ObServer: A Storage System for Object-Oriented Applications

Mary F. Fernandez[1]
Stanley B. Zdonik[2]
Alan N. Ewald[3]

Technical Report No. CS-90-27

September 22, 1990

[1]Department of Computer Science, Princeton University, Princeton, NJ 08544
[2]Department of Computer Science, Brown University, Providence, RI 02912
[3]Digital Equipment Corp., 290 Donald Lynch Blvd., Marlboro,MA 01752

ObServer: A Storage System for Object-Oriented Applications

Mary F. Fernandez*
Stanley B. Zdonik
Alan N. Ewald†
Department of Computer Science
Brown University
Providence, RI 02912

September 22, 1990

Abstract

ObServer is a storage system designed to support object-oriented applications such as computer aided design systems, interactive programming environments and object-oriented databases. For these systems to be used effectively, the underlying storage server must support interactive transactions, provide shared access to objects and perform efficiently in a distributed workstation environment.

In this paper, we present the design and implementation of ObServer, consider its limitations and discuss areas of current and future research. We also provide performance statistics obtained from three types of example databases.

1 Introduction

ObServer is an object server designed to manage the efficient transfer of information from the secondary storage of a centralized server to the memory of an application running on a workstation. It is intended to support object-oriented applications that need to maintain a space of persistent objects and that require that this space be shared by multiple applications running on possibly many workstations.

The current version of ObServer has two major goals. The first is to support efficient graph traversal within an object-space that supports identity [KC86], and the second is to provide the primitives to support transaction synchronization in a cooperative design setting.

ObServer is designed to support a minimal type model. In this way, it is possible to construct arbitrary type systems above ObServer in higher-level layers. For example, ObServer should be usable by any object-oriented database system, regardless of the details of the data model, as a backend repository to manage the persistent store.

ObServer is meant to address computer-aided design applications. This includes electrical and mechanical CAD, software CAD, and office information systems. Some of the assumptions that we make regarding the characteristics of these applications are:

1. **Interactive applications.** The applications involve a user at a workstation designing an artifact using an interactive editor.

*Author's current address: Deptartment of Computer Science, Princeton University, Princeton, NJ 08544
†Author's current address: Digital Equipment Corp., 290 Donald Lynch Blvd., Marlboro, MA 01752

2. **Cooperating processes.** A given user will interact with the data through multiple processes, or multiple users (each with a separate process) will interact with the same data.

3. **Long interactions.** An interaction with the system will be measured in hours instead of fractions of a second.

4. **Pointer traversal is the dominant mode of access.** Many design applications currently spend a great deal of their time following pointers from one object to a related object.

The first goal restricts the way in which an application and the server can be expected to interact. Since there is a user connected to the transaction, it is not reasonable for the server to abort the transactions for any reason. This means that we cannot allow deadlocks to occur. It also means that communication with the server should be asynchronous. Object requests to ObServer do not cause the requesting process to wait. When the request is granted (or not), a message to that effect is sent back to the application.

ObServer is not an object-oriented system in that it does not contain any support for data abstraction (i.e., object classes), inheritance, or methods. It is, however, a persistent storage system for object-oriented applications since it supports object identity and can store arbitrary sized objects including classes and methods.

The principal way that ObServer supports efficient graph traversal is by a very flexible object clustering and prefetching (i.e., caching) scheme that includes replication. The support for long, cooperative transactions is addressed through a very large set of lock types including notification modes that allow users to construct non-serializable transactions.

ObServer runs on a server machine that is accessed by multiple workstations. Each of the workstations runs an independent application and requests information from the centralized server.

There are many different memory spaces that must be considered in this architecture. Data first flows from the secondary storage of the server into the main memory of the server. It next travels from these buffers across the network into the main memory of the client process that is running on the workstation. It is finally copied into the virtual memory of the application that is running on the same workstation.

When the application accesses information that is not in its local virtual memory, that information must be located in one of the other memories. The greater the distance from the applications, the greater the expense of access. The trick in making this kind of system work with reasonable performance is to make sure that data that is likely to be used in the near future is staged as close as possible to the application.

2 System Overview

In this section, we first describe the computing environment in which ObServer functions and then give a high level description of the system components.

2.1 System Components

Because ObServer is intended to support design applications, it was designed to function in a network of distributed workstations [1]. ObServer adopts a *client-server* model in which a single *database server* has centralized control over all data in its secondary store and its *clients* provide a local interface to the server for applications executing on remote workstations. We assume that client applications do not interact directly with each other and that their only source of accessing shared data is via the database server. We also assume in this implementation that the servers do not interact with each other. Although a single ObServer client application is free to access data from multiple ObServer database servers executing in the same network,

[1] ObServer II was developed on a network of Sun Microsystem workstations running SunOS^{TM} V4.0.

a server can not search for nor provide access to data controlled by other servers. In this situation, the application must keep track of where its objects came from and which server should be used to interpret an embedded reference.

There is an active process called the *binder* which maintains name and state information for each server. The purpose of the binder is to locate a server for a client, remotely initiate server execution if the server is idle, guarantee the server has recovered from any previous crashes and then complete the connection between the client and server.

2.2 Server Functionality

The server provides secondary storage for and controls access to arbitrarily sized *objects* that may be accessed by multiple, distributed clients. To the server, an object is a *unique identifier* coupled with a block of bytes. The unique identifier (UID) labels the object's data and provides clients with a handle to the object. The server provides procedures that may be invoked by a client for creating, reading, updating and deleting objects. Because ObServer was intended to support applications with differing type systems and object formats, the server has no access to the type or structure of an object. Only client applications can interpret the semantics of an object. § 3.1 discusses identifying and locating objects.

In design applications, data complexity is increased due to the high connectivity between the large number of objects that may compose a single design object. The cost of navigating a complex graph can be minimized if related objects (e.g., hierarchically linked objects) are clustered together on disk and can be retrieved as a group with the minimum number of seeks. Since ObServer does not know about connections between objects, it cannot infer which sets of objects are related. However, it does provide the application with a mechanism, called the *segment*, the unit of physical clustering. A segment can contain an arbitrary number of objects and is guaranteed to occupy a logically contiguous area on the disk. The segment is also the unit of transfer between the server and the client. The format, storage and memory mapping of segments is described in § 3.2.

When an object is created, the application specifies the segment in which the object should be stored. The server provides procedures for replicating objects in multiple segments and for migrating objects between segments. This enables a design application to locate and store objects according to varying clustering strategies. Whenever any object in a segment is referenced, the entire segment is transferred so that the client can cache them for future operations. If an object is in more than one segment, ObServer chooses one from the transaction's *segment context*, a group of segment identifiers. Segment groups are discussed in § 3.2.2.

By traditional definitions, a *transaction* is an atomic grouping of database operations. Atomicity guarantees that the transaction's modifications to the database will be done in their entirety or not at all. Transactions preserve the consistency of data and are recoverable in the event of database failure. The database scheduler interleaves transactions' operations according to some correctness criterion. Serializability is one possible criterion for determining the correctness of an interleaving. To enforce serializability in a database that uses locks, the transactions must acquire and release locks in two phases, a lock acquisition phase and a lock releasing phase. The strict ordering of the operations for locking objects and for committing transactions limits the potential for sharing objects between cooperating transactions.

Two important characteristics of design transactions are cooperation and interactivity. Cooperation implies that the transactions need to share some aspect of their work in progress. Imposing serializability restricts the permissible interleavings of transactions' operations and thus prohibits transactions from sharing uncommitted changed to objects. However, the ability to let other transactions read intermediate changes is often necessary in a design environment. We do not advocate the total absence of a concurrency control mechanism. Instead, ObServer provides read and write operations that permit transactions to share objects

in a non-serializable manner and an extended locking protocol that controls access to objects and informs cooperating transactions of the objects' usage. § 3.3 gives a more complete discussion of access control and § 3.4 discusses the nature of transactions in ObServer.

Interactivity implies that an autonomous user is controlling his interaction with the database through an on-line interface. This characteristic restricts the database from unilaterally aborting a transaction should the transaction violate a predetermined ordering of operations. ObServer's policy is never to abort a transaction to resolve possible anomalies such as deadlock or a read dependency on an aborted transaction. This policy has an obvious implication for recovery methods since interdependencies develop between transactions sharing uncommitted changes to objects. Recovery is discussed in § 3.5.

2.3 Client Functionality

To invoke a server operation, the client packages a message containing the operation's parameters and sends the message to the server via an open socket. Upon receiving the message, the server unpackages it into parameters that are passed to the server interface procedures. To simplify the application's interface to the server, the ObServer client module provides procedures for remotely invoking server operations. The client module is a library of communication and interface procedures that is bound to the executable application program at compile time. Within the application, server operations are invoked by calling identically named procedures in the client library.

