SMOS: A MEMORY-RESIDENT OBJECT STORE

BY

YONG YUAN

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

1997

ABSTRACT

In a complex real-time environment, there is often a large amount of temporal data that needs to be promptly collected and made available for processing. To support time constrained access to this data, real-time databases are needed in order to provide predictable transaction time. However, the time constraints can be very tight, requiring a very high database performance level that a conventional database system usually cannot provide.

In this thesis, we present a solution for the need of high-performance databases through the design of a memory-resident object repository called SMOS (Shared Memory Object Store). SMOS features an unique client-only architecture in which the server processes and their communications with the clients are eliminated through System V shared memory support. By providing an object model that fully utilizes the type system of the database application programming language, we alternatively unified multiple address spaces into a single address space in SMOS and thus eliminated object moving and format translations. Further performance gain in SMOS was achieved by relaxing some of the transaction ACID properties that are unnecessary in a high speed real-time environment.

SMOS has been implemented with Open OODB from Texas Instruments, Inc., it was initially developed on an Intel x86-based platform running Solaris 2.5 and has been ported to Sun SPARC running Solaris 2.5. The SMOS/Open OODB prototype has demonstrated superior system performance through various timing experiments.

ACKNOWLEDGMENTS

It has been my good fortune to have Dr. Victor Fay Wolfe serve as my advisor while working on my thesis. He provided me such a great opportunity to work on this challenging thesis project. This thesis would not be possible without his vision, guidance, encouragement, and support.

I would like to thank the other members of my thesis committee: Dr. Joan Peckham, Dr. Yin Sun, and Dr. John W. King for their suggestions, time, and interest.

I am also grateful to Mike Squadrito, John Black, Dr. Lisa Cingiser, Dr. Janet Prichard, and other members of the real-time research group at the University of Rhode Island; discussing research ideas with them have enriched my knowledge.

Finally, I wish to thank my parents and my wife Jiongdong Pang for their support and patience through this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Motivation

Many real-world computing systems are associated with time constraints. These time constraints require that the computations must complete before their deadlines, otherwise various degrees of damage may occur. Such systems are called real-time systems. Typical real-time systems are often seen in the military command and control, nuclear power plants, automatic manufacturing factories, and air traffic control systems. In these environments, real-time systems often have to deal with large volumes of temporal data that can be better managed by database systems. This introduces the need for real-time database systems (RTDB).

A real-time database system combines the features from both real-time systems and database systems; it not only has to satisfy the time constraints required by a real-time system but also has to maintain the data consistency required by a database system. These requirements introduce the major difficulties in the design of a real-time database, because the two fundamental requirements are not compatible: real-time requires the transaction to be performed in a timely fashion with predictability; whereas a database often has to suspend transactions to provide data consistency, making transaction time unpredictable. In (Ramamritham, 1993) several other sources of unpredictability were identified in a conventional database system, including transaction aborts and the resulting rollbacks or restarts, data and resource conflicts, dynamic paging, and I/O.

In the past, research on real-time databases have always been focused on how to make the transactions predictable, in particular it has focused on the concurrency control protocols that are suitable for real-time databases. On the other hand, the degree of the predictability[1] that a real-time database can provide is often ignored. As a result, the system performance level, though fundamental to system predictability, is often not considered as the most important issue, because it is usually assumed that the performance level is sufficient without special treatment. However, in industries, military and other practical environments, the transaction rates can be very high, the data volume per transaction can be very large, and the time constraints associated with the data can be very tight, a guaranteed high system performance level becomes essential for providing high degrees of predictability.

One effective way to increase the system performance level is to keep the database in memory. However, a conventionally designed memory-resident database (MMDB) is still inadequate to satisfy those high demand environments, the best reported simple transaction time we found was 69 milliseconds (Lehman, 1992). Instead of using database systems, those environments are forced to rely on sophisticated *ad hoc* techniques to manage data, and the resulting systems unavoidably have significant drawbacks in system upgrade and maintenance compared with a software database approach. The need for a high-performance real-time database system is clear.

Motivated by such need, in this thesis we explored possible ways to improve main memory database performance level. We applied our ideas to the design and implementation of a memory-resident object repository called SMOS (Shared Memory Object Store), and integrated SMOS into Open OODB from Texas Instruments, Inc. for various performance tests.

SMOS features an unique client-only architecture. Unlike the client-server architecture, the server process is eliminated in SMOS. As a result the communications between the server and the clients are also eliminated. SMOS also supports the notion of *real-time object* and *non-real-time object*. For real-time objects, SMOS provides the highest possible performance level by trading off some of the conventional database functionality

[1] The value of the bounded worst-case transaction time

and object features; when there are no real-time objects, the performance is not as good, but more database and object features are preserved. In the timing tests of SMOS/Open OODB, we observed[2] a worst-case transaction time of 10 milliseconds for real-time objects with size under 1Mb. For none real-time objects, the worst-case transaction time increased to 17.8 milliseconds.

1.2 Background

SMOS first is an object database; it provides persistent object storage for object-oriented programming languages such as C++. Because SMOS keeps persistent objects in memory, it is also a main memory database and thus can deliver very fast transactions. In addition, our efforts intend to provide guarantees for the worst-case transaction time, therefore SMOS is also oriented toward a real-time database. In the following sections we briefly describe the characteristics of these databases as well as related research in the University of Rhode Island.

1.2.1 Object-Oriented Databases

An Object-Oriented Database Management System (OODBMS) stores, shares, and manages objects instead of tables of data as in Relational Database Management Systems (RDBMS). Objects in an OODBMS reassemble real-world objects in many ways; they all have states, behaviors, and identities, and the structure and behavior of similar objects can be defined in common classes. An OODBMS maintains all the basic functionality of a traditional database management system, such as persistency, concurrency, and recoverability, by transparently integrating database capabilities with an object-oriented programming language such as C++ or SmallTalk. Unlike a RDBMS which stores simple and often fixed length data in tables and thus has difficulties in representing complex relations, an OODBMS can easily support complex structures by naturally using objects,

[2] The tests were performed on a Sun SPARC 10 workstation and its hardware configuration is listed in *Table Chapter 5 .1*

and thus can "*maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity, and identity and can easily be identified and operated upon*" (Elmasri, 1994). In other words, if the real-world information can be better represented by objects, it can often be easier and faster to store and manage this information in the form of objects rather than translating between application objects and tables of record. More importantly, in OODBMS the same object can be used transparently in many aspects of the system, including analysis, design, implementation, query, GUI, object store, etc. Therefore, developers who use object technologies desire OODBMS, because the combination of object-oriented analysis (OOA) and design (OOD), object-oriented programming language (OOP), and object-oriented database (OODB) offers the benefits of a synergistic development environment (Loomis, 1995).

1.2.2 Memory-Resident Databases

The cost for semiconductor memory chips, especially the Single Inline Memory Module (SIMM[3]) has dropped dramatically in recent years. In the meantime, the memory density has also increased significantly. Nowadays, a personal computer can hold as much as several hundred megabytes of memory, whereas an industry customized computer can go up to several gigabytes and more. These advances in memory technology make it possible to keep the entire database in the main memory. However, a main memory database (MMDB) is quite different from a traditional database that can cache all its data in the memory. A database that caches all the data in main memory is not designed to take advantage of the data memory-resident features, because its data access method is still oriented toward disk-resident data. A study of index structures for main memory database systems (Lehman, 1986) demonstrated that B/B+ trees do not have overall good performance in MMDB when compared to their performance in disk-based databases, because although B/B+ trees have the ability to minimize disk accesses and to use disk space efficiently, they cannot use the CPU cycles and memory space as efficiently. On the other hand, because of the low latencies between the CPU instructions and memory

[3] An industry standard for placing a grouping of memory chips on a pluggable board

accesses, MMDB is usually designed to take advantage of pointer following for its data access and representation, efficiently using the CPU cycles. An excellent overview of main memory databases can be found in (Garcia-Molina, 1992).

1.2.3 Real-Time Databases

A computing system is considered real-time if the correctness of the computation depends not only on the logical correctness of the results but also on the timing correctness of the computations. A real-time database (RTDB) provides data management services for applications that require logical consistency as well as temporal consistency for the data. To maintain temporal consistency a transaction is required to be completed by a certain deadline. Typical real-time database systems approach the guaranteed transaction time with various time-driven scheduling and resource allocation algorithms, they usually explicitly deal with time constraints by tailoring the traditional concurrency control and transaction management techniques (Ramamritham, 1993). However, these approaches do not always provide satisfactory solutions especially for guaranteeing *hard transaction deadlines*. As we mentioned earlier, this difficulty comes from the incompatibilities caused by maintaining both logical and temporal consistency, as well as system resources and hardware limitations. As a result, most real-time database systems only guarantee *soft transaction deadlines*.

A comprehensive overview of real-time databases can be found in (Ramamritham, 1993). This paper describes the characteristics of data and transactions in real-time databases, the issues that relate to the processing of time-constrained transactions, and the possible approaches to resolving contention over data and processing resources.

1.2.4 RTSORAC

RTSORAC (Real-Time Semantic Objects Relationships And Constraints) is a database model developed by the real-time research group at the University of Rhode Island (Wolfe, 1993; Peckham, 1994 and Prichard, 1994). This model incorporates real-time database concepts into an object-oriented database model, supporting time constrained objects and transactions. Based on this RTSORAC model, a prototype of a real-time object-oriented database system has been proposed (Wolfe, 1994). This thesis project is part of the implementation efforts of this RTSORAC database prototype.

1.3 Objectives

In this study, our primary research objective is to develop a persistent object storage manager that is suitable for object-oriented real-time applications requiring an extremely high-performance level. Our approach is step-wise and explores different balances between the database features and the system performance levels. Therefore, another important research objective is to investigate a flexible object management configuration that can be easily tuned to satisfy different application requirements. Our final research objective is to implement a prototype of our design and integrate it into an existing OODBMS with various timing tests to evaluate the design decisions.

1.4 Thesis Outline

Chapter 2 describes work that is related to SMOS, including making objects persistent, persistence in main memory, and meeting transaction deadlines. Chapter 3 presents the requirements, analysis, and design of SMOS. Chapter 4 describes the implementation and integration of SMOS with Open OODB (the host OODBMS for SMOS). Chapter 5 demonstrates the performance evaluations of SMOS in Open OODB. Chapter 6 concludes the thesis with a summary and discussion of the contributions, limitations, and future work.

Chapter 2

RELATED WORK

In this chapter, we describe work that relates to the design and implementation of SMOS. We will first introduce the object model, because its construction is the first step of building an OODBMS. We will then describe how objects are mapped between transient and persistent address space, as the mapping methods determine many aspects and features of an object store. We will also describe how objects can become persistent in main memory, because memory-residency is the key feature of SMOS. At the end of this chapter, we will introduce deadlines that are associated with real-time transactions, as SMOS intends to serve as a real-time database engine.

2.1 Object Model

The *object model* of a OODBMS specifies the semantics that can be explicitly defined for the system. It determines the characteristics of objects, how objects can be related to each other, and how objects can be named and identified (Cattell, 1996). Recall that an OODBMS is closely coupled with one or more object-oriented programming languages, thus we can classify OODBMS object models into language-independent and language-dependent models, according to their relationships with the coupled languages. In this section, we will compare these two categories of object models; this comparison provides the background of the object model independency requirement that will be described in Chapter 3.

2.1.1 Language-Independent Object Models

Language-independent models have the advantages of being able to specify a wide range of semantics. When targeting different applications, these models can have very different emphasis. For example, the RTSORAC Object Model specifically incorporates real-time constraints to support real-time persistent objects (Prichard, 1994). The ODMG-93 Object Model is another example in which a set of object constructs are established as standard requirements for a ODMG-93 compliant OODBMS. Though the language-independent object models have the advantage of flexibility, e.g., they can be very simple or very complex depending on the specific application requirements, they need to be bound with the coupled programming language. Sometimes, the bindings of certain semantics can be a challenge to system design and implementation. For example, to provide object relationship integrity, the coupled programming language needs to bind inverse object attributes; however, to provide support for such binding in the OODBMS is not trivial, because the system has to transparently maintain the references between the relationships in synchronization with the transactions that manipulate them.

2.1.2 Language-Dependent Object Models

Any object-oriented programming language (OOPL) has an implied object model that supports the concept of objects, operations, interfaces, types, subtyping, inheritance, etc. When an OODBMS chooses an OOPL with which to couple, it can also decide to use the language object model with or without modification for its database object model. Such a language-dependent object model approach makes it possible to provide only one execution environment, programming language, and type system through out the database system, eliminating *impedance mismatch*[4] between the programming language and the database. In addition, because the database and the language use the same object model, there is no need to map the database object model into the language, eliminating

[4] Impedance mismatch arises when the application programming language and the database language have very different object models; it refers to the problems related to translating objects in these two language environments.

language binding that is required by a language-dependent object model. Although the semantics of such a language-dependent object model may not be as rich as a language-independent object model that is specifically designed for an OODBMS (e.g. ODMG-93), the computationally complete programming language that is closely coupled with the system, such as C++, can usually overcome the resulting shortcomings (Cattell, 1994).

2.2 Making Objects Persistent

A persistent object is one that continues to exist after the process in which it is created has terminated. Because an object can have pointers or references to other objects, making objects persistent as well as retrieving persistent objects are rather complex procedures and require close attention from the object store. Depending on how mappings between transient objects and persistent objects are performed, there are basically two different ways to make objects persistent in a database. In the following sections, we take a closer look of these two mapping approaches.

2.2.1 Indirect Mapping

In this approach, the mappings between transient and persistent objects are indirect (*Figure Chapter* 2 .1). A transient object in the local heap of an application process must be first transformed into a format that is suitable to store in the persistent addressing space (PAS), it is then moved into the PAS where it becomes persistent. Though the actual format varies with different implementations, the object is simply a segment of some storage units (such as bytes, blocks, pages) with a length that is determined by the size of the object.

