **VSF** standard is designed to maintain the integrity of a volume's bit map, file descriptors and index blocks. It is up to each operating system to maintain the consistency of other information, which may be arbitrary. For example, UNIX information, such as access times or an owner's id, could be manipulated freely through the interface. Each operating system is free to add whatever information that it wants to either file descriptors or index blocks.

This flexibility is achieved by partitioning the descriptor and index blocks into two parts. One part can be manipulated arbitrarily by the host operating system through a protected interface. The second part can only be used in certain restricted, but always safe, ways. The integrity of the protected information, which contains disk block addresses, guarantees the integrity and recoverability of a volume's data.

The protected part of an index block or file descriptor contains index slots. Each index slot can identify an extent, which can be as small as one block, or another index block. For high performance applications, each file can be implemented as a single extent consisting of a file descriptor followed by the data. This organization avoids the overhead associated with the traditional UNIX implementation.

The design also supports the creation of multilevel index structures. Since an operating system can store into the unprotected part of an index block, it is possible to efficiently implement keyed access methods, such as B-trees, that do not "fit" into the UNIX filesystem model. Although we have not tried it yet, it is also possible to create indices that span multiple files.

The proposed standard is flexible, supports volume interchange, and can be used to achieve predictable, high-performance operation. Proprietary file systems can still be defined, but low-level access to data across systems is guaranteed.

5.2 Synchronization

The StarLite operating system is implemented using the hierarchy of synchronization abstractions listed in Figure 3. Operators lower in the hierarchy have higher performance but have undesirable side-effects associated with their use. Disabling interrupts to protect critical sections is fast (usually one machine instruction) but its indiscriminate use can increase interrupt latency times, which in turn can affect critical event response times. The technique is also inappropriate for multiprocessors where disabling interrupts on one processor has no effect on the execution of the others. The use of DISABLE in StarLite is restricted to two standard modules plus any device drivers that implement device synchronous operations.

As a result, StarLite minimizes interrupt latency. Furthermore, the fine granularity of locking supports kernel preemption as well as simultaneous system or IO operations.

Synchronization Operation	Level
DISABLE/RESTORE	1
Spin Locks	1
Semaphores	2
Monitors	3
Blocking	4

Figure 3. Synchronization Operations

At the higher layers of the StarLite implementation, Semaphores are used in protect critical sections that consist of straight-line code; Monitors are used for critical sections with blocking conditions; and Blocking operations are used for the cases in which a delayed thread can be swapped out. For swapped, blocked threads, the unblock operation is reflected as a state change that defers the wakeup signal until the process is swapped in and scheduled to run.

In addition to experimenting with fine-grained locking and synchronization techniques for operating system construction, we are also investigating the enhancements necessary to support real-time. Two areas of interest are priority inheritance schemes and an integrated view of criticality.

5.3 Resource allocation

Management of resources is one of the most difficult problems to solve in order to produce a full-function UNIX operating system that is capable of providing hard, real-time guarantees.

The problem occurs when a low-priority process holds a resource requested by a high-priority process. If the resource cannot be preempted or released quickly enough, the high-priority process can miss its deadline. The second part of the real-time guarantee problem is to make system timings predictable in the absence of resource contention.

The current StarLite implementation attempts to guarantee that the highest-priority process executes in an interference-free manner as long as its resources are disjoint from other processes. For example, disk writes would circumvent disk scheduling and would supercede other requests.

Our approach to the resource contention problem is based on priority-ordered avoidance[6]. This technique requires that tasks with "hard" deadlines submit claims describing future actions and timing requirements. The system then guarantees that the deadline will be met as long as the task does not exceed its computation and resource limits and neither the hardware nor

software fail.

Each process with "hard" deadlines must submit a claim list identifying the resources to be used and the timing requirements. The system then associates a data structure with each resource that restricts access by competing processes during critical periods. The key to success is making the avoidance test fast enough, which is achieved by using priority to totally order the necessary comparisons.

6.0 Summary

StarLite is a research project that is exploring new ideas for operating system structuring, interface design, analysis, and implementation. It is one of the few projects that provides a standard UNIX interface together with an implementation strategy that addresses the critical system needs of high-performance, openess, and predictability. Its layered interface design and software backplane implementation strategy make the StarLite system unique. Furthermore, its distribution as part of the StarLite programming environment means that any researcher with a SUN workstation can work with the StarLite design to make it better.

References

[1] Cook, R.P., StarLite, A Visual Simulation Package for Software Prototyping, **Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments**, (Dec. 1986) 102-110, also **SIGPLAN Notices 22**, 1(Jan. 1987).

[2] Cook, R.P., StarMod, A Language for Distributed Programming, reprinted in **Concurrent Programming**, Addison-Wesley, edited by N. Gehani and A.D. McGettrick, (1988).

[3] Son, S.H. and R.P. Cook, Scheduling and Consistency in Real-Time Database Systems, **Sixth IEEE Workshop on Real-Time Software and Operating Systems**, (May 1989) 42-45.

[4] Gray, J.N., Notes on Database Operating Systems, in **Operating Systems--An Advanced Course**, edited by Bayer, Graham, Seegmuller , (1979).

[5] Zimmermann, H., OSI Reference Model--The ISO Model of Architecture for Open Systems Interconnection, **IEEE Transactions on Communications COM-28**, (April 1980) 425-432.

[6] Munch-Anderson, B. and T.U. Zahle, Scheduling According to Job Priority With Prevention of Deadlock and Permanent Blocking, **Acta Informatica 6**, 3(1977) 153-175.

RDB,
An Open, Real-Time, Relational Database Kernel

Robert P. Cook* and Sang H. Son
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

1.0 Introduction

In this paper, we discuss the attributes of RDB, which is an "open", real-time, relational database kernel. RDB was implemented using the StarLite[1] software development environment. It was inspired by the SDB system[2] created by Betz and Smith.

RDB is intended for use in embedded systems with requirements for high-performance and real-time priority and predictability guarantees. RDB is a tool that can be used to achieve these goals but it is the user's responsibility to use it properly. For example, RDB is completely reentrant and can be preempted in one context switch time to perform an action for a high priority process. Thus, a query can be interrupted for an update action and then restarted. However, if the low priority process holds locks that the high priority process needs (priority inversion), it is the user's responsibility to resolve the difficulty.

RDB is an "open"[3] system. It is implemented as a hierarchy of modules that are structured so that they can be easily modified or replaced. Furthermore, RDB does not depend on any operating system services. As a result, it is possible to manipulate ROM or memory-resident databases without having to depend on the access methods traditionally provided by operating systems. As a final point, RDB uses upcalls[4] to implement late binding of query and I/O operations.

By providing the user of RDB fine-grained control over its operation, it is simple to select implementation strategies that achieve performance and predictability goals. This can be contrasted with traditional database systems that operate as closed boxes, often with poor performance and predictability characteristics.

The following sections describe the relational model supported by RDB and a simple example that illustrates its use.

2.0 The RDB Model

The RDB kernel supports the following abstract data types: Schema, Relation, Attribute, Cursor, SortKey, SortList, SortLists, Selection, and Expression. Figure 1 illustrates the relationships among the various types.

A Schema in RDB describes the tuple format in terms of the position and type of the Attribute fields. At present, the only field types are text and numeric. The schema is disjoint from the data composing a Relation in order to provide options for real-time systems that are not normally found in traditional database systems. For example, it is possible to define a Relation's content as a file,

*This work is supported by ARO under contract DAAL03-87-K0090 and by ONR under contract N00014-86-K0245.