Client calls do not block. The server will eventually return a message indicating the status of previous calls. The application must handle server responses by reading messages from the socket. The client does provide an additional procedure for converting a message of packed bytes into data structures accessible by the application program.

The current client does not buffer data returned from the server. This simple client module assumes that the application has its own strategy and mechanism for locally caching object and lock data. We intend to provide a more adequate client that caches objects locally for an application.

2.3.1 Initiating Server/Client Communication

To understand the interaction of the binder, server and client, we describe how communication is initiated between an application and a server. The binder executes on a single, known host in the network. The client contacts the binder providing the name of the database with which it intends to communicate. The purpose of the binder is to map a mnemonic database name into the name of the host where the server executes and the directory location of the database files. This allows a database manager to relocate a database in the network without having to inform all applications that use the database. The binder maintains the server location for the applications.

Figure 1 depicts the server, client and binder programs executing on three separate machines. In step 1, a client application contacts the binder executing on a known host and requests a connection to a server. The binder determines whether the requested server is active in step 2. If the server is active, the client is immediately given an Internet address at which the server can be reached (step 6) and the client-server link is completed in step 7. If the server is not active, the binder contacts **inetd**, the Internet services daemon [Sun87], executing on the remote host to start up a server process (steps 3 and 4). In step 5, the server and binder exchange startup information. If the server was previously shut down without any errors, the final connection between the client and server proceed in steps 6 and 7. Otherwise the server performs operations to recover from the previous crash before completing the connection.

When the client receives the Internet address of the requested server, it opens a UNIX[TM2] inet socket on which messages will be sent to and received from the server. Similarly, the server opens a socket when

[2] UNIX is a registered trademark of AT&T Bell Laboratories.

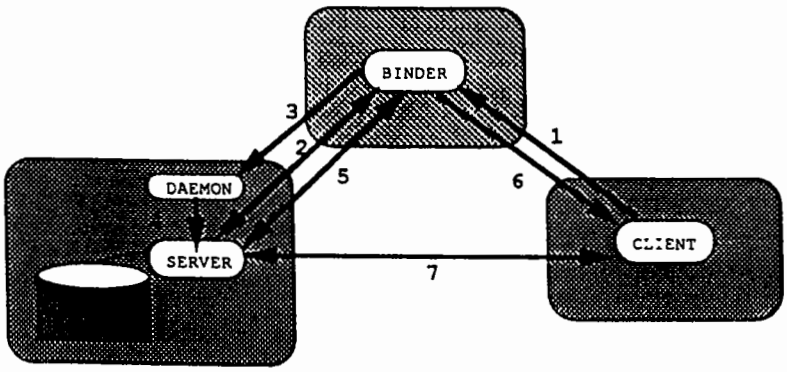

Figure 1: Interaction of ObServer programs.

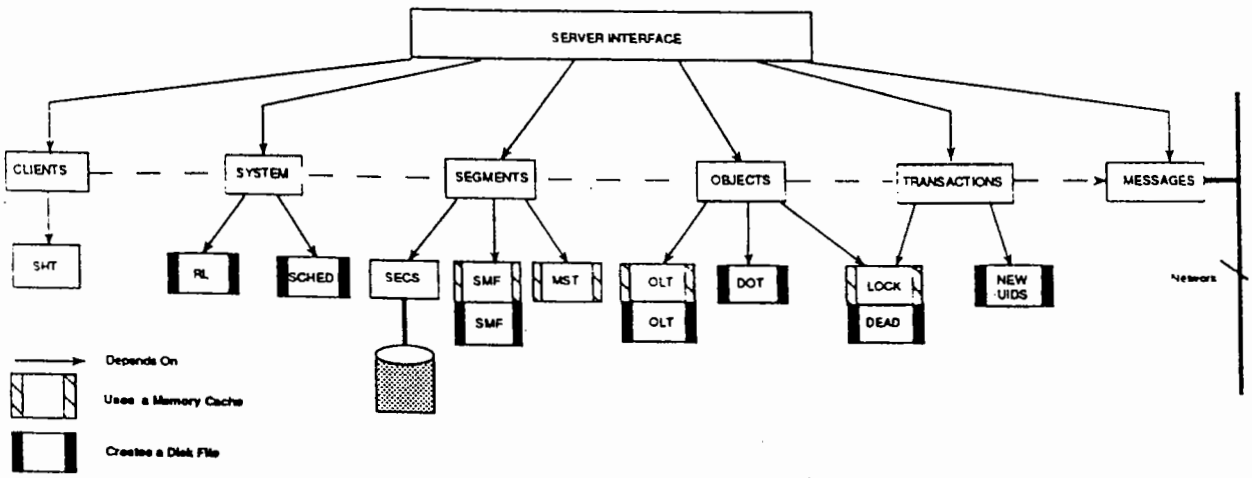

Figure 2: Server Module Interconnection Diagram.

contacted by the new client and multiplexes between open sockets servicing client requests in a round-robin fashion. The client issues requests for object access and the server responds by sending messages over the sockets.

3 Server Architecture

In this section, we concentrate on the architecture of the server. The following subsections describe in detail the structures, algorithms and policies of the modules in the server that identify, store, and control access to objects, create and destroy transactions, and perform recovery operations. Figure 2 is a module interconnection diagram of the server. There are five major modules which maintain internal information on clients, segments, objects, transactions and the server system state. The message module handles all interaction with the network including sending and receiving messages as well as buffering and packaging data for the other server modules. The names of the modules used in the diagram appear in typewriter text (e.g., `LOCK`) when used.

3.1 Objects

To an application, an object is a *unique identifier, byte-stream* pair. An application accesses an object in the server by providing an immutable external unique identifier (UID). Before creating an object, a transaction requests that the server allocate new UIDs (`NEW UIDS`) for the transaction. These UIDs are reserved for use by the transaction until it creates the object. When the transaction later writes the object for the first time,

the server maps the external UID of the new object to an internal UID. Internal and external UIDs have the same length but their structures differ. The server uses the internal UID to determine an object's location. An internal UID can be either a single location UID (SUID), a duplicated object UID (DUID), a deleted object UID or a null UID. The mapped UID indicates whether the object resides in one or more segments in secondary storage or whether the object has been deleted. A null UID indicates that the UID has not yet been assigned to a byte stream.

When an object is deleted, its internal UID becomes a *tombstone* and is never recycled. This feature is necessary for data models in which UIDs are used as embedded references to related objects. If a deleted objects' external UID were recycled, a dereference of the UID could return an object different from the original object. The tombstone UID indicates that the original object no longer exists in the database. An obvious disadvantage of this policy is that ObServer can not store many temporary objects. Although the UID space is quite large (2^{28} UIDs), creating and deleting many objects can deplete the available UIDs rapidly.

The server uses two disk files for storing the external-internal UID maps. The *Object Location Table* (OLT) maps each external UID into one of the internal UIDs described above. Each internal UID contains fields whose values are determined by the UID type. An SUID contains the segment identifier of the containing segment and the position of the object in the segment's object directory. The directory is a list of entries that include the object's UID, size and byte offset in the segment. If an object is stored in more than one segment, its external UID will be mapped to a DUID which is an index into the *Duplicate Object Table*(DOT). The DOT maps a DUID into a set of single location UIDs. To locate a duplicated object requires two mappings, first from an external to an internal UID, then from the internal UID to a set of single location UIDs.

For objects that will never migrate from their original segment nor be replicated in multiple segments, the overhead of mapping an internal UID into an external UID is unnecessary. Such an object can be assigned a fixed location UID (FUID) when created. An FUID contains the same information as an SUID. FUIDs minimize the time required to locate an object as the server never searches the OLT cache or disk file. The obvious disadvantage of FUIDs is that an FUID object can never be relocated within the database.

The OLT could potentially exceed the virtual memory of the server. For this reason, only portions of the OLT are cached in the server's memory. The OLT is segmented into pages of UIDs which are read from the OLT disk file as needed.

3.2 Segments

As previously discussed, segments are used to cluster related objects and are the unit of transfer from the server. This enables the server to pre-fetch related objects quickly and minimize the number of disk operations to read a set of objects. In this section, we describe how segments are located and stored in the server.

Client applications request that the server create new segments for the storage of objects. New segments are assigned immutable segment identifiers (SIDs) similar to the UIDs assigned to objects. Transactions use SIDs when creating new objects, when copying or moving objects between segments or when fetching an entire segment from the server. A SID, like a UID, is a location-independent identifier. Segments are not bounded in size and may grow to the maximum size of the secondary storage available in the server.