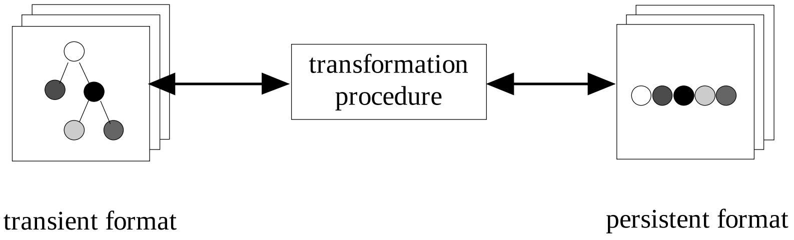


Figure Chapter 2 .1 Indirect Mappings between transient and persistent objects

If objects of any type are allowed to become persistent, the transformations between an object's transient and persistent representations can be very complicated. Because an object can have references/pointers to other objects, the memory addresses contained in the references/pointers are most likely invalid when the objects are fetched back from the persistent store to the memory; therefore, the transformation has to find a way to represent references/pointers without using memory addresses. In such an indirect mapping approach, an object identifier (OID) is typically used to represent a reference/pointer in a persistent object; and when dereferencing the pointer, the OODBMS performs a table lookup to find the persistent address of the target object using its OID.

As the object transformation has to traverse references in the object, indirect mapping can be time-consuming when the object contains many references or pointers. However, indirect mapping also has its important advantages. Firstly, if the OODBMS needs to support cross-platform transfer of objects over networks, or support multiple application programming languages, a general object format is needed other than the in-memory transient representation. This general object format is often best represented using a persistent object format which utilizes fixed length contiguous storage units. Secondly, from the architecture point of view, the object store should serve as a module in the OODBMS if extensibility is a key requirement. Under object oriented design (OOD) principles, such object store modules should be coupled with the OODBMS only through an interface and should not care about the actual structure or layout within each individual object. This can be done by providing a persistent object format that is known to the object store only as a sequence of uninterpreted storage units. Such a design has great flexibility; for example, it can easily adapt a new object store for a different system functionality emphasis. Because of this, we used the indirect mapping approach in SMOS.

The idea of indirect mapping has been successfully applied in several OODBMS, including research prototypes such as E/Exodus (Carey, 1986), Open OODB (Well, 1992), YOODA (Abecassis, 1994) etc., as well as commercial products such as MediaDB, O2 (Deux, 1991), etc. Further, the OO7 OODBMS benchmark has demonstrated that E/Exodus has a very competitive performance when compared with several commercial OODBMS products that are based on direct mapping technology (Carey, 1993).

2.2.2 Direct Mapping

In the direct mapping approach for making object persistent, the mappings between transient and persistent objects are direct (*Figure Chapter* 2 .2). No transformations are needed. Objects are stored in their native language format such as C++ format. The only difference between transient and persistent objects is their pointer values. Instead of containing a memory address when it is in a transient object, the pointer will have a *persist storage address* when the transient object is mapped into the persist object store. Therefore, the direct mapping procedure simply converts a pointer's addresses between the two address space before moving objects across the boundary, this procedure is also called *pointer swizzling*. An in-depth description of various pointer swizzling techniques can be found in (Moss, 1992). reminder

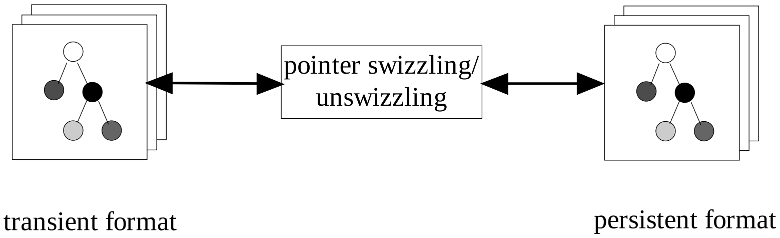


Figure Chapter 2 .2 Direct Mappings between transient and persistent objects

Although the direct mapping approach seems more efficient at run-time, it complicates the system design and implementation and has limited extensibility compared with indirect mapping, because the storage manager has to be closely tied to many components of the OODBMS and can no longer exist as a standalone system module.

2.3 Persistency in the Memory

Unlike secondary storage, main memory is usually volatile. As a result, to maintain data persistence in memory requires a running process that is capable of dynamic memory allocation. This process needs to control a pool of memory space, allocating and deallocating blocks of memory space upon request. Data stored in the allocated memory space becomes persistent as long as this process is running. Similarly, a database can achieve data persistency in memory with a client/server architecture in which the server process allocates memory space to store data for the clients. A client-only architecture can also be used for data persistence in memory. Such architecture requires shared memory support from the operating systems, a standard feature[5] in System V UNIX. Typically, shared memory is used for interprocess communications; data stored in shared memory is accessible by different processes and is persistent until the segment containing this data is explicitly deleted or the kernel stops running. Such a feature can be directly used for sharing persistent data in main memory without introducing a server process, and is fundamental to this thesis project.

2.4 Meeting Deadlines

As we mentioned in Chapter 1, a database maybe required to manage time-constrained data. As a result, the transactions using this data have to meet certain deadlines, otherwise

[5] Though shared memory is not standard in BSD UNIX, it is supported in many extended BSD UNIX systems.

this data is no longer considered consistent. A database that is capable of dealing with transaction deadlines is qualified as a real-time database (RTDB).

We mentioned earlier that it is very difficult for a database to provide an absolute guarantee on meeting transaction deadlines because of the incompatibilities between logical and temporal consistency, as well as limitations in system resources. Most research in the real-time database community has focused on finding solutions for the incompatibilities between logical and temporal consistency (Abbott, 1988; Abbott, 1990; Buchmann, 1989; Chen, 1990; Huang, 1990; Huang, 1991; DiPippo, 1995; Sha,1991; etc.). These approaches usually first construct a real-time transaction model, and then devote most of their efforts to establishing suitable transaction scheduling and concurrency control protocols around the transaction model. A typical transaction model may include the following attributes: deadlines, value function, resource requirements, execution time, semantic information, etc. These attributes are used for transaction scheduling of the CPU, I/O, memory, and other resources. This scheduling typically involves priority assignment to the transactions. In addition, concurrency control is also considered differently, because traditional lock-based protocols are no longer acceptable due to possible *priority inversion* and *deadlock*. Strategies, such as *Wait Promote* (Sha, 1990), *High Priority* (Abbott, 1988), *Conditional Restart* (Abbott, 1988), are proposed to prevent priority inversion for lock-based concurrency control protocols; and strategies, such as aborting a transaction that has already missed its deadline, has the longest deadline, or is least valuable (based on value function), are proposed to break deadlock.

Chapter 3

THE REQUIREMENTS, ANALYSIS AND DESIGN

3.1 Overview

The goal of this thesis research is to develop a persistent object store that is suitable for the demands of real-time database applications. Our focus on SMOS is performance. Typically, the performance of a particular software application is determined by three factors; the features required, the design, and the implementation. The inclusion of more features, especially advanced features, often requires more run-time overheads as well as more efforts in the design and implementation. Given the same required features, the design can choose different system architectures with different emphasis, and these design decisions can have great impact on the performance; in addition, because design also determines system portability and expandability, it can affect future system performance. Implementation, on the other hand, affects performance through the kind of data structures and algorithms it uses, and unlike the design it is much easier to correct or improve. In this chapter, we first describe the requirements of SMOS, and then present the design based on the requirements.

3.2 The Requirements

Like any software system, SMOS has a set of functional requirements that describe the functional capabilities or features it has to provide (*Table Chapter* 3 .1). In addition, SMOS also has a set of system requirements that describe the system capabilities it must have when satisfying the functional requirements (*Table Chapter* 3 .2). There requirements directly affect the design of SMOS.

3.2.1 The Functional Requirements

Object Model Independence
Multi-Level Object Persistence
Guaranteed Persistent Object Access Time
Adjustable System Performance
Concurrent Persistent Object Access

Table Chapter 3 .1 SMOS Functional Requirements

Object Model Independency

As we mentioned in Chapter 2, an OODBMS can choose an object model that is either dependent or independent upon its binding language object model. To serve as an independent object storage manager, SMOS does not make any assumptions about the object model of the OODBMS. This gives SMOS the potential to provide persistent object storage service for a wide range of OODBMS.

Multi-Level Object Persistence

An object is persistent if its lifetime is extended beyond the execution of the creating program. To provide performance advantages, we put the main store of SMOS in main memory. However, main memory is normally volatile, the lifetime of a main memory persistent object is therefore limited by the up time of the OS kernel. To provide recovery from kernel crash or power failure, SMOS must be able to coordinate nonvolatile media persistent object storage and retrieval, such as asynchronically mirroring main memory persistent objects on disk following certain policies.

Guaranteed Persistent Object Access Time

One of the major unpredictabilities of a disk-based database is data access time. This unpredictability is caused by blocking time uncertainties associated with disk I/O (Singhal, 1988). Because SMOS is supposed to provide object storage service for real-time databases, it must be able to guarantee the access time of persistent objects.

Adjustable System Performance

As we mentioned earlier, the system performance is determined partially by the features or function requirements. We can often improve performance by leaving out some of the functionalities. On the other hand, the same functionalities maybe mandatory in a different application or even the same application but different stages. SMOS should be able to adjust its performance and functionality balances according the application requirements.

Concurrent Persistent Object Access

Persistent objects are a shared resource. If this shared resource can be accessed concurrently, performance can be greatly improved. For this reason, SMOS must provide persistent object sharing among multiple transactions, users and applications. In order to ensure data integrity, SMOS must only allow controlled concurrent access to persistent objects. However, as we discussed in Chapter 2, concurrency control undermines performance; therefore SMOS should be able to provide flexible concurrency control to balance performance and data integrity according to the application requirements.

3.2.2 The System Requirements

Performance
Portability
Reusability
Extendibility
Flexibility

Table Chapter 3 .2 SMOS System Requirements

Performance

Because the goal of SMOS is to provide persistent object storage for demanding real-time applications, performance is the most important system requirement for SMOS. As a persistent object store, the performance of SMOS is measured on persistent object access time. Given the functional requirements in last section, persistent object access time in SMOS should approach transient object access time which is the speed of main memory. Such performance level is fundamental to satisfy real-time transactions with very tight deadlines (e.g. in several milliseconds range). Essentially, the performance of SMOS should be guaranteed at a extremely high level in order to provide persistent object storage service for demanding real-time database management systems..

Portability, Reusability, Extendibility, and Flexibility

Portability is very important when cross-platform is to be supported. SMOS is required to support various UNIX platforms, therefore, it should avoid using platform dependent function calls. SMOS should also have high reusability in order to make its service available for a wide range of OODBMS, this requires SMOS to be a well designed module with interfaces can be easily adapted to various systems. Extendibility is another important system requirement, because an OODBMS may have specific storage requirement which SMOS does not initially have. Finally, SMOS should be able to easily turn on and off some of its features; such flexibility is essential to satisfy different performance and functionality requirements between different applications.

3.3 The Analysis

Although the requirements described in last section all have influences on our design of the persistent object store, among them guaranteed high-performance has the greatest challenge and plays the most important role in our design decisions.

To satisfy the performance requirement, we must keep the object store in main memory. Such approach not only improves performance but also eliminates disk I/O which in turn eliminates blocking time uncertainties. Conventionally, a memory-resident store is either encapsulated within the private address space of a server process or allocated to the memory shared by one or more server processes (*Figure Chapter* 3 .1); the server process then communicates with application processes using IPC, RPC or other networking methods. Though these approaches have many advantages provided by the much matured client/server architecture, such as clearly defined functionalities and boundaries for easy maintenance and high reliability, the overhead imposed by the communications between the clients and the servers undermine the performance. Clearly, we can further improve performance if we are able to eliminate this communication overhead; and the only way to achieve this is to remove the server process. However, without any process with which to attach, the store must find a way to become persistent in memory and to be sharable by all the application processes. The System V or POSIX.1b Shared Memory facility actually makes this possible. The original purpose of shared memory is to provide fast interprocess communications; once created, a shared memory segment can be attached to one or more processes for their addressing; any changes made to the shared memory by a process become immediately available to others. Most importantly to us, the shared memory is protected by the kernel; the content of a shared memory segment is preserved until it is explicitly deleted or the kernel is stopped. Therefore, if we can make the object store appear as one or more shared memory segments, the basic database framework is established.

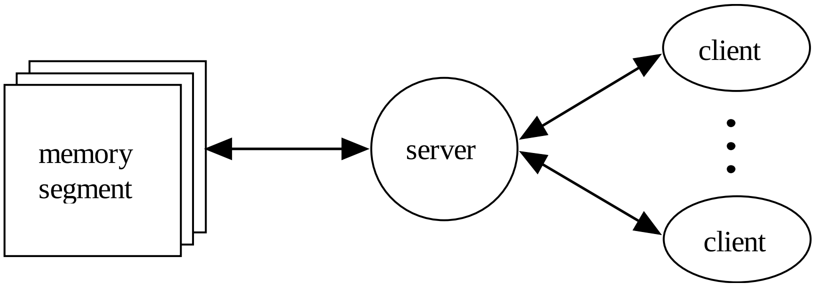


Figure Chapter 3 .1 A Conventional Memory Resident Database

Another source of overhead commonly found in a conventional main memory database is the object format translation and object moving and copying. The translation is needed if a general object format is introduced in order to support networking, cross-platform, and multi-languages; whereas object moving and copying is for supporting multiple address space and recovery. Though these are important features to many applications, we realize that they are not necessary to some applications. For example, in a real-time environment, conventional recovery becomes useless if the recovered data is outdated due to the temporal constraints. For this reason, if we can turn off these features when possible, we will gain further performance improvement.

3.4 The Design

Based on the analysis in last section, we can make three major design decisions to improve the performance of the persistent object store (*Table Chapter* 3 .3). In the following sections, we first present our high level system architecture which reflects our major design decisions, and then describe our detailed object-oriented design using Booch notation.

Design Decisions	Impacts
Memory-Resident	Improves access time, eliminates blocking uncertainty
Client-Only/Serverless	Eliminates communication overhead
Address Space Unification	Eliminates object format translation and object moving

Table Chapter 3 .3 SMOS Design Decisions

3.4.1 The Architecture Framework

As shown in *Figure Chapter* 3 .2, the main store of SMOS resides in shared memory. As the content of the store is naturally persistent with the protection from the kennel, no hosting process is needed in order to prevent this memory area being reused by other processes. By utilizing shared memory facilities provided by the operating system, client processes can also gain direct access to the store concurrently without communicating with each other (we will describe how concurrency control is established without a server process in section 3.3.2). Based on the above features provided by shared memory, we eliminated the server process in the design which is otherwise essential in a database.