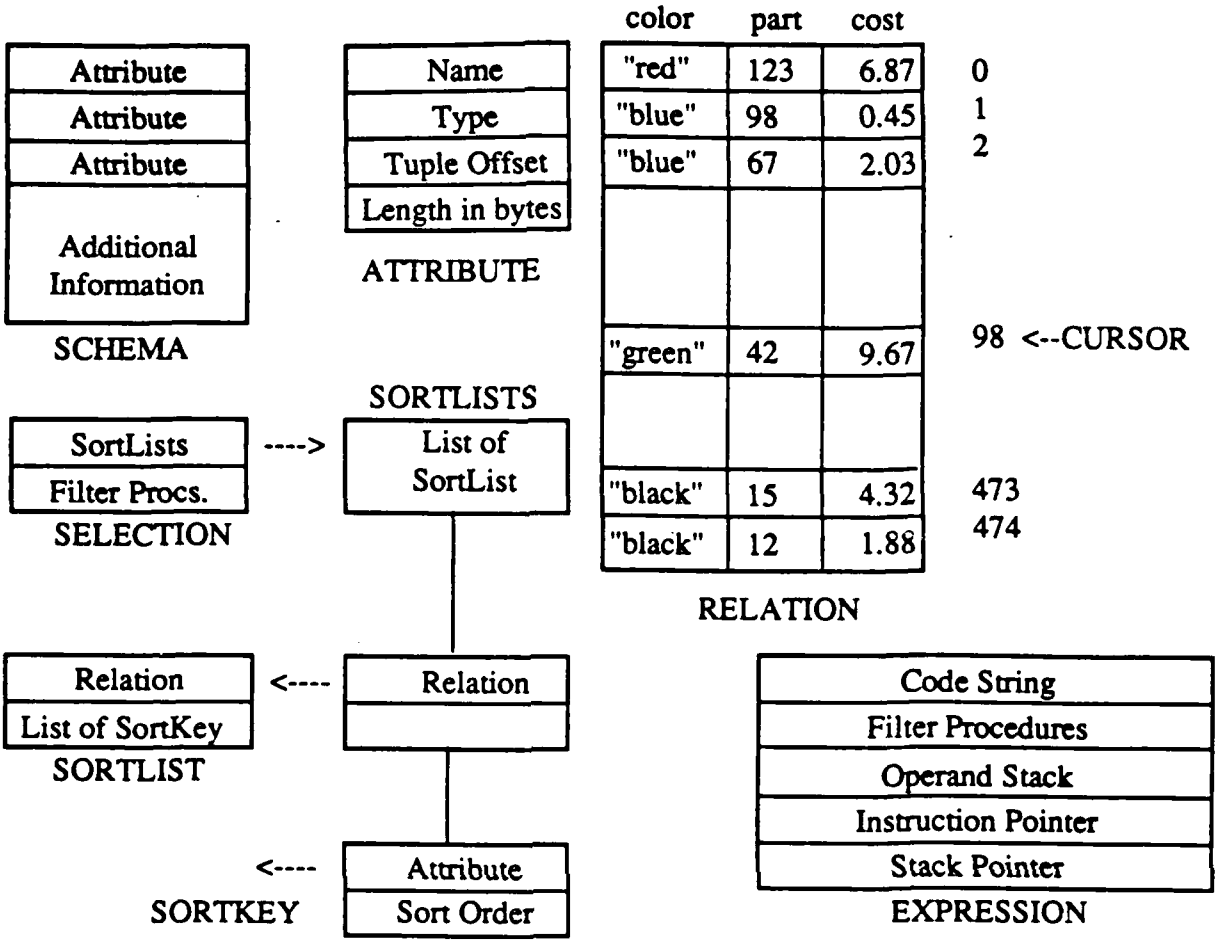


Figure 1. The RDB DataBase Model

but is is also possible to define derived relations. That is, the tuples making up a relation can be computed as requested or they can be generated from a data stream of sensor inputs[5]. Figure 1 illustrates a relation composed from three attributes: color (text), part number (numeric), and cost (numeric). Each Attribute specifies the field name and type as well as its position in the tuple and its length in bytes. The file consists of 475 tuples.

Once a schema has been defined and a relation selected, any number of Cursors can be opened. A cursor identifies a tuple in a relation. Cursors are used to "mark" the positions at which I/O operations are to occur in a database. In RDB, the actual I/O is performed using upcalls. An upcall is an invocation of a procedure variable that is bound after RDB is loaded but before a relation is accessed. The procedures to be invoked are specified when a relation is "connected" to the system for I/O.

Upcalls give the system the flexibility to implement ROM or memory-resident databases, derived or computed relations, and relations based on data streams. In essence, each relation can have a set of access procedures that are tuned to meet system performance and predictability goals. The system provides several traditional access methods, which the user is free to modify or to augment.

The SortKey is a central data structure in most database implementations. It defines the order relation to be used for an attribute when it is accessed. For example, the "part number" attrib-

ute in Figure 1 is sorted in descending numeric order.

A SortList identifies a relation and a list of sort keys. The keys represent the projection of the relation over which a particular operator is to be applied. If an attribute is referenced through a secondary index, the projection can sometimes be loaded by referring to the index rather than reading the tuples of the original relation. For a real-time system, the update costs associated with a secondary index must be compared with its efficiency advantages for query processing.

A SortLists is a list of Sortlist elements, where each SortList is itself a list of SortKeys. The SortLists data type represents the list of projections of relations that participate in multi-relation operations, such as a join.

The join operation is implemented by selection based on one or more expressions. A Selection data structure contains the input SortLists (relations and keys) as well as the upcall procedure variables that are used to filter the tuples in the input relations. Filtering can be applied during selection either when each tuple of a relation is input or when one tuple in each relation has been input.

Tuple filtering can be combined with expression filtering to achieve results that are not possible in a traditional database system. For example, any tuple filter has the ability to terminate selection. As a result, a query that cannot be completed by its original deadline can return a partial, or less accurate result, and still meet its timing constraints.

The Expression data type is implemented in a fashion that makes it orthogonal to the rest of the database kernel. It operates on code strings such as "e0000 s04blue =", which compares field zero in relation zero to the string "blue". If they are equal, the top of the operand stack is set to the Boolean value TRUE.

As with several other components of RDB, the Expression module uses an upcall procedure to bind the interpretatation of the "e" operator (load external). The user typically provides an Expression with a procedure that will return attribute values when presented with the arguments to "load external". However, the Expression module does not know that it is being used by a database system.

As a result, the user of RDB is free to generate the operands of an expression in a manner that is application dependent. For example, a traditional database system would lock out updates to a relation while a query was in progress. Locking can result in priority inversion which is an anathema in real-time systems.

With RDB, the selection filters and expression processing can be specified such that "compensation" is possible. That is, the updates are made to the relation and are simultaneously factored into the query so that neither the query process nor the update process are delayed. In a similar fashion, if records are deleted, the effect may be "subtracted" from a query in progress.

3.0 An RDB Example

The following example illustrates the use of RDB and the Phoenix real-time operating system, which is also part of StarLite, to implement a cyclic process that prints a "parts" report once every hour starting at a particular hour.

Phoenix provides an operator that transforms a procedure into a lightweight thread. Other operators allow a thread to set or change its priority and to delay until a selected time has arrived. Even though a delay operator for a relative amount of time has the same expressive power, we have found that using an absolute time specification results in programs that are more likely to perform as the user intended. This is particularly true for very fine-grained timing control operations.

```
PROCEDURE reportGenerator(initHr:CARDINAL);
BEGIN
```

```
  SetPriority(Self(), 7);
  LOOP
    AtSecond.At(initHr*60);
    IF initHr =24 THEN initHr := 1;
    ELSE initHr := initHr+1;
    END;
    print();
  END;
END reportGenerator;
```

Figure 2. A Cyclic Report Generator

```
PROCEDURE print();
  VAR r : Relation;
      s : SortList;
      input : SortLists;
      sel : Selection;
      e : Expression;
BEGIN
  r := RFind ("partsFile");
  ROpenSort(r, s);
  RAddKey(s, "cost", "ascending");
  RAddKey(s, "color", "");
  ROpenSortLists(input);
  RAddSortList(input, s);
  sel := ROpenSelect(input, e, FIONULL, printr);
  RInterpret.ROpen(e, "e0000 s04blue =", f, sel);
  RSelect(sel);
  RInterpret.Close(e);
  RCloseSelect(sel);
  RCloseSortLists(input);
END print;
```

Figure 3. Initiate Record Selection

In Figure 3, the "print" procedure associates a Relation variable with the database file and schema. Next, the SortLists is constructed to describe the projections of the input relations that are to be printed (in this case just one relation). Notice that the "cost" and "color" keys are permuted from the storage order. In general, a SortList can permute the keys for both the input and output relations.