3.2.1 Accessing Objects in Segments

When an object is created by a transaction, the segment identifier of the containing segment(s) must be specified. Once the object is created, it is no longer necessary for a transaction to provide the segment identifier to access the object. The server returns objects accompanied by time-stamps indicating the last time the individual object or containing segment was accessed. The client uses these time-stamps when

making subsequent requests for objects from the server. For example, if a transaction requests object x contained in segment s, the server will return s accompanied by a time stamp t_s. If the transaction subsequently needs to access y, also stored in s, instead of requesting an object which it already has cached in its memory, it can send the time stamp t_s to the server to determine whether its cached copy of y is up to date. The server replies by affirming that the copy is the most recent version or by sending the new copy. This helps minimize the size of messages and extra disk seeks for objects that are already cached by a client. The server also maintains a table that lists the segments that have been sent to each client (SHT). This helps the server determine whether an individual object or an entire segment must be retrieved for the transaction. The application may also request that a server return an individual object without the other objects in its segment.

The server also provides procedures for copying and moving objects between segments. These routines allow an application to adopt its own strategies for clustering related objects and for modifying those strategies over time. The segment identifier(s) of the segment(s) in which an object is stored may be requested by a transaction. Replicating an object requires the segment identifier of the new segment location while moving an object requires the originating and the destination segment identifiers. Copies of objects may be deleted by specifying the segment in which the copy is stored. When the last copy of an object is deleted, it becomes a tombstone and its UID is not recycled.

3.2.2 Storage of Segments

To minimize segment access time, the secondary storage manager (SECS) stores each segment in consecutive pages on the disk. We developed SECS since the UNIX file system does not provide this service. To make this guarantee, SECS uses an entire raw disk partition to store segments. The manager is responsible for reading segments from and writing segments to the disk and for moving segments into consecutive pages when they grow beyond the bounds of their current location. An extendible hashing directory is used to guarantee access to any segment in two disk accesses [Kit87].

To find a specific number of consecutive free blocks when writing a new segment or moving an existing segment, SECS uses a free block bitmap in which free blocks are assigned zero and used blocks one. A modified version of the Knuth-Morris-Pratt string searching algorithm [Sed88] is used to find the first sequence of free blocks large enough to accommodate the segment. Over time, this first fit strategy causes disk fragmentation. To reduce fragmentation, the manager compacts the segments during periods when the server is idle or at user defined intervals. If this compaction process is interrupted by a server request, SECS completes the current move and relinquishes control to ObServer.

3.2.3 Mapping Segments into Memory

To access individual objects in a segment, the server caches the segment retrieved from disk in dynamically allocated memory. Once in memory, individual objects can be added to, deleted from or updated in the segment. The server can also return individual objects or segments to a requesting transaction. The mapped segment module (MST) manages the segments mapped into the server's memory. Because memory is allocated dynamically for the segments, it is important that memory limits are not exceeded. The amount of memory used by the MST has a direct effect on performance (see § 5). Based on the memory of the server, the user can set the maximum memory size in bytes of the mapped segment cache.

3.2.4 Segment Groups

When a server receives a request to read an object that is stored in multiple segments, it first checks if one of the segments containing the object is already mapped into memory. If one is, the server returns the mapped segment without causing any disk accesses. Otherwise, it randomly chooses one segment containing

the object. This policy does not consider the possibility that the application might prefer to pre-fetch from a containing segment not already mapped into memory. To inform the server of which segments should be pre-fetched when multiple segments are available, the application creates *segment groups*.

A segment group is a named set of segment identifiers. A transaction's *segment group context* informs the server that one group of segments is preferred over others. Whenever the server has a choice of segments to fetch for the transaction, it checks the transaction's context to determine which segment to retrieve. Applications can set the appropriate context dynamically and thus benefit by having the correct segments pre-fetched automatically. A future extension would be to return the segment that contains the object immediately and return the other segments in the group asynchronously.

3.3 Access Control

A characteristic of design environments is the need for communication between cooperating transactions. A transaction may need to lock an object, but may also like information regarding other transactions' requests to lock the object. For this reason, ObServer supports *communicative locking* of objects. An ObServer lock is a (*lock mode*,*communication mode*) pair. The lock mode specifies whether the transaction intends to read or write the object. In addition to standard read and write lock modes, ObServer provides non-restrictive lock modes that permit multiple readers/one writer of an object or multiple readers/multiple writers. The communication mode specifies whether the transaction wants notification when another transaction is queued for a lock on an object or when another transaction has updated an object. This allows a transaction to lock an object and receive information regarding other transactions' use of the object. The six types of lock modes and eight communication modes are enumerated in Tables 1 and 2.

N	Provided to allow communication on object.
NR	Non-restrictive Read. Permits other transactions non-restrictive read and write access to object.
RR	Restrictive Read. Prohibits other transactions from writing object.
MW	Multiple Write. Allows multiple transactions write access to object.
NW	Non-restrictive Write. Permits non-restrictive read access to other transactions.
RW	Restrictive Write. Provides exclusive access to an object.

Table 1: Lock Modes

N	No notification. Only lock modes are effective.
U	Inform transaction when the object has been updated.
R	Inform lock holder if another transaction cannot acquire a read lock.
W	Inform lock holder if another transaction cannot acquire a write lock.
RW	Inform lock holder if another transaction cannot acquire a read or write lock.
UR	Combination of U-notify and R-notify.
UW	Combination of U-notify and W-notify.
URW	Combination of U-notify and RW-notify.

Table 2: Communication Modes

Because of the semantics of lock and communication modes, not all forty eight possible combinations result in valid locks. Table 3 specifies valid lock mode. communication mode pairs. An example of a "cooperative

Lock Modes	Communication Modes							
	N	*U*	*R*	*W*	*RW*	*UR*	*UW*	*URW*
N	I	V	I	I	I	I	I	I
NR	V	V	I	V	I	I	V	I
RR	V	I	I	V	I	I	I	I
MW	V	V	V	V	V	V	V	V
NW	V	I	V	V	V	I	I	I
RW	V	I	V	V	V	I	I	I

Table 3: Valid Lock and Communication Mode Pairs

lock" is NR_U[3]. The lock mode gives the transaction read access to the object but permits other transactions to lock the object non-restrictively for writing. If another transaction does write the object, the transaction holding the NR_U lock is notified that the object has changed. The reading transaction may then read the new copy of the object. Another example is RW_{RW}. The transaction obtains exclusive access to the object but is notified if other transactions are queued to obtain a read or write lock. This enables a transaction to release an object lock on demand.

Requested Lock	Existing Lock					
	N	*NR*	*RR*	*MW*	*NW*	*RW*
N	G	G	G	G	G	G
NR	G	G	G	G	G	QD
RR	G	G	G	QD	QD	QD
MW	G	G	QD	G	QD	QD
NW	G	G	QD	QD	QD	QD
RW	G	QD	QD	QD	QD	QD

G : lock granted. QD: lock queued or denied.

Table 4: Compatibility of Lock Modes.

Transactions request locks on individual objects. Because of the segment pre-fetching mechanism, the transaction will receive the requested object as well as objects stored in the same segment. For this reason, a transaction can request either "hard" or "soft" locks on objects. A hard lock request indicates that the transaction will wait for the object lock (i.e., the request can be queued until the lock is available), while a soft lock indicates the transaction should not be queued (i.e., grant the lock if it is available, otherwise ignore the request). In general, a hard lock is requested for a single object and soft locks are requested for all other objects in the same segment.

When a transaction requests a lock, the server either grants, denies or queues the request. Granting a lock implies that the lock does not conflict with any existing locks. For example, a request for a non-restrictive read lock would not conflict with an existing non-restrictive write. The server queues a lock request if it conflicts with an existing lock and the transaction is willing to wait. The lock is denied if it conflicts and the transaction is not willing to wait. The server will also deny a lock request if queuing it causes a potential deadlock. Table 4 gives the compatibility between requested and existing locks. Because communication

[3]Locks are written with the communication mode as a subscript to the lock mode i.e.,LM_{CM}.

modes do not effect lock compatibility, only the lock modes are compared. The server uses a lock table stored in memory (LOCK) for querying and updating the relationships between transactions and the objects on which they hold locks.

3.3.1 Deadlock Detection and Resolution

Because hard lock requests that cannot be immediately granted are queued, there is a potential for deadlock between transactions competing for shared objects. The server searches a "waits-for" graph for cycles that indicate potential deadlock (DEAD). When deadlock is detected, the server resolves it by removing a queued request from the lock queue. It also sends a message to the waiting client informing it that the lock request was denied because it would have caused a deadlock. This resolution technique is different from traditional methods because it does not effect locks already granted to transactions nor does it abort an active transaction. This philosophy guarantees that a transaction will never be aborted to resolve deadlock.