Because the maximum lifetime of regular[6] main memory content is limited to the kernel up time, we have a daemon (*Figure Chapter* 3 .2) that provides asynchronous backup services for the main object store. Following certain policies, this daemon extends object lifetime by copying whole or part of the main store to a backing store on disk, whereas the policy can be configured so that it balances the extent of recovery and the system performance according to application requirement.

[6] There are special memory chips/boards that have battery backup to against power failure.

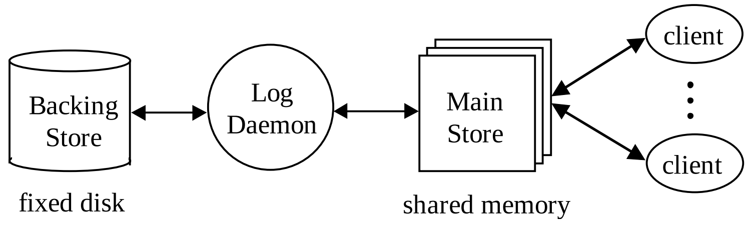


Figure Chapter 3 .2 SMOS System Architecture Framework

Conventionally, for safety and recovery, a client is not allowed to directly address the persistent address space; in other words, the persistent and transient address spaces are separate. In our design, after a client attaches the persistent address space to its local transient address space, it can optionally unify the two address space when addressing time critical objects. Because the client directly operates on the persistent object without copying the object out from and back to the store, the access time of any persistent object is bounded regardless the size of the object. However, we must realize that use of such operations should be restricted because it can put the object in a inconsistent state if failure should happen during the transaction.

3.4.2 The Classes

In this section, we will have a closer look at our design. In particular, we will decompose the system and use Booch OOD notation (Appendix I) to represent our detailed design decisions.

The class diagram in *Figure Chapter* 3 .3 demonstrates most of the SMOS system components. At the top level is the class AddressSpaceManager; it is a system interface class that serves as an entry to SMOS. The attribute shmASM/SharedMemoryManager[7] in class AddressSpaceManager leads to SMOS low level. High level persistent object operations, such as createObject, modifyObject, getObject, writeObject, and deleteObject,

[7] This notation represents that the attribute shmASM is an instance of class SharedMemoryManager or is a pointer to an instance of class SharedMemoryManager.

eventually use the interface functions in class SharedMemoryManager to get to the objects in shared memory.

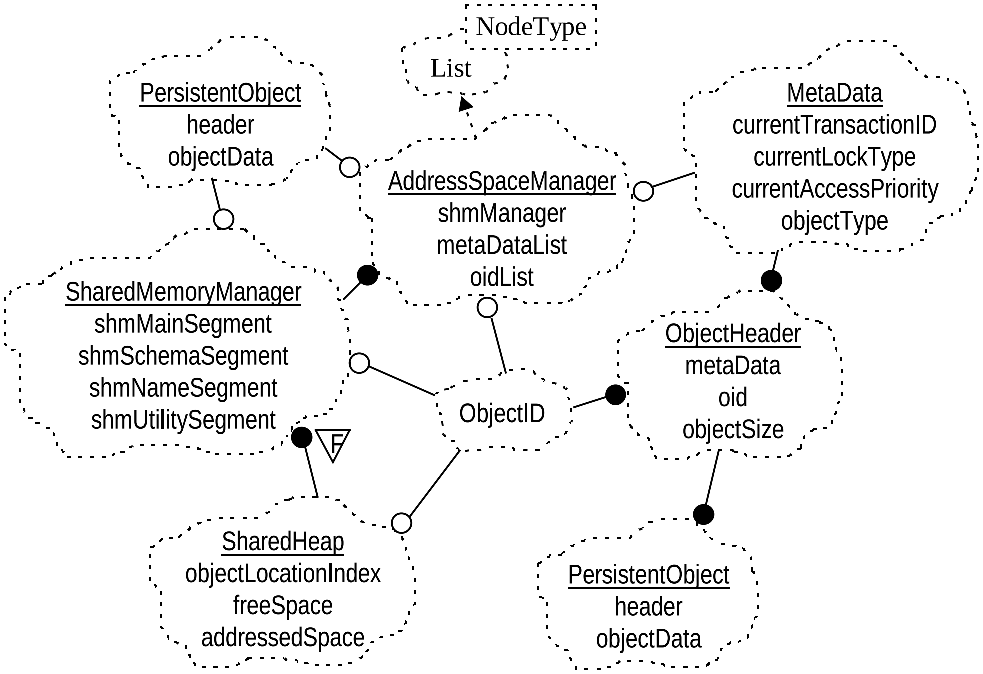


Figure Chapter 3 .3 SMOS Major Class Diagram

In class SharedMemoryManager, the attribute shmMainSegment/SharedHeap contains the actual shared memory segment that is used for persistent object storage as well as a memory allocator for object storage allocation/deallocation within this segment, the attribute shmSchemaSegment[8] stores database schema such as persistent object type information, the attribute shmNameSegment mapps each user defined object name to the object's unique identifier (OID), and the attribute shmUtilitySegment is used for supporting various storage and transaction management.

The class PersistentObject is very special. Its attribute objectData stores the user object, and because it is a region of storage that occupies some contiguous bytes, it is independent of the object model being used by the OODBMS. Although the class PersistentObject is not itself a system component class, the data carried in its attribute header/ObjectHeader provides important supports for storage and transaction management (detailed in Chapter 4).

[8] A single attribute alone indicates that the attribute is of primitive type or is a pointer to a primitive type.

Chapter 4

THE IMPLEMENTATION

In this chapter we will describe the implementation of SMOS based on the design presented in Chapter 3. However, to make SMOS fully functional, we need to integrate SMOS into an existing OODBMS. In the following sections, we first introduce Open OODB, the hosting OODBMS for SMOS, and then describe the implementation of SMOS in Open OODB.

4.1 Open OODB

4.1.1 An Overview

The Texas Instruments Open OODB Project was sponsored by the Advanced Research Projects Agency (ARPA) and was managed by the US Army Communications-Electronics Command (CECOM). It was an effort to develop an architecture framework (*Figure Chapter* 4 .1) in which database functionalities can be easily tailored to meet application requirements. The fundamental characteristics of the Open OODB architecture is its high degree of modularity; important database functionalities, such as persistent storage management, transaction management, and query processing, are designed and implemented as independent system modules. In addition, the interfaces of these modules are also made available to the users, making it possible for application developers to add or remove modules for different applications. Because of such openness, modularity and availability[9], we have chosen Open OODB to serve as the testbed for SMOS in this thesis research.

4.1.2 Architecture

Because extensibility is the focus of the Open OODB Project; it requires that database functionalities can be tailored easily for different applications. Such requirement led to the Open OODB modular architecture (*Figure Chapter* 4 .1).

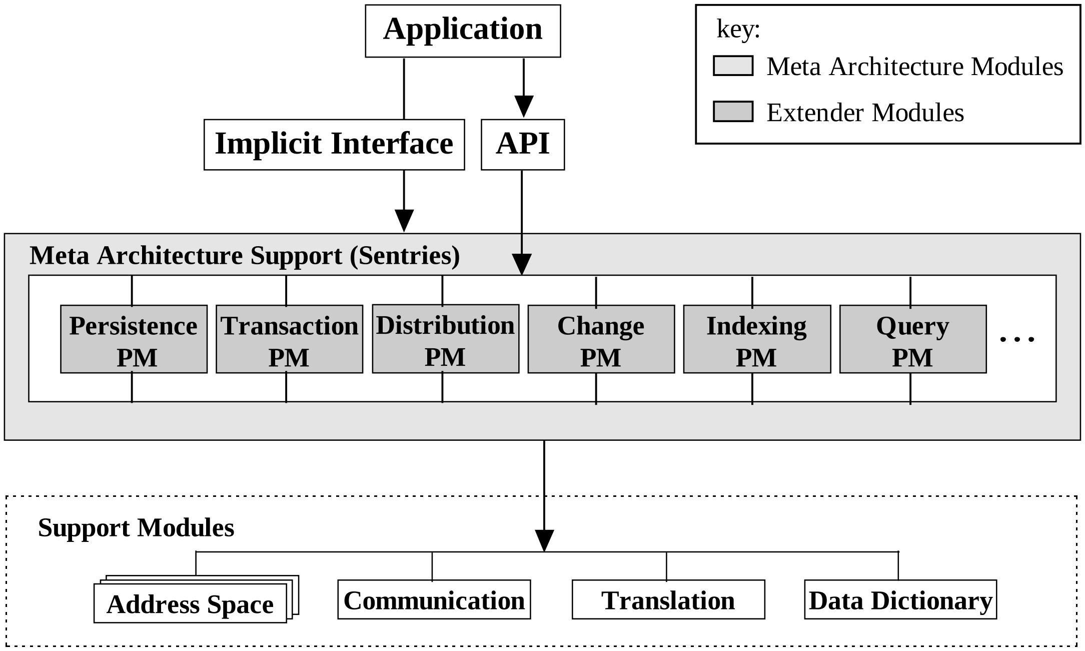


Figure Chapter 4 .1 Open OODB Architecture

Among the modules in Open OODB, some are essential and some are optional to a particular system configuration. The essential modules, called *meta architecture support*

[9] The University of Rhode Island was one of the 20 test sites of Open OODB and has access to Open OODB full source code.

modules, provide common services needed by other modules and therefore must be present if any of the supported modules is present; these modules include *Address Space Manager, Communication, Translation*, and *Data Dictionary*. The optional modules, called *extender modules*, provide services that may or may not be needed in a particular application and therefore must be independent to each other; these modules include *Persistence*, *Transaction*, *Distribution*, *Change*, *Indexing*, *Query*, *etc*. With such modular partitioning of database functionalities, Open OODB can be configured in various ways depending on the application requirements. On one extreme end, if none of the optional functionalities are needed, both extender and support modules can be excluded, this makes Open OODB simply a regular C++; on the other extreme end, if all of the optional functionalities are needed, every module has to be included, this makes Open OODB a full featured and thus heavy weight OODBMS. In between the two end, many configurations are available.

Because Open OODB is closely coupled with C++, the extender modules are actually language extensions. For example, the persistence module extends C++ to support persistent objects. The detection of the need for extension is a service provided by the meta architecture support modules. For each extender module, Open OODB provides a *sentry* in the meta architecture support modules to detect the corresponding extension event. Once detected, the event will be trapped and its handle will be passed to the extender module by the sentry.

The above modular design of Open OODB makes it possible for SMOS to replace the address space manager in Open OODB, a modified version of EXODUS. In the next section, we will describe how SMOS is implemented and integrated into Open OODB.

4.2 SMOS Implementation

The major class hierarchy for Open OODB address space management (ASM) is shown in *Figure Chapter* 4 .2. The class OODB is the database interface class, and exactly one instance of this class can be instanciated in each application process[10]. Because instances of module classes are members of class OODB, an application can seamlessly access a database as soon as it instanciates the class OODB:

```
class OODB {
private:
        ASM_Client              * asm_mgr;
        Trans_Mgr               * trans_mgr;
        Transaction             * current_trans;
        LASM                    * lasm_mgr;
        TYPE_MGR                * type_mgr;
        NAME_MGR                * name_mgr;
        PERSIST_MGR             * persist_mgr;
        XTRANSLATE_MGR          * xtranslate_mgr;
        ...
}
```

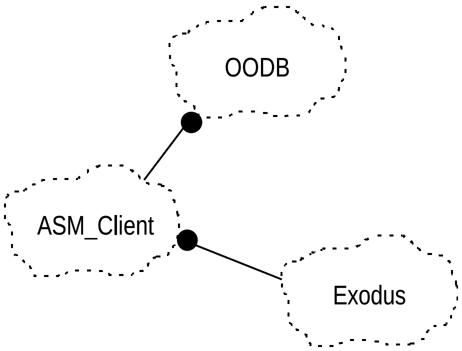


Figure Chapter 4 .2 Open OODB ASM Class Hierarchy

[10] We consider this as a limitation in Open OODB, because an instance of class OODB is initialized with exactly one database, therefore it only allows an application to access one database at a time.

To make Open OODB work with SMOS, we replaced class Exodus with our class SharedMemoryManager (*Figure Chapter* 3 .3) and reimplemented class ASM_Client without changing its interfaces (the new class ASM_Client is actually a form of our class AddressSpaceManager). The four shared memory segments (shmMainSegment, shmSchemaSegment, shmNameSegment, and shmUtilitySegment) in class SharedMemoryManager are installed by a database management utility during database initialization and are attached to the application processes after instanciating the class OODB (remember that class SharedMemoryManager is now a member of class OODB). To demonstrate the details of this integration, in the following sections we will have a closer look of the implementation of persistent object addressing and transaction management with respect to Open OODB.

4.2.1 Persistent Object Addressing

The shared memory segment shmMainSegment/SharedHeap in class SharedMemoryManager provides the persistent address space for SMOS. The underlying memory allocator is based on a *binary buddy system* scheme (Kruth, 1973), and its implementation was obtained from work done by John Black in the Real-Time Research Group at the University of Rhode Island.

To access a persistent object, we first need to obtain its address in the main store using its unique object identifier (OID); the mapping between an OID and address is kept in an persistent index (objectLocationIndex/SharedHeap as shown in *Figure Chapter* 3 .3). The well-known main memory database index study in (Lehman, 1986) demonstrates that *chained bucket hashing* , though has relatively high storage costs, has the fastest execution time compared with *array, AVL tree, B tree, T tree, extendible hashing,* and *linear hashing*. In SMOS, because the size of the shared memory segment must be pre-fixed[11], this in turn requires a fixed size index table, therefore the high storage cost is unavoidable independent of the kind of index being used. In addition, the execution time

[11] There is no support for dynamically changing shared memory size in System V.