After the SortLists is initialized, the Selection and Expression data structures are initialized. One of the arguments used to create the "sel" variable is the upcall procedure "printr" that actually produces the report. One of the arguments (procedure "f") to create an expression is an upcall procedure that converts attributes in the input tuples into expression operands. Figure 4 presents an outline of "f" and "printr".

```
PROCEDURE f(relation, attr:CARDINAL
        arg:ADDRESS; VAR (*out*) o:Operand);
  (* set Operand to field "attr" in "relation" *)
  VAR s : Selection;
    pT : pTuple;
    pSK : pSortKey;
BEGIN
  s := arg;   (* remembered by Expression *)
  pSK := RGetSKey(s, relation, attr, pT (*out*) );
  o.pT := pT;
  o.offset := pSK^.offset;
  o.length := pSK^.length;
END f;

PROCEDURE printr(s:Selection; arg:ADDRESS);
  (* filter items to be printed in the report *)
  VAR e : Expression;
    pT : pTuple;
    pO : pOperand;
BEGIN
  e := arg;  (* remembered during selection *)
  pO := RInterpret.Evaluate(e);
  IF NOT pO^.b THEN RETURN; END;
  (* FOR EACH SortList in s DO
    pT := RGetSBuf(s, i);
    Write the selected attribures in pT^
  *)
  InOut.WriteLn;  (* terminate output line *)
END printr;
```

Figure 4. Upcall Procedures

When one tuple has been read for each relation selected as input, the "printr" procedure is invoked. This procedure in turn evaluates an expression to select the tuples to be printed in the report.

Whenever the expression evaluator encounters the "Load External", "e" operator, it performs an upcall to the "f" procedure that was passed as an argument to ROpen to create an Expression variable. One of the arguments to "f" is the address of an operand descriptor. It is the procedure's responsibility to map the relation and attribute indices to a SortKey. The SortKey is

then used to retrieve an attribute value, which is passed back to the evaluator as an operand.

When the evaluator completes, it returns an operand descriptor for the value on the top of the operand stack. For the "printr" procedure, the result is a Boolean value. If it is true, the fields are printed in the report. If it is false, the fields are ignored and selection continues.

RDB implements a number of very flexible options for expression evaluation that space does not permit us to describe. For real-time systems, the two most important are expression preemption and contextual reevaluation.

In the former, any expression can be preempted at any time by more critical expressions or other system actions. Using contextual reevaluation, it is possible for a query to modify its expression while it is being evaluated to return a less accurate result in order to meet timing constraints.

4.0 Summary

The RDB database kernel is intended for use in stand-alone applications that have "hard" timing requirements. It is an "open" system in that the user can manipulate interfaces not normally available in traditional database systems to "tune" performance to application requirements. The use of upcalls also adds great flexibility to both selection and expression processing options.

RDB does not currently support transactions, locking, or recovery. It can, however, operate on either local relations or remote files by using RPC. We are implementing additional functionality as part of a layered design for database operations. The layers are implemented so that the end-user can add or subtract features to meet the performance or timing requirements of embedded systems.

References

[1] Cook, R.P., StarLite, A Visual Simulation Package for Software Prototyping, **Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments**, (Dec. 1986) 102-110, also **SIGPLAN Notices 22**, 1(Jan. 1987).

[2] Betz, D. and D.N. Smith, SDB - A Simple Database System, from documentation provided by Pat Watson at IBM Manassas (Nov. 1988).

[3] Lampson, B.W. and R.F. Sproull, An Open Operating System for a Single-User Machine, **Proc. of the 7th Symposium on Operating System Principles**, (Dec. 1979) 98-105.

[4] Clark, D., The Structuring of Systems Using Upcalls, **Proc. of the 10th Symposium on Operating System Principles**, (Dec. 1985) 171-180.

[5] Snodgrass, R., A Relational Approach to Monitoring Complex Systems, **ACM Transactions on Computer Systems 6**, 2(May 1988) 157-196.

On Priority-Based Synchronization Protocols for Distributed Real-Time Database Systems

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

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. This paper describes a prototyping environment for investigating distributed real-time database systems, and its use for performance evaluation of priority-based scheduling protocols for real-time database systems.

Keywords: distributed database, real-time, prototyping, synchronization, transaction, priority.

1. Introduction

As computers are becoming essential part of real-time systems, *real-time computing* is emerging as an important discipline in computer science and engineering [Shin87]. The growing importance of real-time computing in a large number of applications, such as aerospace and defense systems, industrial automation, and nuclear reactor control, has resulted in an increased research effort in this area. Since any kind of computing needs to access data, methods for designing and implementing database systems that satisfy the requirement of timing constraints in collecting, updating, and retrieving data play an important role in the success of real-time systems.

Researchers working on developing new real-time systems based on distributed system architecture have found out that database managers are assuming much greater importance in real-time systems [Son88]. One of the characteristics of current database managers is that they do not schedule their transactions to meet response requirements and they commonly lock data tables indiscriminately to assure database consistency. Locks and time-driven scheduling are basically incompatible. Low priority transactions can and will block higher priority transactions leading to response requirement failures. New techniques are required to manage database consistency which are compatible with time-driven scheduling and the essential system response predictability/analyzability it brings. One of the primary reasons for the difficulty in successfully developing and evaluating a distributed database system 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 control mechanisms for distributed database systems. A *database prototyping tool* is a software package that supports the investigation of the properties of a database control techniques in an environment other than that of the target database system. The advantages of an environment that provides prototyping tools are obvious. First, it is cost effective. If experiments for a twenty-node distributed database system can be executed in a software environment, it is not necessary to purchase a twenty-node distributed system, reducing the cost of evaluating design alternatives. Second, design alternatives can be evaluated in a uniform environment with the same system parameters, making a fair comparison. Finally, as technology changes, the environment need only be updated to provide researchers with the ability to perform new experiments.

A prototyping environment can reduce the time of evaluating new technologies and design alternatives. From our past experience, we assume that a relatively small portion of a typical database system's code is affected by changes in specific control mechanisms, while the majority of code deals with intrinsic problems, such as file management. Thus, by properly isolating technology-dependent portions of a database system using modular programming techniques, we can implement and evaluate design alternatives very rapidly. Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation. Especially if the system designer must deal with message-passing

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

protocols and timing constraints, it is essential to have an appropriate prototyping environment for success in the design and analysis tasks.

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

2. Structure of the Prototyping Environment

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

Although an integrated approach is desirable, the system needs to support flexibility which may not be possible in an integrated approach. In this regard, the concept of developing a library of modules with different performance and reliability characteristics for an operating system as well as database control functions seems promising. Our prototyping environment follows this approach [Cook87, Son88b]. It is designed as a modular, message-passing system to support easy extensions and modifications. 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.

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

- system configuration: number of sites and the number of server processes at each site.
- database configuration: database at each site with user defined structure, size, granularity, and levels of replication.
- load characteristics: number of transactions to be executed, size of their read-sets and write-sets, transaction types (read-only or update) and their priorities, and the mean interarrival time of transactions.
- concurrency control: locking, timestamp ordering, and priority-based.