When a deadlock cycle is detected, the most recently queued request in the cycle is removed and the transaction is notified that the previously queued request has been denied. These are the only circumstances under which a queued lock will later be denied. We attribute a certain degree of sophistication to the cooperative transactions using ObServer and thus assume they can handle the possibility that a lock request will be denied. The transactions, of course, can put themselves in a state of "livelock" by immediately re-requesting the denied lock, however we assume this will not occur. An enhancement to the current scheme would inform the transaction of which transactions and objects were involved in the potential deadlock. This information could help a transaction resolve the problem by releasing the object(s) needed by other transactions.

The default method of detection does not guarantee that a queued request will subsequently be granted. To guarantee that all queued locks are eventually granted, the user can request that the server invoke deadlock detection after each addition to the queue. This is obviously an expensive method for detecting deadlock but simplifies the interaction between the server and an application.

3.4 Transactions and Operations

In ObServer, a client application must begin a transaction to gain access to objects in the server. The transaction represents a unit of work within the application that requires controlled access to objects. It is also the entity that receives information regarding other transactions' usage of shared objects. ObServer provides operations for obtaining locks on objects, for reading and writing objects and for releasing locks on objects. In addition to the object operations, the transaction operations commit and abort are provided (see Table 5).

ObServer does not enforce serializability. That is, it is permissible for a transaction to read an object updated by another transaction before the writer commits. It is also possible for a transaction to update objects and then abort while the updates remain visible in the database. The only imposed restriction is that a transaction hold an appropriate lock at the time it requests an operation (e.g., $r_i(x)$ requires that t_i hold a read lock on object x before reading x). It should be noted that if all transactions use the restrictive-read and restrictive-write locks in a two-phase manner, the result will be a serializable schedule. § 7 discusses current work on a concurrency control mechanism for controlling the interleaving of cooperating transactions' operations while protecting non-cooperating transactions from the possibly non-serializable interactions of others.

Because ObServer permits transactions to share work in progress, the failure of a transaction to complete successfully does not guarantee that changes it made to the database will be undone. This implies that ObServer's transactions are not necessarily atomic. Currently, any changes to objects that are received by

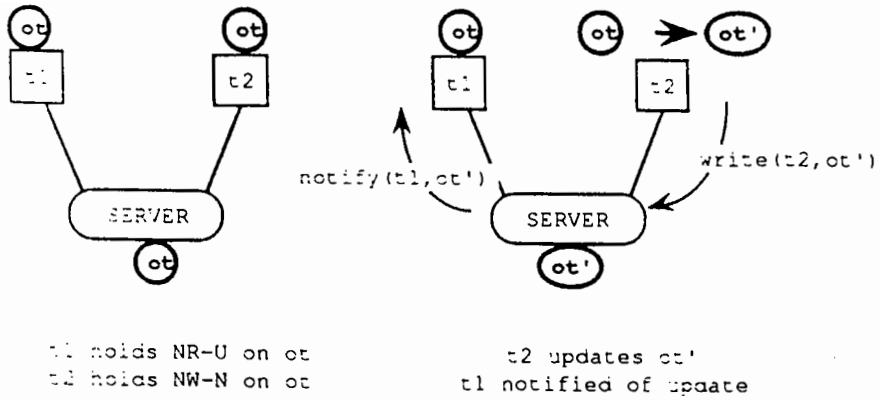

Figure 3: Use of update notification

the server become permanent. This policy assumes that all transactions that share objects non-restrictively are privy to the changes made by all other transactions.

$r_i(x)$	Transaction t_i reads object x.
$w_i(x)$	Transaction t_i writes object x.
$rl_i(l, x)$	Transaction t_i requests a read lock l on object x.
$wl_i(l, x)$	Transaction t_i requests a write lock l on object x.
$ru_i(x)$	Transaction t_i releases its read lock on object x.
$wu_i(x)$	Transaction t_i releases its write lock on object x.
c_i	Transaction t_i commits. All locks held by t_i are released.
a_i	Transaction t_i aborts. All locks held by t_i are released.

Table 5: Operations on Objects and Transactions

The following example taken from [FZ89] illustrates the use of ObServer locks and object operations between two cooperating transactions (see Figure 3). Consider two transactions t_1 and t_2 which correspond to two design applications. The applications are used by two people collaborating on the design of an airplane jet engine. t_1 is updating the fuselage *of* and t_2 is modifying the turbine *ot*. $t1$ is also reading *ot* and wants to be notified if the object is updated. $t1$ requests a NR_U lock on *ot*. When t_2 updates *ot* to *ot'*. t_1 is notified of the update. t_1 may then reread *ot* from the server.

3.5 Recovery

The recovery method chosen for a database must handle failures caused by external events, such as site crashes and network partitions, as well as internal events, such as transaction aborts and database failures. Regardless of the cause, the chosen recovery method must preserve the permanence and consistency of the data. It is recognized that recovery interacts subtlely with the chosen concurrency control method [Wei89]. For instance, in a database that enforces serializability via two phase locking (2PL), cascading aborts can be avoided if the system uses *strict 2PL* [BHG87]. However, in an unconventional system like ObServer, where non-serializable histories are permitted, system-initiated aborts of transactions are not acceptable.

Just as when a transaction aborts unexpectedly, when a database or system failure occurs the system must decide which transactions to redo or undo. In a system that performs *selective redo* during recovery, transactions that were uncommitted at the time of the crash are undone and committed transactions are redone. In the current ObServer implementation, we ascribe to a *redo-only* crash recovery scheme. That is, we "repeat history" since it is possible that committed transactions read from uncommitted transactions

and are thus dependent upon their changes. This is similar to the scheme used in ARIES [MHL+89]. For example, if transaction t_1 shares its work with other transactions by updating an object and then the system crashes before t_1 commits but after transactions that read from t_1 commit, potential anomalies may exist if t_1 is not redone. The server, however, cannot unilaterally undo all transactions that read from t_1 since cascading undos can ensue. The dependencies that existed between t_1 and transactions that read from it should be maintained so that recovery mechanisms can choose a combined strategy of redoing and undoing transactions. § 7 discusses current work on a recovery mechanism which accounts for the complex dependencies that exist between non-serializable transactions.

Each operation requested by a transaction is recorded in a recovery log file (RL). In the case that the server crashes, the recovery log is used to reexecute each transaction request. During recovery, the operations are executed to restore the server to its state immediately preceding the crash. Because no transactions are actively connected to the server during recovery, no messages are sent to clients during recovery. Once the server has completed recovery procedures, client transactions may reconnect to the server and continue executing. This feature allows interactive transactions temporarily suspended by a server failure or network partition to resume interaction without losing any work.

4 Example Applications

4.1 GARDEN

GARDEN is a multiparadigm programming environment that is designed to support conceptual programming [RM89]. It allows users to define their own graphical representations for programs, and provides a framework for combining program pieces that have been specified using these different visual metaphors.

GARDEN achieves this multiparadigm integration by treating everything as an object that supports a method called *execute*. In this way, entities like statements, expressions, and values are all objects in the GARDEN environment. These objects must all be stored as separate objects in a persistent storage component. GARDEN now uses ObServer as its backend.

GARDEN uses ObServer's non-restrictive read, update notify locks (NR_U) as read locks and it's restrictive write, read-write notify locks (RW_{RW}) as write locks. In this way, it need only reread an object when another process has modified it, and it need only communicate a changed value back to the server when another process requires it.

It also reads an entire working set of objects into its main memory for the duration of a session. By pre-specifying the segmentation policy well, this pre-fetching is accomplished as efficiently as possible.

GARDEN uses the asynchronous client interface successfully by scheduling light weight processes to invoke server operations. A process invokes a server operation then suspends itself on a wait queue. A reader process listens for messages on the open socket, decodes each message to determine the process for which the message was intended, then reschedules the process to continue execution. This enables GARDEN to catch messages related to other clients' activities while waiting for object or segment data to be returned from the server.

ObServer provides a *database handle* for storing distinguished UIDs. GARDEN uses the handle when a database is opened to access the root objects that reference all other persistent objects in the environment. Only objects reachable through these objects are available for GARDEN's use.

4.2 ENCORE

ENCORE [ESZ89] is an object-oriented database system. It allows users to define new types and to create instances of these types.

Objects are the denotable values in an ENCORE database. As such, they possess a unique identity which can be used to form references. All objects are instances of a common type called type Object.

A type is an abstract data type, and consists of an interface and implementation. Queries are concerned with the type interface, although their optimization may be concerned with the type implementation. An abstract type definition includes the *Name* of the type, a set of *Supertypes* for the type, a set of *Properties* defined for instances of the type and a set of *Operations* which can be applied to instances of the type. In addition to user-defined abstract data types, we assume a collection of atomic types (Integer, String, etc.).