(including insert, search, scan, and delete) of the hashing is constant independent of the size of the index; this is particularly important for SMOS, because SMOS intends to provide persistent object storage for real-time applications. Based on the above facts, we decided to implement the index with a chained bucket hashing. However, as we do not allow dynamic allocating space for collided entry, the regular chained bucket hashing is modified so that free entries in the index are used for collisions (*Figure Chapter* 4 .3).

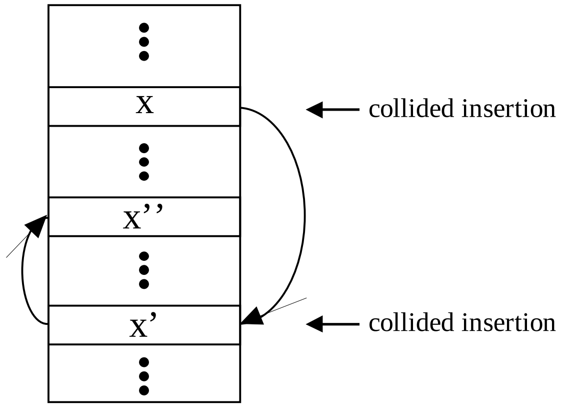


Figure Chapter 4 .3 Index Structure (x' and x'' indicates chained entry)

Based on the analysis in Chapter 3, after we obtained the persistent object address, we can provide two choices for addressing, they are indirect addressing and direct addressing. The particular addressing method for an persistent object is determined by the users through the way they create this object, and this information is kept in an one bit field (addressingMethod) in the meta data as shown below:

```
class PersistentObject {
private:
        ObjectHeader   * header;
        ...
};


class ObjectHeader {
```

```
private:
        MetaData *  metaData;
        OID         oid;
        int         objectSize;
        ...
};

class MetaData {
private:
        BIT lockType:2;
        BIT transactionID:16
        BIT priority:8;
        BIT addressingMethod:1
        ...
};
```

In our implementation we also provide facilities for the users to change the addressing method after a persistent object is created or to force an addressing method in a transaction no matter what the default method is; therefore, the same persistent object can be accessed differently according to application requirements. In the following sections we provide some more information about the implementation of the two addressing schemes.

4.2.1.1 Indirect Addressing

Indirect addressing is a conventional method, it provides safeguard against various failures. In our implementation, a copy of the persistent object is made in the transient address space for the application to work on, and it is copied back to the persistent address during commit time. In Open OODB any transient object can become persistent by calling a common member function that is added by the Open OODB preprocessor. During commit, the object header (header/ObjectHeader) is composed by SMOS and added to the object data potion, they are then copied together as a whole to the main store. In the object meta data the default value for addressingMethod marks indirect addressing; therefore, when this object is later accessed, the addressingMethod field will lead to indirect addressing again. As we mentioned earlier, the users can also skip checking addressingMethod and force a particular addressing method. In either cases, if indirect addressing is to be used, we will copy objectSize bytes of shared memory content from the persistent object address to a local address; the users should make sure that this local address is a valid address for an transient object with the same type.

4.2.1.2 Direct Addressing

The direct addressing approach is considered controversial and has not been reported in any literature. The basic idea is to allow an application to directly operate on a persistent object without making a copy. Clearly, this approach has the greatest performance advantages because the data is accessed and updated at memory speed. However, for the same reason, traditional recovery is impossible in this approach. As a result, the state of the database cannot be guaranteed to be valid if the transaction aborts or failure occurs; therefore, such direct addressing is only useful for specific applications such as those managing high speed real-time data. In these real-time applications, data may not be able to afford recovery because the time used for recovering may be too long compared to the temporal constraint; therefore, recovered data may have become outdated and useless. Instead of recovering the data, we can simply obtain the data from its source which has the most updated information.

As described earlier, direct addressing can be forced; however such forced direct addressing is only valid on an existing persistent object. To directly address new persistent objects, we provide an overloaded *new* operator:

```
void * operator new(size_t sz, OODB * oodb, char * name);
```

Handles of all the database facilities are passed with oodb/OODB to the implementation of this overloaded new operator. The parameter name is a user defined name for the object, it is mapped to an unique OID and can be used to identify the object at user level. Objects created with this new operator resides directly in the main store. Because of this, any changes made to them are persistent and irreversible, therefore, direct addressing should be used with great caution. Objects created with this overloaded new operator also have their addressingMethod field marked for direct addressing. As a result, the next time these objects are accessed, their persistent addresses are directly used for addressing unless a indirect addressing is forced. Again the users have to make sure the persistent addresses are cast to objects with correct types.

4.2.2 Transaction Management

Because Open OODB directly uses transaction management that comes with Exodus, we have to provide a new transaction manager after removing Exodus from Open OODB. To provide more concurrent access which is very important for real-time applications, we used object level locking, a finer locking granularity compared with page-level locking in Open OODB/Exodus. Although it has been reported that page-level locking has better performance(Carey, 1994), we realize that such argument was based on the fact that page level locking has less communication overhead between servers and clients. In SMOS, because of its special architecture, the communication has been eliminated (Chapter 3); therefore, choosing object-level locking will not incur any performance degradation in SMOS.

Our current implementation of locking uses a strict two-phase exclusive locking (2PL) protocol. That is, locks can be obtained any time before a transaction commits but can only be released during or after commit time. The lock is held in a 2 bit field (lockType) in the meta data of the object being accessed. In the mean time, we also store transaction ID and priority in the meta data; these information will be used to support preventing *priority inversion* and detecting deadlocks as described below.

4.2.2.1 Preventing Priority Inversion

The problem with two-phase locking in a real-time transaction is the possibility of priority inversion. A priority inversion occurs when a higher priority transaction is requesting a conflicting lock that is held by a lower priority transaction. Because we use exclusive locking, any lock held by a lower priority transaction will block a higher priority transaction. There are basically two protocols for preventing such priority inversion, *Priority Abort* and *Priority Inheritance*[12]. The priority abort method aborts the blocking low priority transaction; whereas the priority inheritance method lets the blocking low priority inheritance the priority of the blocked high priority transaction. Because there are conflicting reports on which protocol performs better (Abbott, 1988; Abbott, 1989; and Huang, 1991), we implemented both protocols in SMOS and provide an option for the users to choose during application compile time. Our implementation of priority abort protocol also raises an exception when aborting a higher priority transaction; the application can use this exception to restart the transaction if needed.

4.2.2.2 Detecting Deadlock

To address possible deadlocks in two-phase locking, we implemented a deadlock detector in SMOS. Our decision to use a deadlock detection scheme instead of a deadlock prevention scheme is based on the fact that we have a fine locking granularity which lessens the interference among transactions when a limited number of objects are accessed. The deadlock detector is implemented by storing a transaction *wait-for graph* in the shared memory utility segment (shmUtilitySegment) and checking if there is a cycle is formed in the graph (*Figure Chapter* 4 .4).

[12] There are other locking based protocols, such as *Priority Ceiling*, that extends *Priority Inheritance*.

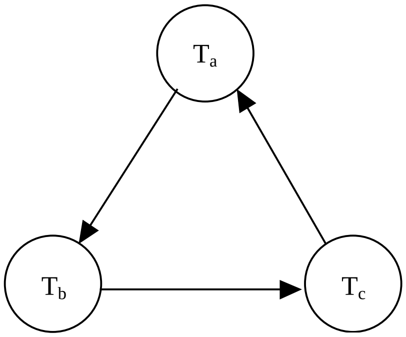


Figure Chapter 4 .4 A Deadlock Transaction Wait-For Graph

In *Figure Chapter* 4 .4, transaction T_a is waiting for T_b, T_b is waiting for T_c, and T_c is waiting for T_a, forming a waiting cycle and no one can proceed. Due to the same reason as our index implementation, we choose to use a pure array based representation of the wait-for graph.

Our initial implementation (*Figure Chapter* 4 .5) includes a chained bucket hash table and a two dimension square array. The hash table is very similar to the one used in the index structure, a bucket keeps the ID (the key) and priority of current active transactions, and is represented as *T(id, p)* in *Figure Chapter* 4 .5. The square array is an adjacency matrix representation of the wait-for graph, its subscripts correspond to the index of the hash table. The array contains Boolean elements, a true value in element [*i*][*j*] indicates that the transaction whose position is *i* in the hash table is waiting for the transaction whose position is j to release a lock. The class specification of the graph is shown below:

```
Class Bucket {
private:
        int transactionID;
        int  priority;
        int chainedBucketEntry;
         BOOL isDeadlockVictim;
        ...
};


class WaitForGraph {
```

```
private:
        Bucket   index[DIMENSION];
        BOOL  table[DIMENSION][DIMENSION];
        int   transactionCount;
        int   nextFreeChainningEntry;
        int   deadlockCheckingInterval;
         int    blockingCount;
         BOOL threading;
        ...
};
```

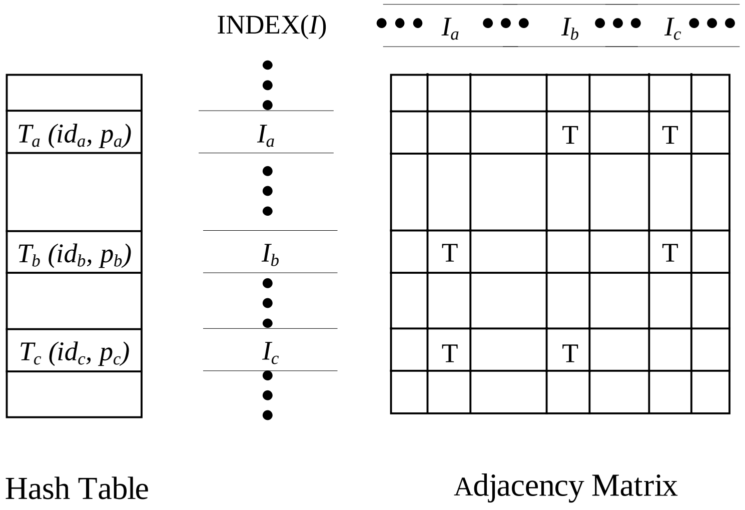


Figure Chapter 4 .5 Implementation of Transaction Wait-for Graph

Figure Chapter 4 .5 demonstrates the representation of the wait-for graph in *Figure Chapter* 4 .4. The checking for transaction wait-for cycles involves traverse the edges in the graph, and can be implemented with a variant of depth-first search algorithm that was discovered by Tarjan (Tarjan, 1972). Because a transaction can only wait for exactly one other transaction, the above implementation of the wait-for graph can be optimized by replacing the adjacency matrix with a linked list. In our implementation, we used a array-based representation of the linked list and packed it into the hash table:

```
Class Bucket {
private:
```

```
            int          transactionID;
            int          priority;
            int          chainedBucketEntry;
            Bucket *     transactionBeingWaited;
            BOOL         isDeadlockVictim;
            ...
    };

    class WaitForGraph {
    private:
            Bucket   index[DIMENSION];
            int         transactionCount;
            int         nextFreeChainningEntry;
            int         deadLockCheckingInterval;
            int         blockingCount;
            BOOL        threading;
            ...
    };
```

In this implementation, the checking procedure simply travels along the linked list; a deadlock is found if a transaction/bucket is visited twice.

We also provide two options for the users to set the checking policies during application compile time. In the first option (*Figure Chapter* 4 .6*a*), checking is done at the beginning of every blocking when there are more than one blocking(blockingCount). This approach has the advantage of detecting a deadlock as soon as it is formed, but it can be inefficient if the rate of blocking is very high or blockings occur very close in time to each, because not every blocking forms a deadlock. However, if there is no blocking it will never check for deadlock; therefore, it is the best choice if we know that limited blocking can occur. In the second option (*Figure Chapter* 4 .6*b*), checking is done after more than one blocking have occurred for a user defined period of time. This approach performs better when the blocking rate is high and the checking interval is properly set. However, unlike the first option, it cannot promptly detect a deadlock unless the interval is set very short. Unfortunately, the shorter the interval the more it will behave like the first option. For the above reasons, choosing the best approach is very difficult unless we have a better understanding of the application behavior.

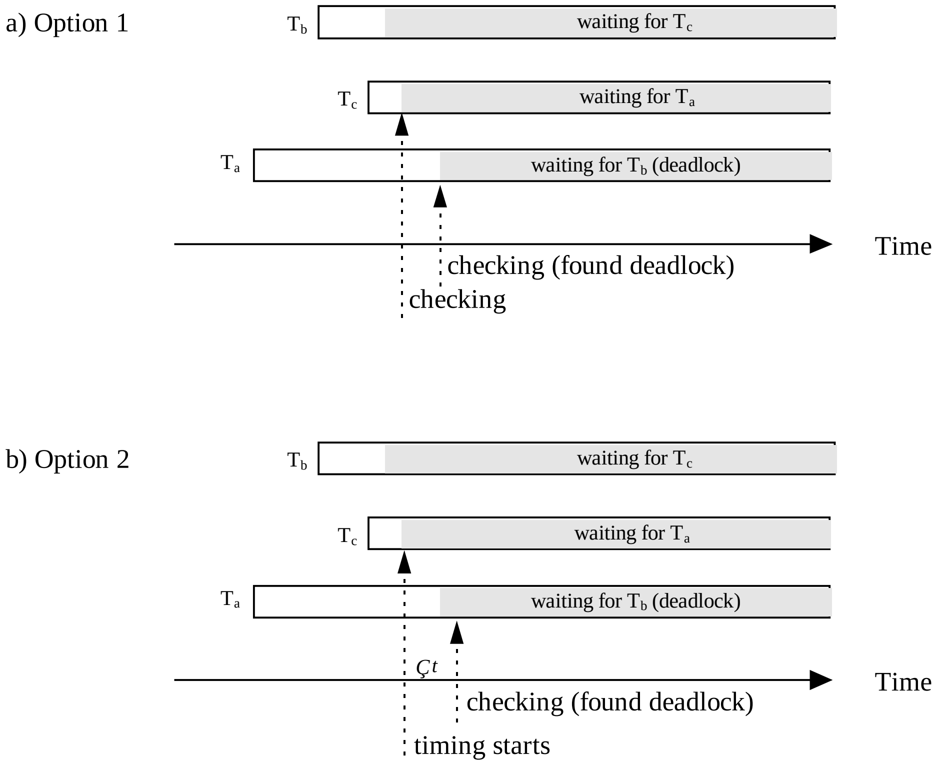


Figure Chapter 4 .6 Deadlock Checking Options

Under either approach, when a cycle is found in the wait-for graph, our policy is to abort the transaction that has the lowest priority. In our implementation, we provide two methods to check for deadlock. In the first method, we start the checking procedure in a new thread[13]. Upon detecting a cycle, the new thread sends a signal to the process whose transaction has the lowest priority on the cycle[14]. When this *deadlock victim* process catches the signal it aborts the transaction and throws an exception. The second method is shared memory based, a blocked transaction will watch for the isDeadlockVictim field in its wait-for graph entry and raises the same exception if this field is *true*. In this method, the deadlock checking procedure is initiated from a transaction main thread. If a deadlock is found, instead of sending a signal, the main thread tags the isDeadlockVictim field of the

[13] We use POSIX1.c threads through out the implementation.