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

Transaction execution consists of read and write operations. Each read or write operation is preceded by an access request sent to the Resource Manager, which maintains the local database at each site. Each transaction is assigned to the Transaction Manager (TM). TM issues service requests on behalf of the transaction and reacts appropriately to the request replies.

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

3. Priority-Based Synchronization

In a real-time database system, synchronization protocols must not only maintain the consistency constraints of the database but also satisfy the timing requirements of the transactions accessing the database. To satisfy both the consistency and real-time constraints, there is the 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*. Priority inversion is said to occur when a higher priority process is forced to wait for the execution of a lower priority process for an indefinite period of time. When the transactions of two processes attempt to access the same data object, the access must be serialized to maintain consistency. If the transaction of the higher priority process gains access first, then the proper priority order is maintained; however, if the transaction of the lower priority gains access first and then the higher priority transaction requests access to the data object, this higher priority process will be blocked until the lower priority transaction completes its access to the data object. Priority inversion is inevitable in transaction systems. However, to achieve a high degree of schedulability in real-time applications, priority inversion must be minimized. This is illustrated by the following example.

Example: Suppose T_1, T_2, and T_3 are three transactions arranged in descending order of priority with T_1 having the highest priority. Assume that T_1 and T_3 access the same data object O_i. Suppose that at time t_1 transaction T_3 obtains a lock on O_i. During the execution of T_3, the high priority transaction T_1 arrives, preempts T_3 and later attempts to access the object O_i. Transaction T_1 will be blocked, since O_i is already locked. We would expect that T_1, being the highest priority transaction, will be blocked no longer than the time for transaction T_3 to complete and unlock O_i. However, the duration of blocking may, in fact, be unpredictable. This is because transaction T_3 can be blocked by the intermediate priority transaction T_2 that does not need to access O_i. The blocking of T_3, and hence that of T_1, will continue until T_2 and any other pending intermediate priority level transactions are completed.

The blocking duration in the example above can be arbitrarily long. This situation can be partially remedied if transactions are not allowed to be preempted; however, this solution is only appropriate for very short transactions, because it creates unnecessary blocking. For instance, once a long low priority transaction starts execution, a high priority transaction not requiring access to the same set of data objects may be needlessly blocked.

An approach to this problem, based on the notion of *priority inheritance*, has been proposed [Sha87]. The basic idea of priority inheritance is that when a transaction T of a process blocks higher priority processes, it executes at the highest priority of all the transactions blocked by T. This simple idea of priority inheritance reduces the blocking time of a higher priority transaction. However, this is inadequate because the blocking duration for a transaction, though bounded, can still be substantial due to the potential *chain of blocking*. For instance, suppose that transaction T_1 needs to sequentially access objects O_1 and O_2. Also suppose that T_2 preempts T_3 which has already locked O_2. Then, T_2 locks O_1. Transaction T_1 arrives at this instant and finds that the objects O_1 and O_2 have been respectively locked by the lower priority transactions T_2 and T_3. As a result, T_1 would be blocked for the duration of two transactions, once to wait for T_2 to release O_1 and again to wait for T_3 to release O_2. Thus a chain of blocking can be formed.

Several methods to combat this inadequacy are under investigation. The *priority ceiling protocol* is one of such methods being investigated at the Carnegie-Mellon University [Sha88]. It tries to achieve not only minimizing the blocking time of a transaction to at most one lower priority transaction execution time, but also preventing the formation of deadlocks. The priority ceiling protocol has been implemented in our real-time database system and compared with other synchronization protocols using the prototyping environment.

Using the prototyping tool, we have been evaluating the priority ceiling protocol and investigating technical issues associated with priority-based scheduling protocols. One of the issues we are studying is the comparison of the priority ceiling protocol with other design alternatives. In our experiments, all transactions are assumed to be *hard* in the sense that there will be no value in completing a transaction after its deadline. Transactions that miss the deadline are aborted, and disappear from the system immediately with some abort cost.

4. Priority Ceiling Protocol

The priority ceiling protocol is premised on systems with a fixed priority scheme. The protocol consists of two mechanisms: *priority inheritance* and *priority ceiling*. We already have explained the priority inheritance mechanism. In the priority ceiling mechanism, a priority ceiling is defined for every data object as the priority of the highest priority transaction which may access the data object. A transaction is allowed to access the data object only if its priority is higher than the priority ceilings of all data objects currently being accessed by some transaction in the system. With the combination of these two mechanisms, it has been shown that in the worst case, each transaction has to wait for at most one lower priority transaction in its execution, and no deadlock will ever occur [Sha88]. In the next example, we show 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 enter O_i, regardless of the number of data objects it may access.

The ceiling manager implements the priority ceiling algorithm in the prototyping environment. The lock on a data object can either be a read-lock or a write-lock. 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. When a data object is write-locked (read-locked), the rw-priority ceiling of this data object is defined to be equal to the absolute (write) priority ceiling.

When a transaction attempts to lock a data object, the transaction's priority is compared with the highest rw-priority ceiling of all data items currently locked by other transactions. If the priority of the transaction is not higher than the rw-priority ceiling, it will be denied. Otherwise, it is granted the lock. In the denied case, the priority inheritance is performed in order to overcome the problem of uncontrolled priority inversion.

Under this protocol, it is not necessary to check for the possibility of read-write conflicts. For instance, when a data object is write-locked by a transaction, the rw-priority ceiling is equal to the highest priority transaction that can access it. Hence, the protocol will block a higher priority transaction that may write or read it. On the other hand, when the data object is read-locked, the rw-priority ceiling is equal to the highest priority transaction that may write it. Hence, a transaction that attempts to write it will have a priority no higher than the rw-priority ceiling and will be blocked. Only the transaction that read it and have priority higher than the rw-priority ceiling will be allowed to read-lock it, since read-locks are compatible.

5. 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 which alternates data access requests with equal computation requests, and some processing requirement for termination (either commit or abort). Thus the total processing time of a transaction is directly related to the number of data objects accessed. Due to space considerations, we cannot present all our results but have selected the graphs which best illustrate the difference and performance of the algorithms. For example, we have omitted the results of an experiment that varied the size of the database, and thus the number of conflicts, because they only confirm and not increase the knowledge yielded by other experiments.

For each experiment and for each algorithm tested, we collected performance statistics and averaged over the 10 runs. The percentage of deadline-missing transactions is calculated with the following equation: *%missed* = 100 * (number of deadline-missing transactions / number of transactions processed). A transaction is processed if either it executes completely or it is aborted. We assume that all 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 disappeared 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 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 such peaks occur. In other words, when a crisis occurs and the database system is under pressure is precisely when making a few extra deadlines could be most important [Abb88].

We normalize the transaction throughput in records accessed per second for successful transactions, not in transactions per second, in order to account for the fact that bigger transactions need more database processing. The normalization rate is obtained by multiplying the transaction completion rate (transactions/second) by the transaction size (database records accessed/transaction). In Figure 1, the throughput of the priority-ceiling protocol (C), the two-phase locking protocol with priority mode (P), and the two-phase locking protocol without priority mode (L), is shown for transactions of of different sizes with balanced workload and I/O bound workload.

As the transaction size increases, there is little impact on the throughput of priority-ceiling protocol over the range of transaction sizes and over the workload type shown in Figure 1. This is because in priority-ceiling protocol, the conflict rate is determined by ceiling blocking rather than direct blocking, and the frequency of ceiling blocking is not sensitive to the transaction size.

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

Since 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. This is 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 2, 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 the frequency of ceiling blocking is very sensitive to the duration of data object locking in the system.

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 3, the percentage of deadline missing transactions increases sharply for the two-phase locking protocol as the transaction size increases. A sharp rise was expected, since the probability of deadlocks would go up with the fourth power of the transaction size [Gray81]. However, the percentage of deadline missing transactions increases much slowly as the transaction size increases in the priority-ceiling protocol, since there is no deadlock in priority-ceiling protocol and the response time is proportional to the transaction size and the priority ranking.