Types are created as instances of the type Type. The ENCORE data model supports a subtype relationship between types based on the principle of substitutability. That is, any instance of the subtype should be usable in any context that is expecting an instance of the supertype. The subtype mechanism was also designed to support strong static typing if this type system (i.e., ENCORE) were embedded in a compiled programming language. An operation named f defined on a type T can be overloaded by redefining it on a subtype S. The signatures of f defined on S and f defined on T must obey countervariance rules. Properties reflect the state of an object while operations may perform arbitrary actions. A property is a typed object whose *get_value* operation may be implemented as a stored value or as a function. Operations are also objects in this model. The operations that are defined on a type are actually types themselves. These operations form a subtype lattice based on the types of their arguments and on any additional operations that are defined on these type objects. Instances of these types correspond to activations.

ENCORE uses several simple heuristics for clustering objects in ObServer. For some types it clusters all its instances in a segment, while for others, it clusters together all objects that are n references away from a given object. It can also cluster sets of objects based on common values of a property (e.g., cluster all blue cars together).

4.3 Other Applications

Any application that needs access to a large space of persistent objects that are sharable across a local area network would be a candidate application for ObServer. For example, a blackboard system or a mail system could be built on top of the basic ObServer platform.

Another class of application that could benefit from a storage system like ObServer is hypermedia systems. In these applications, the dominant mode of access is pointer traversal from one hypermedia document (i.e., object) to another. We have recently begun a project to connect ObServer with the Intermedia system [Mey86] developed at Brown University. Here one of the major design problems is to determine what a reasonable clustering strategy is for a hypermedia document. This will involve determining which links constitute the main path(s) through the hypermedia web.

5 Performance

In order to measure the performance of ObServer, a set of benchmarks were devised to test ObServer in various application settings. The benchmarks were designed to simulate a transaction processing (e.g., credit/debit transactions on bank accounts), a computer aided design (CAD), and programming environment applications. The first two benchmarks were carried out by creating databases for the application being simulated, then running programs to simulate accesses and updates of the stored objects. Objects were stored in the databases according to the semantics of the application. The third benchmark involved the creation of a database to store the object hierarchy necessary to support the GARDEN programming environment.

Each benchmark did very little computation other than operating on the database. Compute time between database operations was simulated by inserting wait loops executed a random number of times within an upper bound. Each benchmark was only run once due to time constraints. This may result in

some anomalies in the reported statistics due to spurts of network activity or disk accesses. The system used for benchmarking was lightly loaded during the testing process. Both the ObServer client and server processes were run on the same machine. This implementation may affect benchmark results in that the database server process had to compete for memory space and I/O bandwidth. Other implementations might execute the server process on a dedicated machine.

The version of ObServer that was benchmarked included no compiler optimization and very little manual optimization. A large percentage of code is used strictly for debugging and tracing. It is likely that concerted effort at optimization would significantly improve the performance of the system.

5.1 System environment

All of the benchmarks were carried out on a Sun SPARCstation 1 ^{TM} [4] running SunOS 4.0. The machine configuration included 8 MB of physical memory and a 105 MB local disk. The local disk was used for operating system files and also contained 32 MB of swap space. Block size and page size for the machine were both 1024 bytes. Secondary storage for user and database files was provided via NFS ^{TM} [5], a distributed file service. The NFS servers were Sun 4/280s running SunOS 4.0 configured with 32 MB of memory and 1-3 gigabytes of disk space. Overhead resulting from NFS is included in all of the gathered statistics. Benchmarks using local disks for all secondary storage would undoubtedly produce different results.

5.2 Parameters

Each benchmark was run multiple times altering one of several parameters for each run. Since all objects are referred to and operated on by ObServer in terms of their containing segments, the major ObServer system parameter altered was the size (in bytes of virtual memory) of the server's segment cache (`MST`). By altering this parameter, it was possible to force the server to page segments with varying frequency. Optimal segment cache size depends on database access patterns and how objects in the database are stored in segments.

Another parameter modified during testing was the number of objects stored in a segment. This parameter affected the size of segments as well as the likelihood that object references would fall within the same segment. This parameter does not affect the overall size of a database, but rather affects how the individual objects are grouped into segments. Choosing an optimal partitioning of objects into segments depends on access patterns and locality of object references. It is analogous to selecting an optimal block size for a file system.

The CAD application benchmarks used cached objects in the client process to show the affect of moving objects closer to the application in the hopes that subsequent accesses would reference objects available locally.

The final parameter adjusted during testing was the overall job mix for accessing a database. Jobs were designed to access objects randomly, locally in a cluster, or both. Accessing objects randomly defeated any object clustering that was implemented during database creation forcing the server and/or client to page segments and objects to/from their caches. Access to objects in a pre-defined cluster optimized usage of the server's segment cache. Mixing random and clustered access provided another data point.

5.3 Explanation of Benchmarks and Results

The analyses of the benchmark results include tables and graphs that contain performance statistics for different values of the parameters previously described. Each table lists a database operation followed by its execution time, minor and major page faults, and disk reads and writes. The times for each operation

[4] SPARCstation is a trademark of Sun Microsystems

[5] NFS is a trademark of Sun Microsystems

are in seconds and measure the average real time between a client's initiation of a database operation and receipt of the return message needed to continue processing. Minor page faults are the average number of page faults incurred which did not require disk I/O. Major page faults are the average number of page faults which did require disk I/O. Disk reads are the average number of blocks read while executing the operation. Similarly, disk writes are the average number of blocks written.

5.3.1 Creating a Transaction Processing Database

The transaction processing benchmark was designed to simulate a banking application. Each database contained 10,000 bank account objects which stored an account balance. Account identifiers were implemented as UIDs. Databases were created in single transactions with either 500 or 1000 accounts per segment. This lead to segment sizes of approximately 10K and 20K bytes respectively. Account objects were stored in segments by increasing identifier. Each database also contained a segment which stored a directory of identifiers. At the end of a creation process, a checkpoint operation was performed on the database and the server was shut down. Each banking database created was approximately 0.8 MB in size. The statistics for the creation of several banking databases are shown in Tables 6 and 7.

Server Operations	Real Time (secs)	Min. Page Faults	Max. Page Faults	Block Reads	Block Writes
begin	2.497	0.00	2.00	0.0	0.0
commit	6.400	1.00	0.00	0.0	0.0
checkpoint	6.142	26.00	38.00	22.000	53.000
create segment	0.115	1.524	0.048	0.0	0.0
write object	0.0179	0.0194	0.0019	0.0009	0.0160
open database	0.440	25.000	41.000	14.000	12.000
close database	0.700	16.000	10.000	1.000	22.000

Table 6: Transaction Processing Database Creation. 500 objects per segment. 1 MB segment cache.

There was not a significant difference between creation statistics using different server segment cache sizes due to the fact that only one segment was accessed at a time and was written only once obviating the need

Server Operations	Real Time (secs)	Min. Page Faults	Max. Page Faults	Block Reads	Block Writes
begin	3.269	0.000	2.000	0.000	0.000
commit	7.600	3.000	1.000	0.000	0.000
checkpoint	13.331	21.000	54.000	14.000	57.000
create segment	0.171	4.091	0.727	0.364	0.000
write object	0.0657	0.259	0.254	0.00127	0.0139
open database	0.510	26.000	40.000	13.000	12.000
close database	0.700	47.000	30.000	8.000	22.000

Table 7: Transaction Processing Database Creation. 1000 objects per segment. 1 MB segment cache.

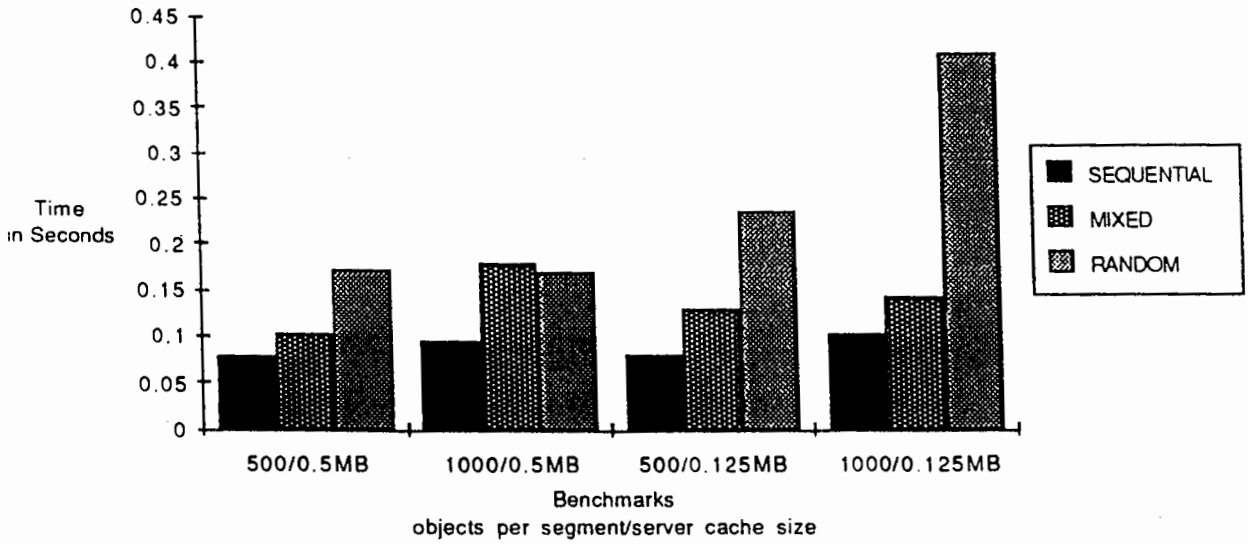

Figure 4: Transaction Processing Database: Object Access Times.