[14] The thread knows where to send the signal because we uses process ID to represent transaction ID (Open OODB only allows sequential transactions).

deadlock victim entry in the wait-for graph. In either method, various decisions can be made by the user if the exception is caught, including restart the aborted transaction immediately, reschedule the transactions, or simply terminate the application process.

4.2.3 Extended Persistency and Recovery

An object is persistent in SMOS main store as long as the kernel is running. However, if the kernel panics, a physical device fails, or power recycles, the main store has to be restored when the kernel re-starts up; otherwise, object persistency is only limited to the lifetime of the kernel. To extend object persistency, we provide a backing store as we described in Chapter 3. We implemented the backing store on fixed disk as an asynchronous mirror of the main store. This backing store is maintained by a daemon which is started by the same database management utility that installs the main store. The daemon also attaches the shared memory utility segment and periodically checks for the active transaction count (transactionCount in class WaitForGraph). If there is no active transactions, the daemon starts copying the main store to a temporary holding area; if a new transaction starts before it finishes[15], it will discard this temporary copy, otherwise, the temporary copy is renamed to the name of the backing store.

The major advantage of such an approach is that recovery never competes with transactions for system resources. Because there is no log for the main store, a transaction does not need to wait to finish writing its data on a disk based log file before it commits, performance is greatly improved. However, this advantage sometimes is also a disadvantage, because it creates two holes in the recovery. The fist one is when failure occurs after a transaction commits but before all the data has been moved to the main store. In this case a commit is not actually made. In another case, if we delay the commit until after the data is moved to main store, the state of the database may be inconsistent if failure occurs during moving the data. The second hole is when failure, which can cause

[15] To be precise, that is when the transaction count becomes none zero in the next check.

kernel restart, occurs after a successfully commit but before the backing store is updated. In this case the update of the main store after the last mirroring is lost.

Because our target user of SMOS are real-time applications that are dealing with high speed data with time constraint, a fully featured recovery is not necessary and can only degrade the system performance. Despite of this, we still need to carefully document the recovery feature of SMOS.

4.2.4 A Process Diagram

To summarize this chapter, a process diagram is presented in *Figure Chapter 4 .7* to demonstrate how SMOS works within Open OODB.

Before SMOS can be used, the system has to be initialized. This initialization is done by running a utility program with root UID. The root UID is required for locking the main store and the other shared memory segments in the main memory (preventing paging) as well as for setting protection mode to prevent unauthorized access[16]. The utility process maps and locks the backing store, type table, and name table in the memory, in addition it also creates a fresh utility segment[17] which will be used for supporting various database management. Before this utility process exits, it creates a daemon process that is responsible for mirroring the main store to the backing store. An application can start to use Open OODB/SMOS after the utility process is successfully returned. Because Open OODB seamlessly extends C++, any application written in C++ can be easily modified to use SMOS. To do so, as shown in *Figure Chapter 4 .7*, an application only needs to instanciate the class OODB, and the rest happens behind the scene. The conditional

[16] Further security is archived by using a password protection scheme on instanciating the OODB interface class, the door to the main store. Finer-grained protection, such as at object level, is too costly, therefore, we eliminated its implementation.

[17] The utility segment current only contains transaction wait-for graph, but it can also keep other information when additional utilities are added.

thread, though branches out from the application process, it is set off automatically to check for possible deadlocks as we described earlier.

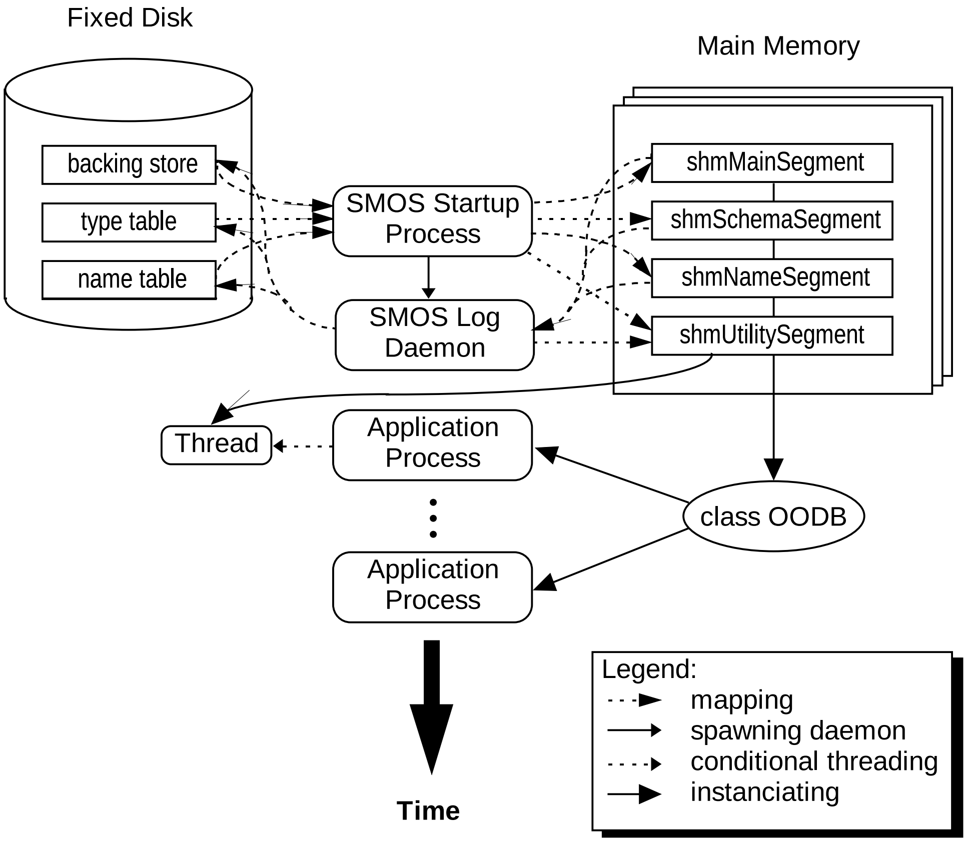


Figure Chapter 4 .7 SMOS Process Diagram

Chapter 5

THE PERFORMANCE EVALUATION

5.1 Overview

In this chapter we describe the timing experiments and present the test results. For comparison, we attempted all the tests on Open OODB/SMOS and Open OODB/Exodus in parallel; the tests include raw performance measurement with concurrency control disabled and scalability measurement under various controlled contentions.

The platform used for the tests is a Sun SPARC-based workstation running Solaris 2.5. *Table Chapter 5 .1* lists some of the hardware and kernel configurations.

Processor Speed	*100 MHz*
Memory Size	64 Mb
shminfo_shmmax	*4194304*
shminfo_shmseg	16

Table Chapter 5 .1 Test Platform Hardware Configurations

To eliminate fragmentation, fresh partitions are used when testing against OpenOODB/Exodus, whereas reinitialized shared memory segments are used when testing against OpenOODB/SMOS. The OS of the workstation was brought down in single user mode while performing the tests. We also stopped all other unnecessary system daemons that may preempt our tests. In addition, all the tests are launched with the same priority.

5.2 Raw Performance Tests

5.2.1 Test Metrics

In this experiment, our target test metric is the access speed of persistent object and the transaction throughput (number of transactions per unit time). For comparison purposes, we idealized the test conditions by introducing only one variable, the size of the object. In addition, we disabled concurrent transactions by launching only one transaction at a time. Our test suite includes 5 individual tests. The first test measures the time used in setting up the communications between the application and the OpenOODB system; because it is the time used to instanciate the OODB class which coordinates various policy managers, the time snapshots were taken just before and after the instanciation of OODB class. The second test measures time used by an empty transaction, this will provide information about the overhead in setting up a transaction. The remaining 3 tests use the same test code to measure the differences in persistent object access speed and transaction throughput between OpenOODB/Exodus and OpenOODB/SMOS (direct and indirect addressing); and as shown below, only one persistent object and two operations are involved in a transaction:

```
begin transaction
    fetch object
    touch object
commit or abort transaction
```

Throughout the tests, each single test case was repeated 100 times to take the average. The reliability of the tests is verified by some simple statistical analysis, such as standard deviation and confidence interval (Appendix B).

5.2.2 Results

In the first test, the mean time used to instanciate the class OODB is 0.475251 seconds in OpenOODB/Exodus and 0.015363 seconds in OpenOODB/SMOS (Appendix B.1). Because no persistent addressing operations are involved, the significant performance advantage comes from a single source, the elimination of the server and client communication overhead.

In the second test, the mean time used in an empty committed transaction is 0.042918 seconds in OpenOODB/Exodus and 0.006079 seconds in OpenOODB/SMOS; for an empty aborted transaction the time is cut down to 0.021429 seconds and 0.004132 seconds, respectively (Appendix B.2). Again, the performance advantage in OpenOODB/SMOS was obtained only from the elimination of client and server communication overhead.

The data from the next three tests is listed in Appendix B-3 and is plotted in *Figure Chapter* 5 .1; the result demonstrates that the transaction time used by OpenOODB/SMOS is close to one order of magnitude less than that of in OpenOODB/Exodus, and the direct addressing scheme also performs better than the indirect addressing scheme. Based on the data collected, we also calculated the transaction throughput (*Table Chapter* 3 .1). In these tests, because persistent addressing operations are involved, the performance gain in OpenOODB/SMOS is from a combination of elimination of communication overhead and object store main memory residency. Because the tests only allow one persistent addressing operation (object fetching) in each transaction, we can expect a larger performance difference when there are more than one persistent objects are fetched in a transaction.

	OpenOODB/Exodus	OpenOODB/SMOS (indirect addressing)	OpenOODB/SMOS (direct addressing)
transactions/second	18-21	63-116	137-148

Table Chapter 5 .2 Comparison of Calculated Transaction Throughput

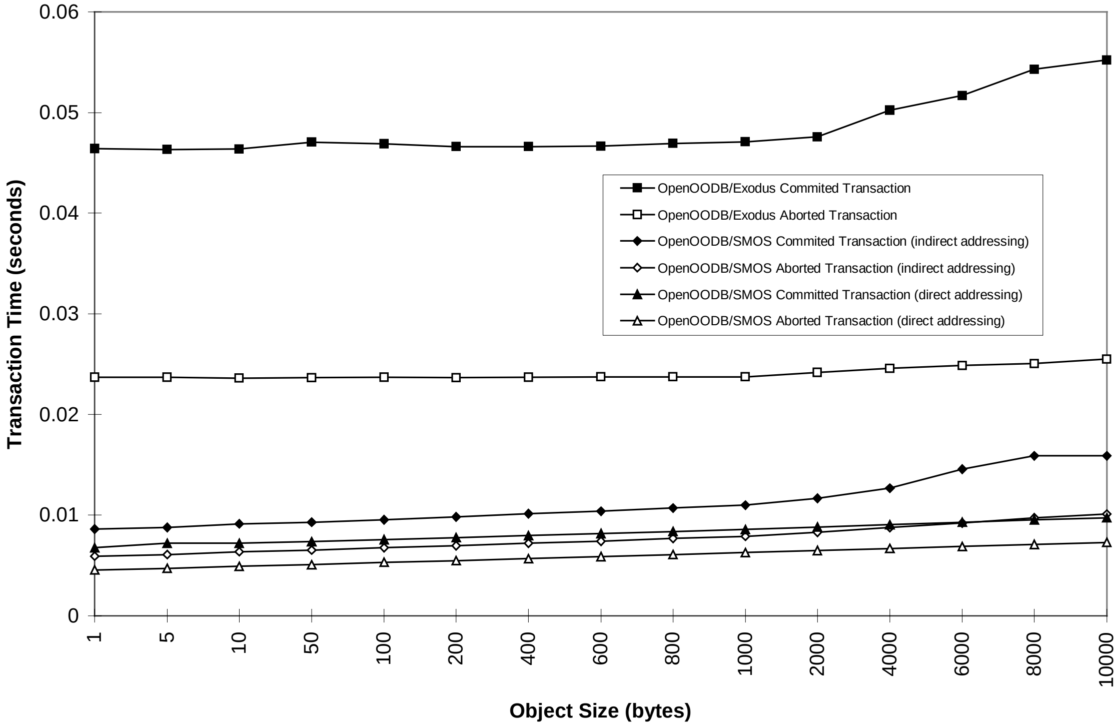


Figure Chapter 5 .1 Raw Performance Test Results

5.3 Scalability Tests

5.3.1 Test Metrics

To test how the addressing scheme may affect scalability under various contentions we designed another test. In this test the scalability metric is measured by the rate of transaction abort that is caused by deadlocks, a higher rate of transaction abort represents a worse scalability. The contention of each of these tests is controlled by specifying the number of concurrent transactions, the number of objects in the object store for which a transaction can randomly access, and the time interval between two consecutive persistent object accessed in a transaction. A higher contention environment is represented by more concurrent transactions, more objects accessed in a transaction, and longer transaction duration. Unfortunately, because OpenOODB/Exodus (v1.0) fails when two or more transactions try to mount the storage group at the same time, we could not apply the same test to it for comparison. Therefore, this test only compares the direct and indirect addressing scheme of OpenOODB/SMOS.