6. Conclusions

Prototyping large software systems is not a new approach. However, methodologies for developing a prototyping environment for distributed database systems have not been investigated in depth in spite of its potential benefits. In this paper, we have presented a prototyping environment that has been developed based on a message-based approach with modular building blocks. Although the complexity of a distributed database system makes prototyping difficult, the implementation has proven satisfactory for experimentation of design choices, different database techniques and protocols, and even an integrated evaluation of database systems. It supports a very flexible user interface to allow a wide range of system configurations and workload characteristics. Expressive power and performance evaluation capability of our prototyping environment has been demonstrated by implementing a distributed real-time database system and investigating its performance characteristics.

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

References

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

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

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

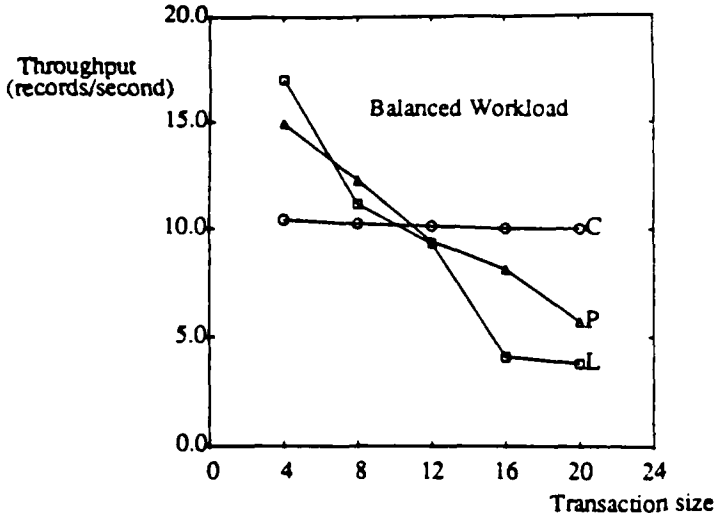
[Sha87] Sha, L., R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocol: An Approach to Real-Time Synchronization," *Technical Report*, Computer Science Dept., Carnegie-Mellon University, 1987.

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

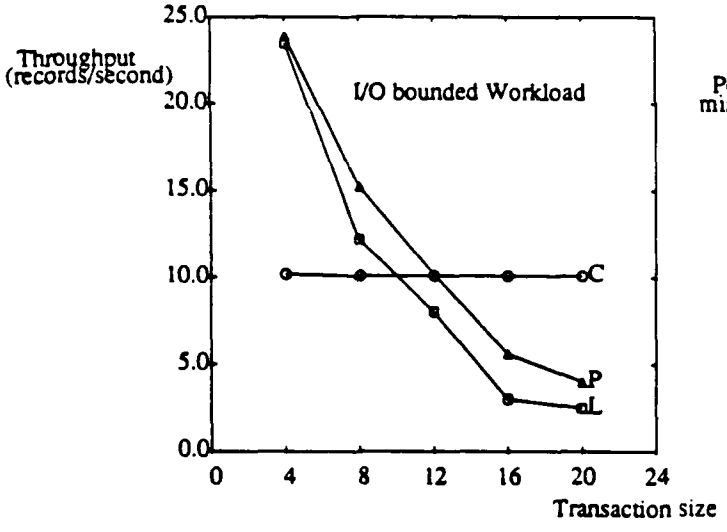
[Shin87] Shin, K. G., "Introduction to the Special Issue on Real-Time Systems," *IEEE Trans. on Computers*, Aug. 1987, 901-902.

[Son88] Son, S. H., "Real-Time Database Systems: Issues and Approaches," *ACM SIGMOD Record 17*, 1, Special Issue on Real-Time Database Systems, March 1988.

[Son88b] Son, S. H., "A Message-Based Approach to Distributed Database Prototyping," *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, May 1988, 71-74.



a) balanced workload transaction



b) I/O bounded workload transaction

Fig. 1 Transaction Throughput.

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

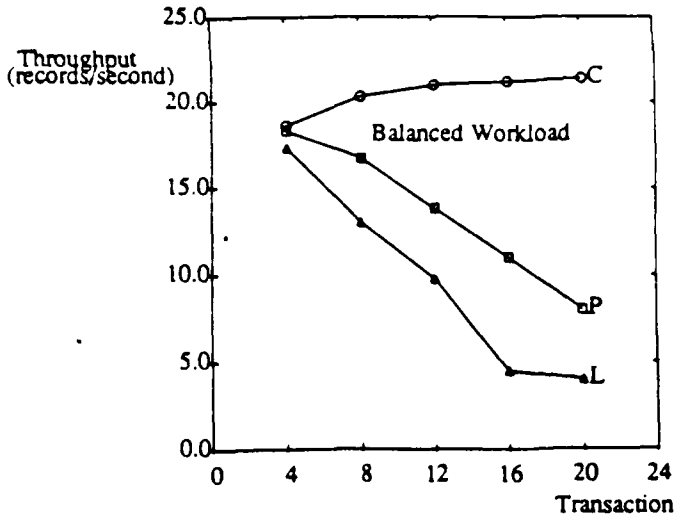


Fig. 2 Transaction Throughput with Intention I/O.

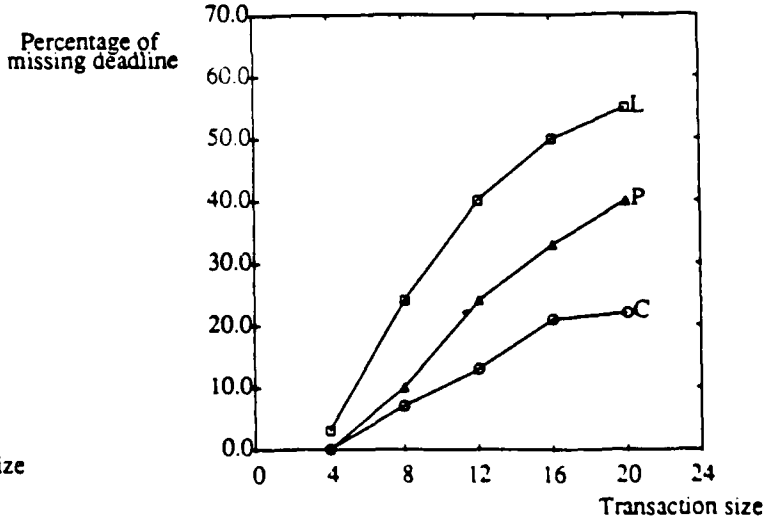


Fig. 3 Percentage of Missing Deadline .

Checkpointing and Recovery in Distributed Database Systems

Sang H. Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

1. Introduction

The need for a recovery mechanism in a database system is well understood. In spite of powerful database integrity checking mechanisms which detect errors and undesirable data, it is possible that some erroneous data may be included in the database. Furthermore, even with a perfect integrity checking mechanism, failures of hardware and/or software at the processing sites may destroy consistency of the database. In order to cope with those errors and failures, database systems provide recovery mechanisms, and checkpointing is a technique frequently used in database recovery mechanisms.

The goal of checkpointing in database systems is to read and return current values of the data objects in the system. A checkpointing procedure would be very useful, if states it returns are guaranteed to be consistent. In a bank database, for example, a checkpoint can be used to audit all of the account balances (or the sum of all account balances). It can also be used for failure detection; if a checkpoint produces an inconsistent system state, one assumes that an error has occurred and takes appropriate recovery measures. In case of a failure, previous checkpoints can be used to restore the database. Checkpointing must be performed so as to minimize both the costs of performing checkpoints and the costs of recovering the database. If the checkpoint intervals are very short, too much time and resources are spent in checkpointing; if these intervals are long, too much time is spent in recovery.