for a segment cache large enough to hold several segments. Objects were written in groups of 500 or 1000. The reported time to write an object is per object. A segment must be created to contain objects before they are written to the database. If writing objects into a segment causes an overflow of the segment, or the object directory in the segment, then the server must expand the segment which may involve relocating objects in the segment. The time to write a 1000 object segment is more than twice the time to write a 500 object segment because a segment expansion occurred in the 1000 account objects per segment case.

5.3.2 Accessing a Transaction Processing Database

After the database was created, it was reopened and operations were performed to test object access time. Each benchmark consisted of multiple, independent processes that simultaneously executed transactions to read from and write to the database. The processes were of two types. One type executed 1000 debit/credit transactions against account objects in the database. Each transaction randomly selected an account identifier, requested a restrictive write lock on the selected account, credited or debited the account, and committed the transaction to the database. Each transaction was followed by a randomly chosen wait period to simulate compute time. Objects were not cached locally in the client because of the type of the transactions being performed, random updates to individual objects. Objects were requested individually leaving all caching to the server process. The other process type performed a single read-only transaction which accessed all 10,000 account objects in the database sequentially by account identifier. A non-restrictive read lock was acquired for each account object and was released immediately after it was read. Each object access was followed by a randomly chosen wait period.

The benchmarks performed included three jobs mixes of the two types of processes. These were

1. three sequential access, read-only processes.

2. one sequential access, read-only process and five random update processes, and

3. five random update processes.

The results of these benchmarks are shown in the graph in Figure 4.

The statistics show the advantage of performing consecutive accesses to objects clustered within the same segment. The sequential access, read-only benchmarks read objects in the least amount of time, followed by the mixed processes and finally the random access processes. This reflects the extent to which the server was able to optimize use of the segment cache. There are noticeable differences in access timed between benchmarks where the number of objects per segment differed. Although the objects were small, the objects clustered in larger segments took longer to access. In some of the random access benchmarks, larger segment sizes translated into longer object access times due to the server having to move larger blocks of data to and from the disk. Larger segment cache sizes improved most operation times except for database close operations which generally took longer because of the increased overhead of emptying a large cache.

5.3.3 Creating a CAD Database

The goal of the CAD benchmark was to simulate the storage and access patterns that are characteristic of a CAD application. When creating the database, some objects were clustered together to form composite objects The sizes of objects were varied to represent different types of design objects. Limited disk space constrained the creation of very large databases or very large objects.

Each database contained 3000 design objects implemented as character strings of various sizes. Half of the objects were clustered into groups of 50 to simulate composite objects. Each of the objects clustered in this fashion ranged in size from 0 to 4096 bytes. The average size of a segment containing a composite object was approximately 65K bytes. The remaining 1500 objects were placed sequentially in segments of 100 or 250 objects. The size of these individual objects ranged in size from 0 to 1024 bytes. The average size for a segment containing 100 individual objects was approximately 53K bytes. The average size for a segment containing 250 individual objects was approximately 132K bytes. One segment stored a directory of identifiers used for design objects. At the end of a creation process, a checkpoint operation was performed and the server was shut down. The design databases created by this process were approximately 4.5 MB in size. The statistics for the creation of several design databases are shown in Tables 8 and 9.

Server Operations	Real Time (secs)	Min. Page Faults	Max. Page Faults	Block Reads	Block Writes
begin	10.693	0.000	2.000	0.000	0.000
commit	2.196	1.000	8.000	0.000	0.000
checkpoint	27.536	68.000	274.000	20.000	213.000
create segment	0.184	1.348	0.109	0.043	0.935
write object	0.02757	0.048	.1150	0.005	0.076
open database	0.650	58.000	68.000	41.000	33.000
close database	0.340	3.000	42.000	4.000	14.000e

Table 8: Design Database Creation. 100 objects per segment. 1 MB segment cache.

The tables show that the time to write 100 or 250 objects into segments was roughly linear in the number of objects. No segment expansions occurred during these write operations. There were fewer objects per segment than in the banking database. However, the objects were much larger on average and the time spent in write operations reflect the increased overhead of large objects and therefore larger segments. For example, it took 0.028 seconds to write an object in the CAD database using a 1 MB segment cache whereas it took only 0.018 seconds to write an account object in the banking database using a cache of equal size. The increased overhead of large objects is also apparent in the statistics for checkpointing the databases.

Server Operations	Real Time (secs)	Min. Page Faults	Max. Page Faults	Block Reads	Block Writes
begin	4.787	0.000	2.000	0.000	0.000
commit	2.591	2.000	2.000	0.000	0.000
checkpoint	44.730	70.000	306.000	16.000	203.000
create segment	0.205	1.730	0.135	0.108	0.919
write object	0.020	0.023	0.058	0.002	0.038
open database	0.510	53.000	51.000	25.000	23.000
close database	0.210	6.000	22.000	1.000	15.000

Table 9: Design Database Creation. 250 objects per segment. 1 MB segment cache.

5.3.4 Accessing a CAD Database

To test object access time, the CAD database was reopened and multiple, independent processes were executed simultaneously to read and write objects. All processes simulated CAD applications by performing relatively long transactions and requesting shared access to design objects. Half of the processes also simulated transitive closure references within composite objects to test the advantages of having logically related objects in the same segment. Each process executed one transaction that performed 1000 read operations. Subsequent write operations were performed 33 percent of the time. In the processes simulating pointer references, if a design object that was part of a composite object was referenced, the next reference was to an object in the same composite object 50 percent of the time.

Another parameter compared the method of caching of objects in the client. In half of the processes, design objects were requested individually and were cached in the client's memory. In this case, segment caching was performed only in the server process. In the other half of the processes, design objects were requested along with their containing segments. In this case, segment caching was performed in both the server and the client. In both cases, subsequent operations to the same object referred to the client cache whenever possible. If an object was in the client cache, the server was queried if the client had the latest copy of the object and whether or not a non-restrictive read lock was held on the object by the transaction. If the object copy was invalid, it was individually reread and placed in the client's cache. When a transaction wrote an object, it first acquired a multiple write lock, then released the lock after the operation completed. Each object access was followed by a randomly chosen wait period implemented as a tight loop executed between 0 and 1,000,000 times.

The benchmarks included two job mixes of the described processes. They were

1. three processes that simulated transitive closure references to objects within the same composite object and other random access references, and

2. three random access processes.

The statistics for several runs of these benchmarks are shown in the graph in Figure 5.

As in the case of creating the CAD databases, access operations generally took longer than in the transaction processing benchmark due to larger object sizes. As in previous cases, larger cache sizes generally produced better results. Processes performing some clustered accesses generally had better read times than processes performing completely random accesses due to better usage of the server's segment cache. The most interesting result was the effect that client caching had on object read times. When objects were cached