5.3.2 Results

There are a total of 550 (5 x 10 x 11) test cases in the test. Each has a different combination of the control factors we mentioned earlier: (1) the number of concurrent transactions range from 2 to 10 with an increment of 2, (2) the number of objects for random access in a transaction range from 5 to 50 with an increment of 5, and (3) the time interval between two consecutive persistent object accesses ranges from 0 to 50 milliseconds with an increment of 5 milliseconds. The same test cases are applied to both indirect addressing and direct addressing schemes for comparison, and the complete data is listed in Appendix B.4. For easy interpretation, we projected the 3-D data in Appendix B.4 separately on each of the control factors and plotted them in *Figure Chapter* 5 .2.

In each of the diagrams shown in *Figure Chapter* 5 .2, the rate of accumulated transaction aborts covers the full variation range of the other two control factors. Overall, we observed a 6% decrease in transaction abort rates in the direct addressing scheme compared to the indirect addressing scheme. This increased scalability is due to the shortened lock holding time under the direct addressing scheme, because improved persistent object access speed can reduce transaction time as demonstrated in Section 5.2. However, we also realized that the advantage of direct addressing was not fully expressed in this test because of the limitation of our concurrency control protocol (under a strict two-phase locking protocol, even though the lock on an object is no longer necessary after it has be accessed, we can only release the lock after commit time; if waiting for the release of this lock means a deadlock, no matter how fast this object is addressed, this deadlock cannot be avoided if commit comes after the waiting).

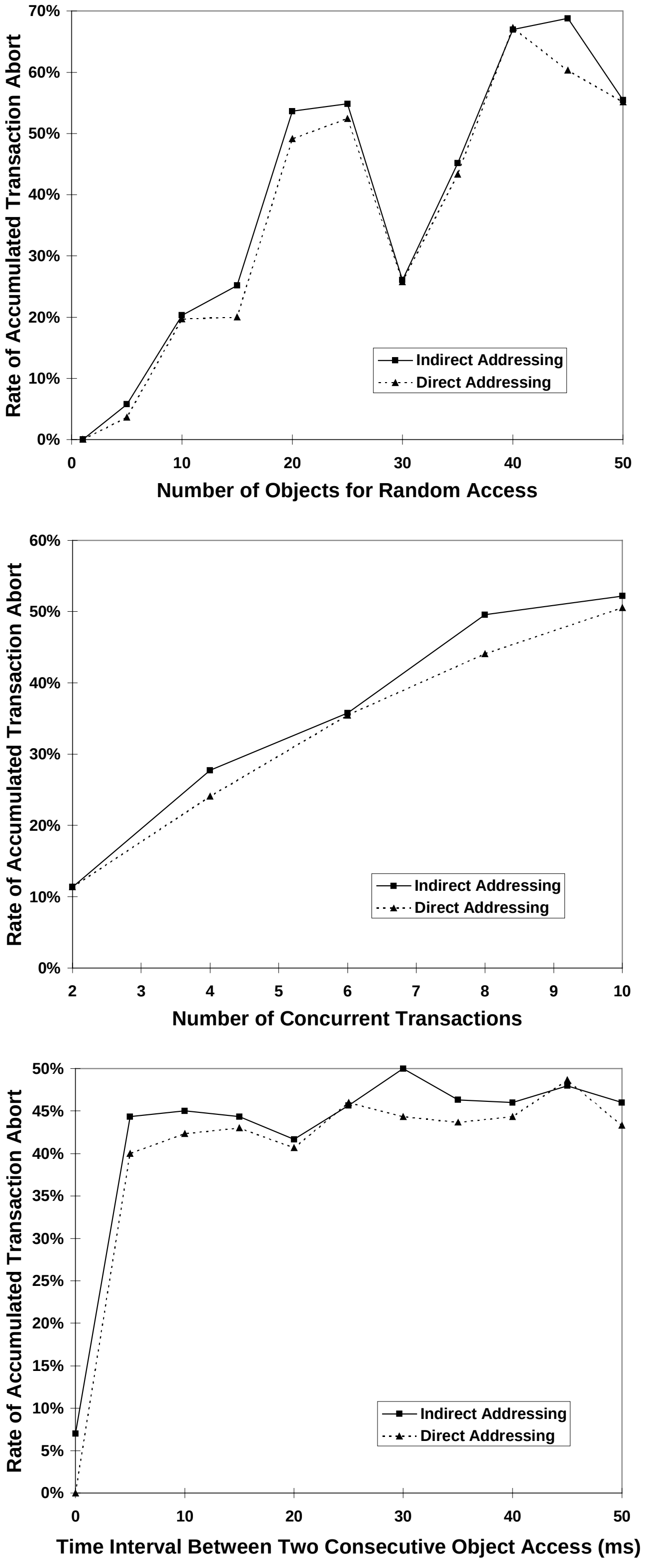


Figure Chapter 5 .2 Scalability Test Results

Chapter 6

CONCLUSION

In this thesis research we have explored several ways to satisfy the performance requirement of a high speed real-time database application. Our approaches have been demonstrated in the design and implementation of a high-performance object store, some of them are controversial and have not been found in any literature. In this chapter, we will summarize what we have achieved in this thesis project, describe our research contributions, and project our future work.

6.1 Thesis Summary

In Chapter 3, we laid out a set of functional and system requirements (*Table Chapter* 3 .1 and *Table* Chapter 3 .2) for SMOS. Our design and implementation presented in Chapter 3 and 4 have demonstrated that these requirements have been fulfilled:

Functional Requirements

Object Model Independency: We managed to keep SMOS from being tied to any particular object model. Other than providing persistent object storage service, SMOS does not assume any other responsibilities that may introduce model dependency.

Multi-Level Object Persistency: We maintained two persistent levels in SMOS; the main store is memory-resident in order to provide greater performance, the backing store is disk-based and supports extended persistency beyond the main memory.

Guaranteed Persistent Object Access Time: We were able to bound the time used to store or retrieve a persistent object by utilizing the memory-resident nature as well as hash-based algorithms; the former eliminates I/O blocking uncertainties and the later delivers constant operation time.

Adjustable System Performance: We provided two addressing schemes which can be chosen at run-time; because these two schemes have different functional and performance emphasis, they can be used to dynamically adjust system performance. In addition, we can also turn on and off concurrency control in SMOS for the same purpose.

Concurrent Persistent Object Access: We provided each persistent object a metadata in which lock as well as other useful information are kept; such object level locking enables controlled concurrent access to an object as well as increased number of concurrent transactions.

System Requirements

Performance: We satisfied the performance requirement through the design of a client-only architecture, the implementation of a direct addressing scheme, and the trade-off of some of the traditional database features.

Portability, Reusability, Extendibility, and Flexibility: We avoided making any platform dependent calls to support portability; we implemented SMOS in an object-oriented fashion to support reusability and extendibility; and we provided flexible configurations for addressing, concurrency control, and recovery.

The performance evaluation presented in Chapter 5 further demonstrated that we have achieved the goal of this thesis project and that the measures used to achieve our goal are indeed very effective. By replacing Exodus with SMOS in OpenOODB, we observed a more than 30 times higher performance in setting up the communications between the application and the OpenOODB system (Appendix B.1), and about an 8 times higher transaction throughput with direct addressing in the worst case.

6.1 Research Contribution

Our main research contribution in this project is the actual implementation of a high-performance main memory resident object store. This object store is in fully working order and can be easily configured to satisfy different application requirements. Unlike previous database systems which are almost universally based on a client-server architecture, our object store features an unique client-only architecture; this enables us to completely eliminate the communication overhead, one of the major sources of performance barrier, between the server and client. Therefore, this unique client-only architecture design is another research contribution in this thesis. In addition, we also experimented with a new persistent object addressing scheme which allows applications to operate directly on a persistent object. By trading off recoverability, this approach unifies persistent address space with the program transient address space and thus eliminates object moving between the database cache and persistent store, further improving the performance level.

6.2 Future Work

As revealed by our test, in a high contention environment, though a high-performance addressing scheme can help to improve the scalability, transaction management seems even more important. Because of the strict two-phase locking technique used in our implementation, locks are often held longer than necessary; as a result the possibility of deadlocks is relatively high. One possible improvement is to reduce the number of persistent objects accessed by a single transaction. In our test, when only one persistent object access is allowed in a transaction, no transaction abort was found in the same test ranges (*Figure Chapter* 5 .2). However, because this approach requires additional transactions to complete the same amount of persistent object access, performance can be affected because of the overhead involved in setting up the transaction. Further, this approach imposes restrictions at the application level and still cannot totally eliminate deadlocks. Therefore, the best solution is to find a concurrency control protocol which can is deadlock-free. Parallel to this thesis research, the priority ceiling protocol is being investigated by Michael Squadrito in the Real-Time Research Group at the University of Rhode Island (Squadrito, 1996). This protocol has the advantage of eliminating deadlocks and bounding the blocking time of high priority transactions to no more than one transaction, and can be a candidate for our future concurrency control protocol.

The current design and implementation limits our object store to a standalone environment. This limitation is because we intend to eliminate the network overhead. However, a database application often needs to access an object store that resides on a different workstation that is possibly running a different operating system. Supporting such networked access and heterogeneous environments is a challenge because it may impact our basic design structure. As a continued study, a research project has been proposed in the Real-Time Research Group at the University of Rhode Island to investigate possible solutions using Common Object Request Broker Architecture (CORBA).

Our performance test suite did not include a pointer traverse timing test. It has been our desire to implement an OO7 benchmark test in this thesis project, because doing so would enable us to compare the performance of OpenOODB/SMOS with several commercial OODBMS. Pointer traverse is tested extensively in OO7 by using object relationships. However, because OpenOODB directly uses the same C++ object model for its database object model, we could not implement OO7 without extending the object model to support object relationships. We attempted such extension with limited time, and found it is a rather complex task that is beyond the scope of this thesis project.

LIST OF REFERENCES

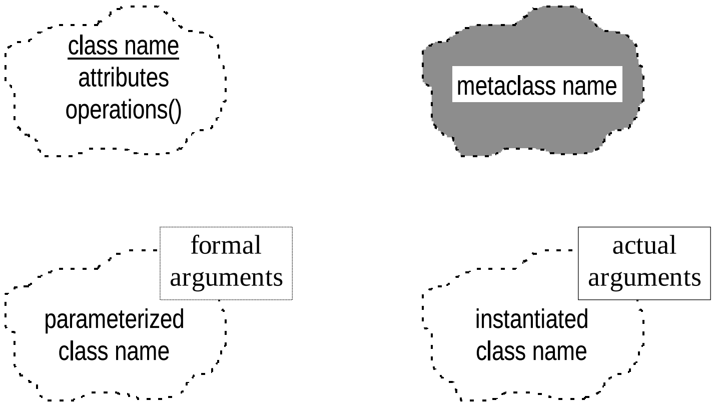
Elmasri, R. and Navathe, S. B. (1994) “Fundamentals of Database Systems”, Second Edition, Benjamin/Cummings, 1994.

Loomis, M. E. S. (1995) “Object Databases: The Essentials”, Addison-Wesley, 1995.

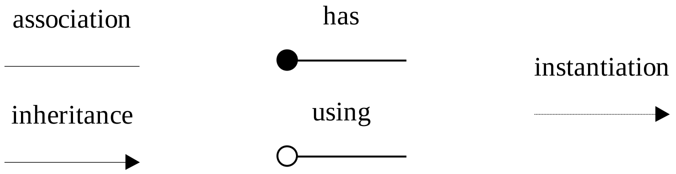
Appendix A

BOOCH OOD NOTATIONS[18]

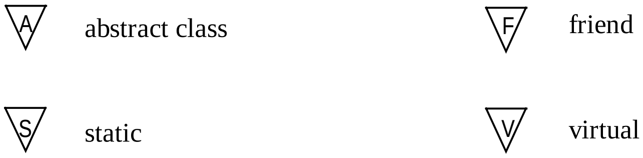
Class Icons



Class Relationships



Properties



[18] A complete list can be found in [Booch 94].

Appendix B

Appendix B.1 Class OODB Instanciation Timing Test Results*

Test Cases	# of repeats	mean (sec.)	min. (sec.)	max. (sec.)	st.dev. (sec.)	95% conf. interval (sec.)	(sec.)
C_1_1	100	0.475251	0.472599	0.479016	0.000774	0.475243	0.475258
C_1_2	100	0.015363	0.015139	0.017152	0.000252	0.015361	0.015364

*C_1_1:	Test with OpenOODB/Exodus
C_1_2:	Test with OpenOODB/SMOS

Appendix B.2 Empty Transaction Timing Test Results[*]

Test	# of	mean	min.	max.	st.dev.	95% conf. interval	
Cases	repeats	(sec.)	(sec.)	(sec.)	(sec.)	(sec.)	(sec.)
C_2_1_1	100	0.042918	0.042473	0.044806	0.000347	0.042916	0.042920
C_2_1_2	100	0.021429	0.021260	0.022129	0.000129	0.021428	0.021430
C_2_2_1	100	0.006079	0.005888	0.006981	0.000149	0.006078	0.006080
C_2_2_2	100	0.004132	0.004038	0.004832	0.000109	0.004131	0.004133

[*] C_2_1_1: OpenOODB/Exodus Empty Transaction (Committed) Timing Test
C_2_1_2: OpenOODB/Exodus Empty Transaction (Aborted) Timing Test
C_2_2_1: OpenOODB/SMOS Empty Transaction (Committed) Timing Test
C_2_2_2: OpenOODB/SMOS Empty Transaction (Aborted) Timing Test