For a checkpoint process to return a meaningful result (e.g., a consistent state), the individual read steps of the checkpoint must not be permitted to interleave with the steps of other transactions; otherwise an inconsistent state can be returned even for a correctly operating system. However, since checkpointing is performed during normal operation of the system, this requirement of non-interference will result in poor performance. For example, in order to generate a commit consistent checkpoint for recovery, user transactions may suffer a long delay waiting for active transactions to complete and the updates to be reflected in the database [CHA85]. A transaction is said to be *reflected* in the database if the values of data objects represent the updates made by the transaction. It is highly desirable that transactions are executed in the system concurrently with the checkpointing process. In distributed systems, the desirable properties of non-interference and global consistency make checkpointing more complicated because we need to consider coordination among autonomous sites of the system.

Recently, the possibility of having a checkpointing mechanism that does not interfere with transaction processing, and yet achieves consistency of the checkpoints, has been studied [CHA85, FIS82, SON86b]. The motivation for non-interfering checkpointing is to improve system availability, that is, the system must be able to execute user transactions concurrently with the checkpointing process. The principle behind non-interfering checkpointing mechanisms is to create a diverged computation of the system such that the checkpointing process can view a consistent state that could result by running to completion all of the transactions that are in progress when the checkpoint begins, instead of viewing a consistent state that actually occurs by suspending further transaction execution. Figure 1 shows a diverged computation during checkpointing.

Non-interfering checkpointing mechanisms, however, may suffer from the fact that the diverged computation needs to be maintained by the system until all of the transactions, that are in progress when the checkpoint begins, come to completion. This may not be a major concern for a database system in which all the transactions are relatively short. However, for database systems with many long-lived transactions, checkpointing of this kind might not

This work was supported in part by the Office of Naval Research under contract number N00014-86-K-0245, by the Department of Energy under contract number DEFG05-88-ER25063, and by the Federal Systems Division of IBM Corporation under University Agreement WF-159679.

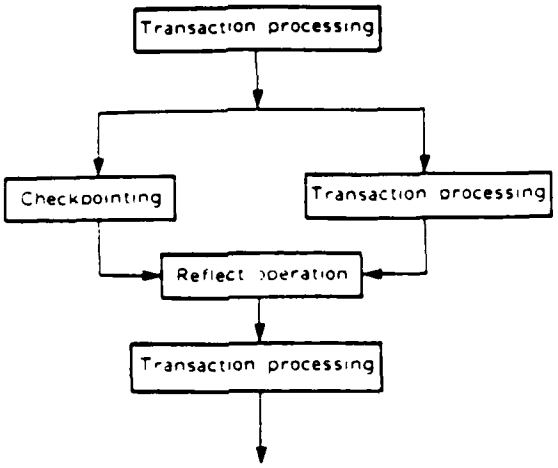


Fig. 1. Diverged computation for checkpointing

be practical for the following reasons:

(1) It takes a long time to complete a non-interfering checkpoint, resulting in high storage and processing overhead.

(2) If a crash occurs before the results of a long-lived transaction are included in the checkpoint, the system must re-execute the transaction from the beginning, wasting all the resources used for the initial execution of the transaction.

In the rest of this paper, we briefly discuss one approach for checkpointing which efficiently generates a consistent database state, and its adaptation for systems with long-lived transactions. Given our space limitations, our objective is to intuitively explain this approach and not to provide details. The details are given in separate papers [SON86b, SON88].

2. Non-interfering Approach

In order to make each checkpoint consistent, updates of a transaction must either be included in the checkpoint completely or not at all. To achieve this, transactions are divided into two groups according to their relations to the current checkpoint: *after-checkpoint transactions* (ACPT) and *before-checkpoint transactions* (BCPT). Updates belonging to BCPT are included in the current checkpoint while those belonging to ACPT are not included. In a centralized database system, it is an easy task to separate transactions for this purpose. However, it is not easy in a distributed environment. To separate transactions in a distributed environment, a special timestamp which is globally agreed upon by the participating sites is used. This special timestamp is called the *Global Checkpoint Number* (GCPN), and it is determined as the maximum of the Local Checkpoint Numbers (LCPN) through coordination of all participating sites.

An ACPT can be reclassified as a BCPT if its timestamp requires that the transaction must be executed before the current checkpoint. This is called the *conversion* of transactions. The updates of a converted transaction are included in the current checkpoint.

Two types of processes are involved in the checkpoint execution: *checkpoint coordinator* (CC) and *checkpoint subordinate* (CS). The checkpoint coordinator starts and terminates the global checkpointing process. Once a checkpoint has started, the coordinator does not issue the next checkpoint request until the first one has terminated. At each site, the checkpoint subordinate performs local checkpointing by a request from the coordinator. We assume that site m has a local clock LC_m which is manipulated by the clock rules of Lamport[LAM78].

Execution of a checkpoint progresses as follows. First, the checkpoint coordinator broadcasts a Checkpoint Request Message with a timestamp LC_{CC}. The local checkpoint number of the coordinator is set to LC_{CC}. The coordinator sets the Boolean variable CONVERT to false, and marks all transactions at the coordinator site with timestamps not greater than $LCPN_{CC}$ as BCPT.

On receiving a Checkpoint Request Message, the local clock of site m is updated and $LCPN_m$ is set to LC_m. The checkpoint subordinate of site m replies to the coordinator with $LCPN_m$, and sets the Boolean variable CONVERT to false. The coordinator broadcasts the GCPN which is determined as the maximum of the local checkpoint numbers.

In all sites, after the LCPN is fixed, all transactions with timestamps greater than the LCPN are marked as temporary ACPTs. If a temporary ACPT updates any data objects, those data objects are copied from the database to the buffer space of the transaction. When a temporary ACPT commits, updated data objects are not stored in the database as usual, but are maintained as *committed temporary versions* (CTV) of the data objects. The data manager in each site maintains permanent and temporary versions of data objects. When a read request is made for a data object which has committed temporary versions, the value of the latest committed temporary version is returned. When a write request is made for a data object which has committed temporary versions, another committed temporary version is created for it rather than overwriting the previous committed temporary version.

When the GCPN is known, each checkpointing process compares the timestamps of the temporary ACPTs with the GCPN. Transactions that satisfy the following condition become BCPTs; their updates are reflected in the database, and are included in the current checkpoint.

$$LCPN < \text{timestamp}(T) \leq GCPN$$

The remaining temporary ACPTs are actual ACPTs; their updates are not included in the current checkpoint. These updates are included in the database after the current checkpointing has been completed. After the conversion of all eligible BCPTs, the checkpointing process sets the Boolean variable CONVERT to true. Local checkpointing is executed by saving the state of data objects when there is no active BCPT and the variable CONVERT is true. After the execution of local checkpointing, the values of the latest committed temporary versions are used to replace the values of data objects in the database. Then, all committed temporary versions are deleted. Execution sequences of two different types of transactions are shown in Figure 2.

As an example, consider a three-site distributed database system. Assume that $LC_{CC} = 5$, $LC_{CS1} = 3$, and $LC_{CS2} = 8$. CC sets its LCPN as 5, and broadcasts a checkpoint request message. On receiving the request message, LCPN of each CS is set to 6 and 9, respectively. After another round of message exchange, the GCPN of the current checkpoint will be set to 9 by the CC and will be known to each CS. If transaction T_i with the timestamp 7 was initiated at the site of CS1, it is treated as an ACPT. All updates by T_i are maintained as CTV. However, when GCPN is known, T_i will be converted to a BCPT and its updates will be included in the current checkpoint.