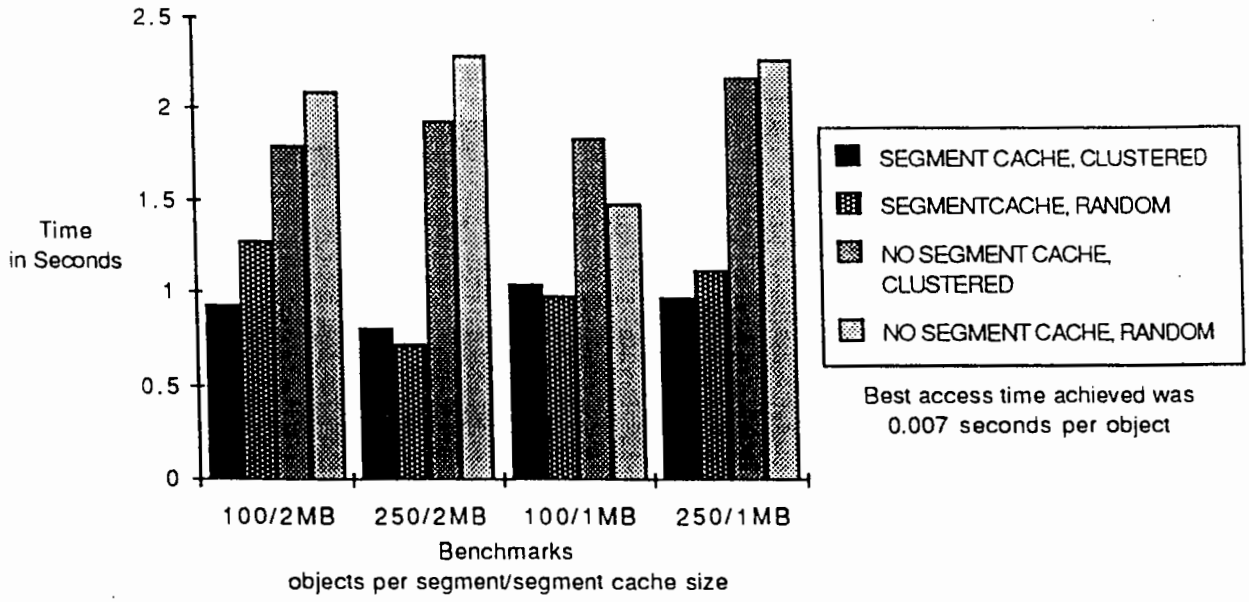

Figure 5: CAD Database: Object Access Times.

individually in the client (i.e., other objects in the containing segment were not pre-fetched), object read times were fairly uniform. In this case, objects clustered 100 per segment took uniformly less time to access than those clustered 250 per segment. When all of the objects in a segment were pre-fetched and cached by the client, read times overall were greatly reduced. The initial read operation that returned the entire segment was expensive, but subsequent individual read requests usually resulted in the server's confirming that a valid copy of the object was cached locally. It is important to note that the times for accessing objects when client segment caching was used are amortized over the time to fetch a segment and subsequent calls to verify that an object copy was valid. Each verification request generated a message sent to the server. These times would be further reduced if verification requests were bundled into a single message.

When objects were prefetched, there was not a noticeable difference in read times for processes doing clustered versus random object accesses because once a number of segments were read and cached in the client, the difference between making multiple successive references to the same segment and multiple references to different segments was negligible because most references were to the client's virtual memory. It appears that optimal use of this type of database would include a relatively small number of objects per segment, a large segment cache in the server and a caching strategy that included pre-fetching an object's segment in the client.

5.3.5 CAD Database Best Case Scenario

We also considered a best case scenario in which segments were pre-fetched by the server and the client obtained restrictive read locks on all objects. In this benchmark, objects were stored 100 per segment and the server used a 1 MB segment cache. Because the client acquired restrictive read locks, it was guaranteed to always have a valid copy and thus never had to send verification requests to the server. The benchmark made 10,000 object references of the 3,000 objects stored in the database. We calculated a best case object access time using the formula

$$\text{best access time} = (\text{average segment access time} * \text{number of segment fetches} / 10,000)$$

The time to access an object in the cache was not included since main memory references are negligible. This resulted in an average best case access time of 0.007 seconds.

5.3.6 GARDEN benchmark

The final benchmark created and stored the type hierarchy used by the GARDEN programming environment. This is an interesting benchmark because objects contain real data that reference other objects in the system. The resulting databases stored approximately 2600 objects with an average size of 38 bytes. These objects were organized into three large segments. The creation process performed most of its work in virtual memory using lightweight processes. ObServer was used primarily to reserve object identifiers and to create segments during this phase of the program execution. At the end of the creation process, the entire hierarchy of objects was stored in ObServer by sequentially traversing the objects in the processes' memory space. Notification was used to ensure that the same object was not written to the database multiple times. Objects were written to their respective segments individually rather than as a group. The results of this benchmark are shown in Table 10.

Server Operations	Real Time (secs)	Min. Page Faults	Max. Page Faults	Block Reads	Block Writes
begin	0.000	4.000	0.000	0.000	0.000
commit	1.340	0.000	1.000	0.000	0.000
checkpoint	0.443	14.667	10.000	2.333	30.667
create segment	0.013	3.667	0.000	0.000	1.667
write object	0.052	0.756	0.005	0.001	0.710
open database	0.240	37.000	13.000	3.000	6.000
close database	0.320	15.000	2.000	0.000	16.000
alter lock	0.003	0.001	0.000	0.000	0.000
notify of update	0.001	0.000	0.000	0.000	0.000
add handle	0.040	0.000	1.000	0.000	1.000

Table 10: GARDEN Database Creation. 875 objects per segment. 1 MB segment cache.

6 Other Work

Many object-oriented databases are implemented as some variant of an architecture that has a high-level object-interpreter (i.e., the object-oriented type system) built on top of a low-level object manager. An important way to distinguish storage systems is by the level of semantics that they support. Some systems like ObServer put little semantics in the object manager while others incorporate a more complex type model into the storage system. A simple storage model like ObServer allows the system to be used in more contexts since it does not put many restrictions on the applications that use it, while at the same time having a lower-level model can limit the kinds of functions that can be performed in the server.

O2 [LRV88] is an object-oriented database being developed by Altair, a French consortium. The O2 object manager [VBD89] is responsible for managing transactions, distribution, complex objects, and access to the details of secondary storage. Like ObServer, it is designed to run in a client/server architecture. Unlike ObServer, it implements a number of primitives from the O2 data model, including tuples, lists, sets, text and bitmaps.

The O2 object manager allows objects to be clustered near each other on the disk. It uses heuristics such as storing all components of a complex object together to determine clustering. The unit of pre-fetching is

the page. If an object x could not be placed on the same page as a related object y, x is not pre-fetched into the workstation when y is accessed. ObServer supports segments, a larger unit of clustering and pre-fetching. A segment is of arbitrary size and is guaranteed to fall on logically contiguous portions of the disk. When any object is referenced the entire segment is pre-fetched regardless of its size. This gives control to the user over what the unit of pre-fetch should be.

The O2 object manager has support for transactions. It does not, however, have any support for non-serial behavior. It does allow updates to happen in the workstation before write locks have actually been granted by the server; however, at commit time, it must check to make sure that the transaction has been granted write locks on all pages that have been updated. If this is not the case, the transaction is aborted. ObServer requires that appropriate locks be granted before operations can proceed, however it supports a number of "weaker" lock types that allow non-standard sharing patterns to occur. The availability of notification modes on locks allows this process to be managed by the application.

Mneme [Mos89, MS88] views objects as having two parts, a pointer part that stores a fixed number of embedded object identifiers and a heap part that contains an uninterpreted sequence of bytes. This is a somewhat higher-level view of objects than ObServer and provides enough additional information so that the the system could do simple tasks like garbage collection.

Mneme uses object identifiers that code location information. In this way, the fetch of an object does not require a lookup in a hash table to find its location. It means, however, that objects are not free to migrate. ObServer uses a mixed scheme in which object id's can be either location independent or location dependent.

Mneme provides a clustering and pre-fetching scheme that is very similar to ObServer's in that they are of arbitrary size and are transferred as a unit when any object in the segment is referenced. Mneme does not allow objects to be replicated across segments as in ObServer.

Other important work on storage systems for objects include the POSTGRES storage system [Sto87] and the Exodus Object Manager [CDG+88].

7 Plans for the Future

ObServer supports cooperative transactions by using more flexible locking schemes to increase concurrency and facilitate the sharing of work-in-progress. However, we have identified several problems with the current ObServer mechanisms, including the following:

- non-cooperating transactions cannot be prohibited from interacting with a set of cooperating transactions in an unauthorized manner;
- the interleavings of cooperating transactions' operations are not controlled; and
- the recovery model does not recognize the dependencies that exist among the operations done by cooperating transactions.

Allowing cooperation among a privileged set of transactions should not subject all transactions to the possibly non-serializable behavior of the group, nor make them privy to the group's uncommitted changes. For those transactions which are cooperating, the system should support user-definable orderings of the transactions' operations and a reliable recovery mechanism that recognizes dependencies between transactions.

We propose a hierarchical model for specifying the logical grouping of cooperating transactions in an object server. A *transaction group* contains a set of members which cooperate to do a single task. It actively controls the interaction of its cooperating members. A *transaction group member* may be either an individual transaction or another transaction group. No member may have more than one parent. Thus, transactions

and transaction groups may be nested in any tree-like manner to form a *transaction hierarchy*. This structure is more flexible than the three-level hierarchies previously proposed in [K+84], [K+85], and [R+88]. It allows similar types of transaction groupings and interactions as those allowed by the abstract model of [Lyn83].