Appendix B.3 None Empty Transaction Timing Test Results*

Test	# of	object	mean	min.	max.	st.dev.	95 conf. interval	
Cases	repeats	size	(sec.)	(sec.)	(sec.)	(sec.)	(sec.)	(sec.)
		(bytes)						
C_3_1	100	1	0.046427	0.045999	0.050855	0.000655	0.046423	0.046431
C_3_1	100	5	0.046338	0.044894	0.047541	0.000311	0.046336	0.046340
C_3_1	100	10	0.046355	0.044928	0.047191	0.000262	0.046354	0.046357
C_3_1	100	50	0.047057	0.046135	0.048236	0.000323	0.047055	0.047059
C_3_1	100	100	0.046871	0.045915	0.048247	0.000453	0.046868	0.046873
C_3_1	100	200	0.046604	0.046120	0.047386	0.000302	0.046602	0.046606
C_3_1	100	400	0.046623	0.045383	0.047977	0.000374	0.046621	0.046626
C_3_1	100	600	0.046643	0.045595	0.047972	0.000322	0.046641	0.046645
C_3_1	100	800	0.046921	0.045746	0.048689	0.000477	0.046918	0.046924
C_3_1	100	1000	0.047088	0.046242	0.048102	0.000406	0.047085	0.047090
C_3_1	100	2000	0.047571	0.046775	0.048536	0.000352	0.047569	0.047573
C_3_1	100	4000	0.050219	0.049535	0.051714	0.000526	0.050216	0.050223
C_3_1	100	6000	0.051665	0.050318	0.054974	0.000924	0.051659	0.051671
C_3_1	100	8000	0.054282	0.052429	0.056354	0.000887	0.054276	0.054287
C_3_1	100	10000	0.055233	0.051267	0.059902	0.001159	0.055226	0.055240
C_3_2	100	1	0.023672	0.023184	0.051303	0.002807	0.023655	0.023690
C_3_2	100	5	0.023683	0.023203	0.045799	0.002254	0.023669	0.023697
C_3_2	100	10	0.023620	0.023216	0.046498	0.002315	0.023605	0.023634
C_3_2	100	50	0.023631	0.023214	0.046141	0.002278	0.023616	0.023645
C_3_2	100	100	0.023703	0.023214	0.046075	0.002311	0.023689	0.023718
C_3_2	100	200	0.023650	0.023227	0.046462	0.002309	0.023636	0.023665
C_3_2	100	400	0.023677	0.023263	0.046027	0.002263	0.023663	0.023692
C_3_2	100	600	0.023714	0.023265	0.046709	0.002328	0.023699	0.023728
C_3_2	100	800	0.023733	0.023275	0.046366	0.002292	0.023718	0.023747
C_3_2	100	1000	0.023737	0.023308	0.045807	0.002233	0.023723	0.023751
C_3_2	100	2000	0.024167	0.023718	0.047723	0.002387	0.024152	0.024182
C_3_2	100	4000	0.024572	0.024039	0.051805	0.002758	0.024555	0.024590
C_3_2	100	6000	0.024847	0.024257	0.049981	0.002558	0.024831	0.024864
C_3_2	100	8000	0.025067	0.024542	0.051786	0.002717	0.025049	0.025084
C_3_2	100	10000	0.025493	0.024927	0.053242	0.002822	0.025476	0.025511
C_4_1	100	1	0.008597	0.008143	0.010776	0.000378	0.008595	0.008599
C_4_1	100	5	0.008785	0.007592	0.009614	0.000251	0.008783	0.008786
C_4_1	100	10	0.009115	0.007830	0.009670	0.000211	0.009114	0.009117
C_4_1	100	50	0.009272	0.008136	0.009822	0.000210	0.009271	0.009273
C_4_1	100	100	0.009540	0.008517	0.010033	0.000217	0.009539	0.009541
C_4_1	100	200	0.009799	0.008556	0.010348	0.000218	0.009798	0.009801
C_4_1	100	400	0.010133	0.008875	0.010581	0.000226	0.010131	0.010134
C_4_1	100	600	0.010374	0.009366	0.010894	0.000213	0.010373	0.010375
C_4_1	100	800	0.010692	0.009403	0.011109	0.000232	0.010691	0.010693
C_4_1	100	1000	0.010997	0.009449	0.011467	0.000241	0.010995	0.010998
C_4_1	100	2000	0.011646	0.010353	0.013549	0.000451	0.011643	0.011649
C_4_1	100	4000	0.012673	0.010592	0.013486	0.000424	0.012671	0.012676
C_4_1	100	6000	0.014568	0.012309	0.015726	0.000624	0.014565	0.014572
C_4_1	100	8000	0.015872	0.014562	0.017344	0.000720	0.015868	0.015877
C_4_1	100	10000	0.015868	0.014206	0.017781	0.000878	0.015862	0.015873
C_4_2	100	1	0.005911	0.005562	0.011026	0.000570	0.005908	0.005915
C_4_2	100	5	0.006084	0.005760	0.007649	0.000259	0.006082	0.006086

C_4_2	100	10	0.006357	0.006101	0.007858	0.000196	0.006356	0.006358
C_4_2	100	50	0.006519	0.006176	0.008087	0.000205	0.006518	0.006521
C_4_2	100	100	0.006738	0.006483	0.008354	0.000203	0.006737	0.006739
C_4_2	100	200	0.006963	0.006593	0.008660	0.000230	0.006962	0.006965
C_4_2	100	400	0.007218	0.006876	0.008771	0.000215	0.007217	0.007220
C_4_2	100	600	0.007418	0.007004	0.009039	0.000223	0.007416	0.007419
C_4_2	100	800	0.007698	0.007369	0.009521	0.000243	0.007697	0.007700
C_4_2	100	1000	0.007892	0.007580	0.009318	0.000197	0.007891	0.007893
C_4_2	100	2000	0.008274	0.008017	0.009957	0.000223	0.008272	0.008275
C_4_2	100	4000	0.008782	0.008479	0.010462	0.000229	0.008780	0.008783
C_4_2	100	6000	0.009228	0.008857	0.011319	0.000305	0.009226	0.009230
C_4_2	100	8000	0.009736	0.009268	0.011777	0.000307	0.009734	0.009738
C_4_2	100	10000	0.010102	0.009734	0.012627	0.000396	0.010099	0.010104
C_5_1	100	1	0.006751	0.006350	0.008994	0.000302	0.006749	0.006753
C_5_1	100	5	0.007187	0.006807	0.007798	0.000237	0.007186	0.007189
C_5_1	100	10	0.007212	0.006840	0.008114	0.000276	0.007210	0.007214
C_5_1	100	50	0.007345	0.007093	0.008073	0.000137	0.007344	0.007346
C_5_1	100	100	0.007569	0.007246	0.008130	0.000153	0.007568	0.007570
C_5_1	100	200	0.007772	0.007417	0.008143	0.000125	0.007771	0.007773
C_5_1	100	400	0.007961	0.007635	0.008517	0.000153	0.007960	0.007962
C_5_1	100	600	0.008181	0.007875	0.008764	0.000166	0.008180	0.008182
C_5_1	100	800	0.008363	0.008104	0.008791	0.000136	0.008363	0.008364
C_5_1	100	1000	0.008566	0.008275	0.009191	0.000142	0.008565	0.008567
C_5_1	100	2000	0.008826	0.008543	0.009173	0.000152	0.008825	0.008827
C_5_1	100	4000	0.009054	0.008682	0.009861	0.000145	0.009053	0.009055
C_5_1	100	6000	0.009294	0.008988	0.009727	0.000153	0.009293	0.009295
C_5_1	100	8000	0.009523	0.009221	0.010163	0.000155	0.009522	0.009524
C_5_1	100	10000	0.009716	0.009448	0.010376	0.000164	0.009715	0.009717
C_5_2	100	1	0.004539	0.004317	0.008910	0.000456	0.004536	0.004542
C_5_2	100	5	0.004702	0.004533	0.007357	0.000283	0.004701	0.004704
C_5_2	100	10	0.004902	0.004722	0.007062	0.000235	0.004901	0.004904
C_5_2	100	50	0.005084	0.004905	0.007389	0.000256	0.005082	0.005085
C_5_2	100	100	0.005301	0.005130	0.007656	0.000257	0.005300	0.005303
C_5_2	100	200	0.005479	0.005291	0.007639	0.000233	0.005478	0.005481
C_5_2	100	400	0.005683	0.005500	0.008136	0.000261	0.005681	0.005684
C_5_2	100	600	0.005891	0.005701	0.008085	0.000241	0.005889	0.005892
C_5_2	100	800	0.006087	0.005889	0.008194	0.000238	0.006086	0.006089
C_5_2	100	1000	0.006282	0.006097	0.008474	0.000240	0.006280	0.006283
C_5_2	100	2000	0.006479	0.006287	0.008732	0.000241	0.006478	0.006481
C_5_2	100	4000	0.006677	0.006467	0.009229	0.000274	0.006675	0.006679
C_5_2	100	6000	0.006884	0.006674	0.009116	0.000244	0.006882	0.006885
C_5_2	100	8000	0.007077	0.006876	0.009464	0.000256	0.007075	0.007078
C_5_2	100	10000	0.007287	0.007084	0.009786	0.000267	0.007285	0.007288

*C_3_1: OpenOODB/Exodus Transaction (Committed) Timing Test
C_3_2: OpenOODB/Exodus Transaction (Aborted) Timing Test
C_4_1: OpenOODB/SMOS Indirect Addressing Transaction (Committed) Timing Test
C_4_2: OpenOODB/SMOS Indirect Addressing Transaction (Aborted) Timing Test
C_5_1: OpenOODB/SMOS direct Addressing Transaction (Committed) Timing Test
C_5_2: OpenOODB/SMOS direct Addressing Transaction (Abort) Timing Test

Appendix B.4 OpenOODB/SMOS Contention Test Results*

# of	# of	time	# of aborts	
objects	txn	interval	I	D
5	2	0	0	0
5	2	5	0	0
5	2	10	0	0
5	2	15	0	0
5	2	20	0	0
5	2	25	0	0
5	2	30	0	0
5	2	35	0	0
5	2	40	0	0
5	2	45	0	0
5	2	50	0	0
5	4	0	0	0
5	4	5	0	0
5	4	10	0	0
5	4	15	0	0
5	4	20	0	0
5	4	25	0	0
5	4	30	0	0
5	4	35	0	0
5	4	40	0	0
5	4	45	0	0
5	4	50	0	0
5	6	0	0	0
5	6	5	0	0
5	6	10	0	0
5	6	15	0	0
5	6	20	0	0
5	6	25	0	0
5	6	30	0	0
5	6	35	0	0
5	6	40	0	0
5	6	45	0	0
5	6	50	0	0
5	8	0	0	0
5	8	5	3	0
5	8	10	1	0
5	8	15	0	0
5	8	20	0	0
5	8	25	2	1
5	8	30	1	0
5	8	35	2	1
5	8	40	0	1
5	8	45	0	1
5	8	50	0	1
5	10	0	0	0
5	10	5	0	0
5	10	10	0	1
5	10	15	3	2
5	10	20	1	1
5	10	25	1	0
5	10	30	0	0
5	10	35	2	1
5	10	40	2	0
5	10	45	1	1
5	10	50	0	1
10	2	0	0	0
10	2	5	1	0
10	2	10	0	0
10	2	15	0	0
10	2	20	0	0
10	2	25	0	0
10	2	30	0	0
10	2	35	0	1
10	2	40	0	0
10	2	45	1	0
10	2	50	0	0
10	4	0	0	0
10	4	5	0	0
10	4	10	1	0
10	4	15	1	0
10	4	20	0	1
10	4	25	1	0
10	4	30	1	1
10	4	35	0	0
10	4	40	1	0
10	4	45	1	1
10	4	50	1	1
10	6	0	0	0
10	6	5	0	0
10	6	10	0	0
10	6	15	1	0
10	6	20	0	2
10	6	25	0	1
10	6	30	1	1

10	6	35	0	1
10	6	40	1	2
10	6	45	1	1
10	6	50	1	1
10	8	0	1	0
10	8	5	2	1
10	8	10	2	0
10	8	15	1	1
10	8	20	3	0
10	8	25	1	1
10	8	30	4	1
10	8	35	1	4
10	8	40	1	2
10	8	45	2	3
10	8	50	2	2
10	10	0	1	0
10	10	5	2	1
10	10	10	3	4
10	10	15	4	3
10	10	20	1	4
10	10	25	4	4
10	10	30	6	6
10	10	35	3	5
10	10	40	2	2
10	10	45	5	2
10	10	50	2	5
15	2	0	0	0
15	2	5	0	0
15	2	10	0	0
15	2	15	1	0
15	2	20	0	0
15	2	25	0	1
15	2	30	0	1
15	2	35	0	0
15	2	40	1	0
15	2	45	0	0
15	2	50	1	0
15	4	0	0	0
15	4	5	1	0
15	4	10	1	0
15	4	15	1	0
15	4	20	1	0
15	4	25	0	0
15	4	30	0	0
15	4	35	0	0
15	4	40	1	1
15	4	45	1	0
15	4	50	1	0
15	6	0	0	0
15	6	5	0	0
15	6	10	0	1
15	6	15	0	2
15	6	20	0	0
15	6	25	1	1
15	6	30	0	0
15	6	35	1	0
15	6	40	2	1
15	6	45	0	1
15	6	50	1	0
15	8	0	0	0
15	8	5	2	0
15	8	10	3	2
15	8	15	4	2
15	8	20	1	2
15	8	25	3	2
15	8	30	5	2
15	8	35	2	4
15	8	40	4	4
15	8	45	5	4
15	8	50	2	3
15	10	0	0	0
15	10	5	4	3
15	10	10	4	2
15	10	15	4	4
15	10	20	4	4
15	10	25	3	5
15	10	30	3	2
15	10	35	4	0
15	10	40	4	3
15	10	45	4	4
15	10	50	3	5
20	2	0	0	0
20	2	5	1	1
20	2	10	0	1
20	2	15	0	0
20	2	20	0	1
20	2	25	1	1
20	2	30	1	1
20	2	35	1	0
20	2	40	1	0
20	2	45	1	0
20	2	50	1	1
20	4	0	0	0
20	4	5	2	1
20	4	10	3	2
20	4	15	1	2
20	4	20	1	2
20	4	25	2	2
20	4	30	2	2
20	4	35	3	1