3. Adaptive Approach for Long-lived Transactions

It can be shown that a non-interfering checkpointing process will terminate in a finite time by selecting an appropriate concurrency control mechanisms [SON87]. However, the amount of time necessary to complete one

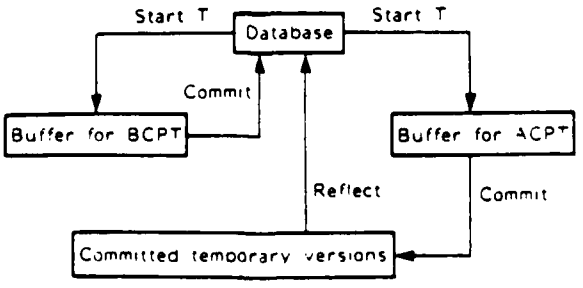


Fig. 2. Execution sequences of ACPT and BCPT

checkpoint cannot be bound in advance; it depends on the execution time of the longest transaction classified as a BCPT. Therefore the storage and processing cost of the checkpointing algorithm may become unacceptably high if a long-lived transaction is included in the set of BCPTs. We briefly discuss the practicality of non-interfering checkpoints in the next section. In addition, all resources used for the execution of a long-lived transaction would be wasted if the transaction must be re-executed from the beginning due to a system failure.

These problems can be solved by using an adaptive checkpointing approach. We assume that each transaction must carry a flag with it, which tells whether it is a normal transaction or a long-lived transaction. The threshold to separate two types of transactions is application-dependent. In general, transactions that need hours of execution can be considered as long-lived transactions.

An adaptive checkpointing procedure operates in two different modes: *global mode* and *local mode*. The global mode of operation is basically the procedure sketched in the previous section. In the local mode of operation, a mechanism is provided to save consistent states of a transaction so that the transaction can resume execution from its most recent checkpoint.

As in the previous approach, the checkpoint coordinator begins checkpointing by sending out Checkpoint Request Messages. Upon receiving this request message, each site checks whether any long-lived transaction is being executed at the site. If so, the site reports it to the coordinator, instead of sending its LCPN. Otherwise (i.e., no long-lived transaction in the system), non-interfering checkpointing begins. If any site reports the existence of a long-lived transaction, the coordinator switches to the local mode of operation, and informs each site to operate in the local mode. The checkpoint coordinator sends Checkpoint Request Messages to each site at an appropriate time interval to initiate the next checkpoint in the global mode. This attempt will succeed if there is no active long-lived transaction in the system.

In the local mode of operation, each long-lived transaction is checkpointed separately from other long-lived transactions. The coordinator of the long-lived transaction initiates the checkpoint by sending Checkpoint Request Messages to its participants. A checkpoint at each site saves the local state of a long-lived transaction. For satisfying the correctness requirement, a set of checkpoints, one per each participating site of a global long-lived transaction, should reflect the consistent state of the transaction. Inconsistent set of checkpoints may result from a non-synchronized execution of associated checkpoints. For example, consider a long-lived transaction T being executed at sites P and Q, and a checkpoint taken at site P at time X, and at site Q at time Y. If a message M is sent from P after X, and received at Q before Y, then the checkpoints would save the reception of M but not the sending of M, resulting in a checkpoint representing an inconsistent state of T.

We use message numbers to achieve consistency in a set of local checkpoints of a long-lived transaction. Messages that are exchanged by participating transaction managers of a long-lived transaction contain message number tags. Transaction managers of a long-lived transaction use monotonically increasing numbers in the tag of its outgoing messages, and each maintains the tag numbers of the latest message it received from other participants. On receiving a checkpoint request, a participant compares the message number attached to the request message with the last tag number it received from the coordinator. The participant replies OK to the coordinator and executes local checkpointing only if the request tag number is not less than the number it has maintained. Otherwise, it reports to the coordinator that the checkpoint cannot be executed with that request message.

If all replies from the participants arrive and are all OK, the coordinator decides to make all local checkpoints permanent. Otherwise, the decision is to discard the current checkpoint, and to initiate a new checkpoint. This decision is delivered to all participants. After a new permanent checkpoint is taken, any previous checkpoints will be discarded at each site.

4. Performance Considerations

There are two performance measures that can be used in discussing the practicality of non-interfering checkpointing: extra storage and extra workload required. The extra storage requirement of the algorithm is simply the CTV file size, which is a function of the expected number of ACPTs of the site, the number of data objects updated by a typical transaction, and the size of the basic unit of information:

$$\text{CTV file size} = N_A \times (\text{number of updates}) \times (\text{size of the data object})$$

where N_A is the expected number of ACPT of the site.

The CTV file may become unacceptably large if N_A or the number of updates becomes very large. Unfortunately, they are determined dynamically from the characteristics of transactions submitted to the database system, and hence cannot be controlled. Since N_A is proportional to the execution time of the longest BCPT at the site, it

would become unacceptably large if a long-lived transaction is being executed when a checkpoint begins at the site. The only parameter we can change in order to reduce the CTV file size is the granularity of a data object. The size of the CTV file can be minimized if we minimize the size of the data object. By doing so, however, the overhead of normal transaction processing (e.g., locking and unlocking, deadlock detection, etc) will be increased. Also, there is a trade-off between the degree of concurrency and the lock granularity[RIE79]. Therefore the granularity of a data object should be determined carefully by considering all such trade-offs, and we cannot minimize the size of the CTV file by simply minimizing the data object granularity.

There is no extra storage requirement in intrusive checkpointing mechanisms[DAD80, KUS82, SCH80]. However this property is balanced by the cases in which the system must block the execution of an ACPT or abort transactions because of the checkpointing process.

The extra workload imposed by the algorithm mainly consists of the workload for (1) determining the GCPN, (2) committing ACPT (move data objects to the CTV file), (3) reflecting the CTV file (move committed temporary versions from the CTV file to the database), and (4) clearing the CTV file when the reflect operation is finished. Among these, the workload for (2) and (3) dominates the total extra workload. As in the estimation of extra storage, the workload for (2) and (3) is determined by the number of ACPTs and the number of updates. Therefore, as long as the values of these variables can be maintained below a certain threshold level, non-interfering checkpointing would not severely degrade the performance of the system. A detailed discussion of the practicality of non-interfering checkpointing is given in [SON86b].

5. Site Failures

So far, we assumed that no failure occurs during checkpointing. This assumption can be justified if the probability of failures during a single checkpoint is extremely small. However, it is not always the case, and we now consider the method to make the algorithm resilient to failures.

During the global mode of operation, the checkpointing process is insensitive to failures of subordinates. If a subordinate fails before the broadcast of a Checkpoint Request Message, it is excluded from the next checkpoint. If a subordinate does not send its LCPN to the coordinator, it is excluded from the current checkpoint. When the site recovers, the recovery manager of the site must determine the GCPN of the latest checkpoint. After receiving information about transactions which must be executed for recovery, the recovery manager brings the database up to date by executing all transactions whose timestamps are not greater than the latest GCPN. Other transactions are executed after the state of the data objects at the site is saved by the checkpointing process.

An atomic commit protocol guarantees that a transaction is aborted if any participant fails before it sends a Precommit message to the coordinator. Therefore, site failures during the execution of the algorithm cannot affect the consistency of checkpoints because each checkpoint reflects only the updates of committed BCPTs.

In the local mode of operation, the failure of a participant prevents the coordinator from receiving OKs from all participants, or prevents the participants from receiving the decision message from the coordinator. However, because a transaction is aborted by an atomic commit protocol, it is not necessary to make checkpointing robust to failures of participants.

The algorithm is, however, sensitive to failures of the coordinator. In particular, if the coordinator crashes during the first phase of the global mode of operation (i.e., before the GCPN message is sent to subordinates), every transaction becomes an ACPT, requiring too much storage for committed temporary versions.