The behavior of a transaction group is characterized by its user-defined *internal protocols* and *external protocols*. A transaction group's internal protocols specify the allowable interactions among its members, including

- either an enumeration of its members or an authentication procedure for authenticating new members;
- a synchronization mechanism to control the interleaving of member operations;
- the rules for mapping internal operations (i.e., operations by the members of the transaction group on the objects in the transaction group's cache) into external operations (i.e., operations by the transaction group on objects in its parent's cache);
- the rules for resolving deadlocks among members, such as whether or not the transaction group should force some a to abort, and if so, how to choose which one; and
- the rules for handling recovery due to member failure.

Its external protocols specify how the transaction group may interact with its parent and siblings in the transaction hierarchy. At any level in the tree, the external protocols of a transaction group member must match with the internal protocols of its parent.

Associated with each transaction group is a *cache* containing objects which currently are being accessed by its members. All operations by the members of a transaction group on any objects are done locally, on the copy of the object in the transaction group's cache. At the root of the transaction group hierarchy, the cache is the database itself. An object is read automatically into a transaction group's cache when one of its members initiates a successful read or write lock request operation on that object. Here, an operation succeeds when it is allowed by the parent transaction group's internal synchronization protocols. An object is written back to the parent transaction group's cache when the member indicates that it has finished modifying the object. Thus a transaction group appears to its members to be the database server responsible for storing objects, controlling member access to objects, notifying members of object use and handling recovery.

Because the members of a transaction group cooperate, they are no longer self-contained. This means that they may not individually produce correct operation histories. That is, the sequence of operations by a single member may not leave the database in a correct state. Thus, the transaction group must be responsible for ensuring that the combined history produced by all of its members is correct.

One of the properties specified by a transaction group's internal protocols is the notion of correctness it enforces. Each transaction group has a notion of the allowable operation sequences by its members on the objects, and thus the allowable interactions among its members. These are called *patterns*. Similarly, there is a notion of forbidden operation sequences, called *conflicts*. This idea was first proposed by Skarra [Ska89]. The concurrency control mechanism for a transaction group understands these sequences, and examines each operation requested by its members to ensure that it will extend the transaction group's history in a way that is allowed by its pattern and conflict specifications. If it is allowed. then the operation is executed. Otherwise, it is refused.

A transaction group is also the basic recovery unit in the database. That is, when a member of the transaction group fails or aborts, the transaction group coordinates recovery among the other members. The recovery process must ensure that the transaction group's cache is left in a correct state, i.e., one that could be reached by some sequence of operations allowable by the transaction group's internal synchronization

protocols. The transaction group's internal protocols also may be used to tailor the recovery process to the specific needs of the group.

Recovery is always made more difficult by the existence of dependencies among the transactions. Normally, these are limited to *reads-from* dependencies, which exist because some transaction read an object that was written by some other uncommitted transaction. However, once the concurrency control mechanism starts enforcing specific patterns, additional dependencies are associated with those patterns. In particular, a transaction's ability to do some operation may be dependent on some other transaction's having previously done some other operation. If the member which did the earlier operation aborts, then the later operation is no longer valid. We call this type of dependency an *operation dependency*. Both reads-from and operation dependencies must be known by the database for any recovery mechanism to work correctly.

When a transaction group ($\mathcal{TG}_{\mathcal{M}}$) is itself a member of another transaction group ($\mathcal{TG}_{\mathcal{P}}$), it must translate the combined history of its members, which conforms to its internal protocols, into an equivalent history which is correct according to its external protocols. By an equivalent history, we mean either an identical history, or one where the operations by the members of $\mathcal{TG}_{\mathcal{M}}$ are collapsed into a shorter sequence of operations by $\mathcal{TG}_{\mathcal{M}}$ to $\mathcal{TG}_{\mathcal{P}}$'s cache, as defined by the supplied mapping from $\mathcal{TG}_{\mathcal{M}}$'s internal protocols to its external protocols. For example, say $\mathcal{TG}_{\mathcal{M}}$ reads an object from $\mathcal{TG}_{\mathcal{P}}$'s cache, then allows its members to execute several read and write operations on the copy of the object in its cache, then writes the results back into $\mathcal{TG}_{\mathcal{P}}$'s cache. The equivalent external operation history from $\mathcal{TG}_{\mathcal{M}}$'s viewpoint contains only the read operation from $\mathcal{TG}_{\mathcal{P}}$'s cache, followed by the final write operation. None of the internal operations are preserved. This translation allows each level in the transaction hierarchy to adhere to its own view of correctness. It also hides the internal operations of the transaction group members from its parent, because it removes all of the operations done on its own copy of the object from the overall history, and only preserves the external operations. This allows, among other things, for encapsulated groups of non-serializable transactions to be members of other serializable groups.

Because we can not determine the clustering of transactions a priori, the transaction hierarchy should be dynamically reconfigurable. The root transaction group, which maintains the database directly, always exists. Other transaction groups and transactions may join or leave existing transaction groups as a design effort progresses. However, an authentication procedure should exist for joining a transaction group, to prevent unauthorized access to the transaction group's object versions.

The transaction hierarchy as we have defined it provides a more structured environment for cooperative transactions than the more flexible locking scheme of ObServer. It provides a mechanism for tailoring the interactions among the different transactions using the object server, both by enforcing certain operation sequences and by forbidding others. However, allowing structured cooperation also means increasing the complexity of concurrency control and recovery in the object server. Our current work includes defining algorithms for specifying and enforcing patterns and conflicts, maintaining dependencies, and recovering cooperatively.

8 Acknowledgements

Marian Nodine is actively researching transaction groups and contributed to the section on future work.

References

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[CDG+88] M.J. Carey, D.J. DeWitt, G. Graefe. D.M. Haight, J.E. Richardson, D.T. Schuh, E.J. Shekita, and S.L. Vandenberg. The Exodus extensible dbms project: An overview. Technical Report 808, University of Wisconsin, Madison, November 1988.

[ESZ89] P. Elmore, G. Shaw, and S. Zdonik. The ENCORE object-oriented data model. Technical report, Brown University, November 1989.

[FZ89] Mary Fernandez and Stanley Zdonik. Transaction groups: A model for controlling cooperative transactions. In *Third International Workshop On Persistent Object Systems*, January 1989.

[K+84] W. Kim et al. A transaction mechanism for engineering design databases. In *International Conference on VLDBs, Singapore*, 1984.

[K+85] P. Klahold et al. A transaction model supporting complex applications in integrated information systems. *ACM SIGMOD*, 1985.

[KC86] S. Koshafian and G.P. Copland. Object identity. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*. ACM SIGPLAN, September 1986.

[Kit87] Luther M. Kitahata. A file system for an object-oriented database. Master's thesis. Brown University, 1987. Technical Report CS-87-M15.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O2, an object-oriented data model. In *SIGMOD Conference, Chicago, IL*. ACM, June 1988.

[Lyn83] Nancy A. Lynch. Multilevel atomicty - a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4), December 1983.

[Mey86] Norman Meyrowitz. Intermedia: the architecture and construction of an object-oriented hypermedia system and applications framework. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*. ACM SIGPLAN, September 1986.

[MHL+89] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. Technical Report RJ 6649, IBM Almaden Research Center, January 1989.

[Mos89] J.E.B. Moss. Addressing large distributed collections of persistent objects: The Mneme project's approaach. In *Second International Workshop on Database Programming Languages, Gleneden Beach, OR*. Morgan-Kaufmann, June 1989.

[MS88] J.E.B. Moss and S. Sinofsky. Managing persistent data with mneme: Issues and application of a reliable, shared object interface. Technical Report 88-30, COINS, University of Massachusetts at Amherst, April 1988.

[R+88] S. Rehm et al. Support for design processes in a structurally object-oriented database system. In *Second International Workshop on Object-Oriented Database Systems*, pages 80–96. ACM, 1988.

[RM89] Steven P. Reiss and Scott D. Meyers. Creating graphical languages in garden. Technical Report CS-89-02, Brown University, January 1989.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1988.

[Ska89] Andrea Skarra. Concurrency control for cooperationg transactions in an object-oriented database. In *Workshop on Object Based Concurrent Programming*. ACM SIGPLAN Notices, April 1989.

[Sto87] M. Stonebraker. The design of the Postgres storage system. In *The 13th VLDB Conference, Brighton, England*, August 1987.

[Sun87] Sun Microsystems, Mountain View, CA. *SunOS Reference Manual*, v4.0 edition, 1987. pg. 1644.

[VBD89] F. Velez, G. Bernard, and V. Darnis. The O2 object manager: An overview. Technical Report 27-89, Altair, Gip Altair, February 1989.

[Wei89] William Weihl. The impact of recovery on concurrency control. *ACM PODS*, pages 259–269, 1989.