20	4	40	3	2
20	4	45	2	2
20	4	50	3	1
20	6	0	0	0
20	6	5	3	4
20	6	10	3	5
20	6	15	3	3
20	6	20	5	4
20	6	25	5	3
20	6	30	4	4
20	6	35	4	4
20	6	40	3	4
20	6	45	5	4
20	6	50	5	4
20	8	0	2	0
20	8	5	6	4
20	8	10	4	5
20	8	15	6	4
20	8	20	4	4
20	8	25	5	4
20	8	30	4	3
20	8	35	4	4
20	8	40	4	6
20	8	45	4	4
20	8	50	5	2
20	10	0	0	0
20	10	5	6	6
20	10	10	6	6
20	10	15	6	5
20	10	20	6	6
20	10	25	5	8
20	10	30	7	7
20	10	35	7	5
20	10	40	5	6
20	10	45	5	7
20	10	50	7	4
25	2	0	0	0
25	2	5	0	0
25	2	10	0	0
25	2	15	0	0
25	2	20	0	0
25	2	25	0	0
25	2	30	1	0
25	2	35	0	1
25	2	40	0	0
25	2	45	0	1
25	2	50	0	1
25	4	0	0	0
25	4	5	2	2
25	4	10	2	2
25	4	15	1	2
25	4	20	1	2
25	4	25	1	1
25	4	30	3	2
25	4	35	2	1
25	4	40	2	2
25	4	45	3	2
25	4	50	2	2
25	6	0	0	0
25	6	5	3	5
25	6	10	2	1
25	6	15	3	2
25	6	20	5	2
25	6	25	3	4
25	6	30	3	2
25	6	35	3	4
25	6	40	5	5
25	6	45	4	4
25	6	50	3	3
25	8	0	0	0
25	8	5	5	5
25	8	10	6	5
25	8	15	4	6
25	8	20	5	5
25	8	25	4	5
25	8	30	7	6
25	8	35	6	6
25	8	40	5	6
25	8	45	6	5
25	8	50	7	6
25	10	0	1	0
25	10	5	7	5
25	10	10	8	6
25	10	15	7	8
25	10	20	7	7
25	10	25	6	7
25	10	30	8	8
25	10	35	8	6
25	10	40	7	5
25	10	45	7	7
25	10	50	6	6
30	2	0	0	0
30	2	5	0	0
30	2	10	0	0
30	2	15	0	0
30	2	20	0	0
30	2	25	0	0
30	2	30	0	0
30	2	35	0	0
30	2	40	0	0
30	2	45	0	0
30	2	50	0	0

30	4	0	0	0
30	4	5	0	0
30	4	10	0	0
30	4	15	0	0
30	4	20	0	0
30	4	25	0	0
30	4	30	0	0
30	4	35	0	0
30	4	40	0	0
30	4	45	0	0
30	4	50	0	0
30	6	0	0	0
30	6	5	0	0
30	6	10	0	0
30	6	15	1	0
30	6	20	0	0
30	6	25	1	1
30	6	30	0	0
30	6	35	0	0
30	6	40	0	0
30	6	45	1	0
30	6	50	0	0
30	8	0	0	0
30	8	5	2	1
30	8	10	3	4
30	8	15	4	3
30	8	20	3	3
30	8	25	3	2
30	8	30	6	2
30	8	35	3	3
30	8	40	3	3
30	8	45	3	5
30	8	50	4	2
30	10	0	0	0
30	10	5	4	5
30	10	10	5	6
30	10	15	5	7
30	10	20	3	4
30	10	25	5	6
30	10	30	6	6
30	10	35	5	6
30	10	40	4	5
30	10	45	5	5
30	10	50	7	6
35	2	0	0	0
35	2	5	0	0
35	2	10	0	0
35	2	15	0	0
35	2	20	0	0
35	2	25	0	0
35	2	30	0	0
35	2	35	0	0
35	2	40	0	0
35	2	45	0	0
35	2	50	0	0
35	4	0	0	0
35	4	5	0	2
35	4	10	2	0
35	4	15	1	2
35	4	20	2	1
35	4	25	2	2
35	4	30	0	1
35	4	35	2	2
35	4	40	1	1
35	4	45	2	2
35	4	50	1	1
35	6	0	1	0
35	6	5	3	2
35	6	10	3	2
35	6	15	2	3
35	6	20	2	1
35	6	25	3	4
35	6	30	3	3
35	6	35	3	2
35	6	40	2	2
35	6	45	1	3
35	6	50	2	3
35	8	0	0	0
35	8	5	6	2
35	8	10	5	4
35	8	15	4	6
35	8	20	3	5
35	8	25	5	5
35	8	30	4	4
35	8	35	5	4
35	8	40	5	4
35	8	45	6	6
35	8	50	3	3
35	10	0	0	0
35	10	5	6	8
35	10	10	6	5
35	10	15	7	6
35	10	20	6	5
35	10	25	7	6
35	10	30	6	6
35	10	35	7	7
35	10	40	6	6
35	10	45	7	7
35	10	50	7	5
40	2	0	0	0
40	2	5	0	1
40	2	10	0	0

40	2	15	1	0
40	2	20	0	0
40	2	25	0	1
40	2	30	0	1
40	2	35	1	1
40	2	40	1	0
40	2	45	1	1
40	2	50	0	1
40	4	0	0	0
40	4	5	2	2
40	4	10	2	2
40	4	15	1	3
40	4	20	2	2
40	4	25	1	1
40	4	30	2	2
40	4	35	2	2
40	4	40	2	2
40	4	45	1	3
40	4	50	3	2
40	6	0	4	0
40	6	5	4	4
40	6	10	2	5
40	6	15	4	4
40	6	20	4	4
40	6	25	5	4
40	6	30	5	4
40	6	35	4	5
40	6	40	5	5
40	6	45	5	5
40	6	50	4	5
40	8	0	0	0
40	8	5	7	7
40	8	10	5	7
40	8	15	6	6
40	8	20	6	7
40	8	25	7	6
40	8	30	7	6
40	8	35	7	6
40	8	40	7	6
40	8	45	6	7
40	8	50	7	7
40	10	0	1	0
40	10	5	9	9
40	10	10	9	9
40	10	15	9	7
40	10	20	9	9
40	10	25	9	9
40	10	30	9	9
40	10	35	9	8
40	10	40	8	8
40	10	45	8	9
40	10	50	8	8
45	2	0	0	0
45	2	5	0	0
45	2	10	1	1
45	2	15	1	0
45	2	20	1	0
45	2	25	1	0
45	2	30	1	1
45	2	35	1	0
45	2	40	1	0
45	2	45	0	0
45	2	50	0	1
45	4	0	0	0
45	4	5	3	0
45	4	10	3	2
45	4	15	1	2
45	4	20	2	2
45	4	25	2	2
45	4	30	2	2
45	4	35	3	2
45	4	40	3	2
45	4	45	2	3
45	4	50	3	2
45	6	0	0	0
45	6	5	5	3
45	6	10	4	4
45	6	15	4	5
45	6	20	4	3
45	6	25	5	5
45	6	30	5	4
45	6	35	4	4
45	6	40	4	4
45	6	45	4	4
45	6	50	4	4
45	8	0	6	0
45	8	5	6	6
45	8	10	6	6
45	8	15	7	5
45	8	20	7	6
45	8	25	5	5
45	8	30	5	5
45	8	35	6	7
45	8	40	7	7
45	8	45	6	5
45	8	50	6	4
45	10	0	3	0
45	10	5	9	8
45	10	10	9	9
45	10	15	9	8
45	10	20	7	7
45	10	25	9	9

45	10	30	8	9
45	10	35	8	8
45	10	40	7	7
45	10	45	8	9
45	10	50	9	7
50	2	0	0	0
50	2	5	0	1
50	2	10	0	0
50	2	15	0	0
50	2	20	1	0
50	2	25	0	1
50	2	30	0	1
50	2	35	0	0
50	2	40	0	1
50	2	45	0	0
50	2	50	0	0
50	4	0	0	0
50	4	5	1	2
50	4	10	2	2
50	4	15	1	1
50	4	20	1	1
50	4	25	1	0
50	4	30	1	1
50	4	35	1	0
50	4	40	1	0
50	4	45	2	2
50	4	50	1	3
50	6	0	0	0
50	6	5	4	5
50	6	10	3	4
50	6	15	2	2
50	6	20	3	4
50	6	25	5	3
50	6	30	4	3
50	6	35	3	4
50	6	40	4	5
50	6	45	3	4
50	6	50	4	3
50	8	0	0	0
50	8	5	6	6
50	8	10	7	5
50	8	15	5	5
50	8	20	6	5
50	8	25	6	7
50	8	30	7	6
50	8	35	5	5
50	8	40	7	6
50	8	45	7	5
50	8	50	5	6
50	10	0	1	0
50	10	5	6	8
50	10	10	9	7
50	10	15	7	8
50	10	20	8	6
50	10	25	7	8
50	10	30	8	8
50	10	35	7	6
50	10	40	6	7
50	10	45	8	7
50	10	50	7	8

*# of objects: the total number of objects that are available for transactions to access

# of txn: the number of concurrent transactions (these transactions start at exactly the same time)

# of aborts: the number of transaction aborts; I denotes indirect addressing scheme; D denotes direct addressing scheme

time interval: the time interval between two consecutive persistent objects access in a transaction

I: indirect addressing

D: direct addressing

BIBLIOGRAPHY

Abbott, R., Garcia-Molina, H. (1988) “Scheduling Real-Time Transactions: A Performance Evaluation”, Proceedings of the 14th Very Large Data Base Conference, August, 1988.

Abbott, R., Garcia-Molina, H. (1988) “Scheduling Real-Time Transactions with disk resident data”, Proceedings of the 14th Very Large Data Base Conference, August 1988.

Abbott, R., Garcia-Molina, H. (1990) “Scheduling I/O Requests with Deadlines: A Performance Evaluation”, Proceedings of the 11th Real-Time Systems Symposium, December 1990.

Buchmann, A. P., McCarthy, D. C., Hsu., M., Dayal, U. (1989) “Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Controls”, Proceedings of the 5th International Conference on Data Engineering, February 1989.

Booch, G. (1994) “Object-Oriented Analysis and Design with Applications”, Second Edition, Addison-Wesley, 1994.

Carey, M., DeWitt, D., Richardson, J. and Shekita, E. (1989) “Storage Management for Objects in EXODUS”, in Object-Oriented Concepts, Databases, and Applications, Addison-Wesley, 1989.

Carey, M., DeWitt, D, Naughton, J. (1993) “The OO7 Benchmark”, Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, DC, May 1993.

Carey., M. J., Franklin., M. J., and Zaharioudakis., M. (1994) “Fine-grained sharing in a page server OODBMS”, In Proceedings of the ACM SIGMOD International Conference on Management of Data, May 1994.

Cattell, R. G. G. (1994) “Object Data Management: Object-Oriented and Extended Relational Database Systems”, Addison-Wesley, 1994.

Cattell, R. G. G. (1996) “The Object Database Standard: ODMG-93”, Release 1.2, 1996.

Chen, S and Lin. K. (1990) “Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems”, Real-Time Systems, 2(4), December 1990.

Deux, O. (1991) “The O2 System”, Communications of the ACM, Volume. 34, Number 10, October. 1991.

DiPippo, L. C. (1995) “Object-Based Semantic Real-Time Concurrency Control”, Ph.D. Dissertation, University of Rhode Island, May 1995.

Huang. J., Stankovic. J., Towsley. D., and Ramamritham. K. (1990) “Real-Time Transaction Processing: Design, Implementation and Performance Evaluation”, Technical Report 90-43, University of Massachusetts, May 1990.

Huang, Jiandong, Stankovic, J., Ramamritham, K., and Towslwy, D. (1992) “On Using Priority Inheritance in Real-Time Database”, Proceedings of the 12th Real-Time Systems Symposium, December 1991.

Moss, J. (1992) “Working with Persistent Objects: To Swizzle or Not to Swizzle”, IEEE Transactions of Software Engineering, 18(8):657-673, August 1992.

Peckham, J, Wolfe, V. F., Prichard, J., and DiPippo, L. C. (1994) “RTSORAC: Design of a Real-Time Object -Oriented Database System”, Technical Report 94-231, Department of Computer Science, University of Rhode Island, 1994.

Prichard, J., DiPippo, L. C., Peckham, J., and Wolfe, V. F. (1994) “RTSORAC: A Real-Time Object-Oriented Database Model”, in Proceedings of the International Conference on Database and Expert Systems Applications, September, 1994.

Ramamritham, K. (1993) “Real-Time Databases”, International Journal of Distributed and Paralled Databases, 1(2), 1993.

Sha, L., Rajkumar, R., Lechoczky, J. P. (1990) “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, IEEE Transaction on Computer Systems, 39(9), 1990.

Sha, L., Rajkumar, R., Son, S., and Chang, C. (1991) “A Real-Time Locking Protocol”, IEEE Transactions on Computer Systems, Volume 40, Page 793, July 1991.

Singhal, M. (1988) “Issues and Approaches to Design of Real-Time Database Systems”, ACM SIGMOD Record, 17(1), March 1988.

Squadrito, M., DiPippo, L. C., and Wolfe, V. F. (1996) "The Affected Set Priority Ceiling Protocol For Real-time Object-Oriented Databases", Technical Report 96-250, The Department of Computer Science and Statistics, University of Rhode Island, 1996.

Stankovic, J., Zhao, W. (1988) "On Real-Time Transactions", ACM SIGMOD Record, Volume 17, Page 4-18, March 1988.

Tarjan, R. E. (1972) "Depth-First Search and Linear Graph Algorithms", Society for Industrial and Applied Mathematics Journal on Computing, Volume 1, 146-160, June, 1972.

Wells, David L., Blakeley, Jose A., and Thompson, Craig W. (1992) "The Open Object-Oriented Database: Obtaining Database Functionality by Extension", Special Issue on Object-Oriented Systems and Applications, IEEE Computer Volume. 25 Number 10. October., 1992.

Wolfe, V. F., Cingiser, L., Peckham, J. and Prichard. J. (1993) "A Model for Real-Time Object-Oriented Databases", Technical Report 93-216, Department of Computer Science, University of Rhode Island, 1993.

Wolfe, V. F., DiPippo, L. C., Prichard, J. J., Peckham, J. and Fortier, P. J. (1994) "The Design of Real-Time Extension to the Open Object-Oriented Database System", Technical Report 94-236, University of Rhode Island, 1994.