One possible solution to this involves the use of a number of *backup* processes; these are processes that can assume responsibility for completing the coordinator's activity in the event of its failure. These backup processes are in fact checkpointing subordinates. If the coordinator fails before it broadcasts the GCPN message, one of the backups takes control. A similar mechanism is used in SDD-1 [HAM80] for reliable commitment of transactions.

6. Recovery

A recovery from site crashes is called a *site recovery*. The complexity of a site recovery varies in distributed database systems according to the failure situation[SCH80]. If the crashed site has no replicated data objects and if all recovery information is available at the crashed site, local recovery is sufficient. Global recovery is necessary because of failures which require the global database to be restored to some earlier consistent state. For instance, if the transaction log is partially destroyed at the crashed site, local recovery cannot be executed to completion.

When a global recovery is required, the database system has two alternatives: a *fast* recovery and a *complete* recovery. A fast recovery is a simple restoration of the latest global checkpoint. Since each checkpoint is globally

consistent, the restored state of the database is assured to be consistent. However, all transactions committed during the time interval from the latest checkpoint to the time of crash would be lost. A complete recovery is performed to restore as many transactions that can be redone as possible. The trade-offs between the two recovery methods are the recovery time and the number of transactions saved by the recovery.

Quick recovery from failures is critical for some applications of distributed database systems which require high availability (e.g., ballistic missile defense or air traffic control). For those applications, the fate of the mission, or even the lives of human beings, may depend on the correct values of the data and the accessibility to it. Availability of a consistent state is of primary concern for those applications, not the most up-to-date consistent state. If a simple restoration of the latest checkpoint could bring the database to a consistent state, it may not be worthwhile to spend time in recovery by executing a complete recovery to recover some of the transactions.

For the applications in which each committed transaction is so important that the most up-to-date consistent state of the database is highly desirable, or if the checkpoint intervals are large such that a lot of transactions cannot be recovered by a fast recovery, a complete recovery is appropriate. The cost of a complete recovery is the increased recovery time which reduces availability of the database. Searching through the transaction log is necessary for a complete recovery. The property that each checkpoint reflects all updates of transactions with earlier timestamps than its GCPN is useful in reducing the amount of searching, because the set of transactions whose updates must be redone can be determined by a simple comparison of the timestamps of transactions with the GCPN of the checkpoint. Complete recovery mechanisms based on the special timestamp of checkpoints (e.g., GCPN) have been proposed in [KUS82, SON86a].

After site recovery is completed using either a fast recovery procedure or a complete recovery procedure, the recovering site checks whether it has completed local-mode checkpointing for any long-lived transactions. If any local-mode checkpoint is found, those transactions can be restarted from the saved checkpoints. In this case, the coordinator of the transaction requests all participants to restart from their checkpoints if and only if they all are able to restart from that checkpoint. The coordinator decides whether to restart the transaction from the checkpoint or from the beginning based on responses from the participants, and sends the decision message to all participants. Such a two-phase recovery protocol is necessary to maintain consistency of the database in case of damaged checkpoints at the failure site. A transaction will be restarted from the beginning if any participant is not able to restore the checkpointed state of the transaction for any reason.

7. Concluding Remarks

During normal operation, checkpointing is performed to save information for recovery from failure. For better recoverability and availability of distributed databases, checkpointing must allow construction of a globally consistent database state without interfering with transaction processing. Site autonomy in distributed database systems makes checkpointing more complicated than in centralized systems.

The role of the checkpointing coordinator is simply that of getting a uniformly agreed GCPN. Apart from this function the coordinator is not essential to the operation of the proposed algorithm. If a uniformly agreed GCPN can be made known to individual sites, then the centralized nature of the coordinator can be eliminated. One way to achieve this is to preassign the clock values at which checkpoints will be taken. For example, we may take checkpoints at clock values as a multiple of 1000. Whenever the local clock of a site crosses a multiple of this value, checkpointing can begin.

If the frequency of checkpointing is related to load conditions and not necessarily to clock values, then the preassigned GCPN will not work as well. In this case a node will have to assume the role of the checkpointing coordinator to initiate the checkpoint. A unique node has to be identified as the coordinator. This may be achieved by using solutions to the mutual exclusion problem[RIC81] and making the selection of the coordinator a critical section activity.

The properties of global consistency and non-interference of checkpointing results in some overhead and reduces the processing time of transactions during checkpointing. For applications where continuous processing is so essential that the blocking of transaction processing for checkpointing is not feasible, we believe that a non-interfering approach provides a practical solution to the problem of checkpointing and recovery in distributed database systems.

Acknowledgement

The author would like to thank Dr. Won Kim and Professor Robert Cook for their valuable suggestions and comments on the previous version of this paper.

REFERENCES

[CHA85] Chandy, K. M., Lamport, L., Distributed Snapshots: Determining Global States of Distributed Systems, ACM Trans. on Computer Systems, February 1985, pp 63-75.

[DAD80] Dadam, P. and Schlageter, G., Recovery in Distributed Databases Based on Non-synchronized Local Checkpoints, Information Processing 80, North-Holland Publishing Company, Amsterdam, 1980, pp 457-462.

[FIS82] Fischer, M. J., Griffeth, N. D. and Lynch, N. A., Global States of a Distributed System, IEEE Trans. on Software Engineering, May 1982, pp 198-202.

[HAM80] Hammer, M. and Shipman, D., Reliability Mechanisms for SDD-1: A System for Distributed Databases, ACM Trans. on Database Systems, December 1980, pp 431-466.

[KUS82] Kuss, H., On Totally Ordering Checkpoints in Distributed Databases, Proc. ACM SIGMOD, 1982, pp 293-302.

[LAM78] Lamport, L., Time, Clocks and Ordering of Events in Distributed Systems, Commun. ACM, July 1978, pp 558-565.

[RIC81] Ricart, G. and Agrawala, A., An Optimal Algorithm for Mutual Exclusion in Computer Networks, Commun. of ACM, Jan. 1981, pp 9-17.

[RIE79] Ries, D., The Effect of Concurrency Control on The Performance of A Distributed Data Management System, 4th Berkeley Conference on Distributed Data Management and Computer Networks, Aug. 1979, pp 221-234.

[SCH80] Schlageter, G. and Dadam, P., Reconstruction of Consistent Global States in Distributed Databases, International Symposium on Distributed Databases, North-Holland Publishing Company, INRIA, 1980, pp 191-200.

[SON86a] Son, S. H. and Agrawala, A., An Algorithm for Database Reconstruction in Distributed Environments, 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 1986, pp 532-539.

[SON86b] Son, S. H. and Agrawala, A., Practicality of Non-Interfering Checkpoints in Distributed Database Systems, Proceedings of IEEE Real-Time Systems Symposium, New Orleans, Louisiana, December 1986, pp 234-241.

[SON87] Son, S. H., "Synchronization of Replicated Data in Distributed Systems," *Information Systems 12*, 2, June 1987, pp 191-202.

[SON88] Son, S. H., An Adaptive Checkpointing Scheme for Distributed Databases with Mixed Types of Transactions, Proceedings of Fourth International Conference on Data Engineering, Los Angeles, February 1988, pp 528-535.

DISTRIBUTION LIST

1 - 6	Director Naval Research Laboratory Washington, DC 20375 Attention: Code 2627
7 - 18	Defense Technical Information Center, S47031 Building 5, Cameron Station Alexandria, VA 22314
19	Dr. James G. Smith, Program Manager Division of Applied Math and Computer Science Code 1211 Office of Naval Research 800 N. Quincy Street Arlington, VA 22217-5000
20 - 21	R. P. Cook, CS
22	S. H. Son, CS
23	A. K. Jones, CS
24 - 25	E. H. Pancake, Clark Hall
26	SEAS Preaward Administration Files
27	Mr. Michael McCracken Administrative Contracting Officer Office of Naval Research Resident Representative 818 Connecticut Avenue Eighth Floor Washington, DC 20006

JO#2739:ph