The Soprano Extensible Object Storage System

Jung-Ho Ahn and Hyoung-Joo Kim

Department of Computer Engineering,

Seoul National University,

Shilim-Dong Gwanak-Gu, Seoul 151-742, KOREA

Tel: +82 2 880 1830/1826, Fax: +82 2 888 2016,

E-Mail: {jhahn,hjk}@oopsla.snu.ac.kr,

URL: http://oopsla.snu.ac.kr

The Soprano Extensible Object Storage System

Abstract

An efficient object manager, a middle layer on top of a storage system, is essential to ensure acceptable performance of object-oriented database systems, since a traditional record-based storage system is too simple to provide object abstraction. In addition, an object storage system - object managers in combination with storage systems - should be extensible to meet the various requirements of emerging applications. In this research, we design and implement an extensible object storage system, called Soprano, in an object-oriented fashion which has shown great potential in extensibility and code reusability. Soprano provides a uniform object abstraction and gives us the convenience of persistent programming through many useful persistent classes. Also, Soprano supports efficient object management and pointer swizzling for fast object access.

This paper investigates several aspects of the design and implementation of the extensible object storage system. Our experience shows the feasibility of using an object-oriented design and implementation in building an object storage system that should have both extensibility and high performance.

1 INTRODUCTION

In recent years, many object-oriented database systems have been developed and have become widely accepted in the next generation of telecommunications, Internet and financial applications around the globe. Due to the complexity of data management in such applications, key issues are performance and the requirement for a flexible and transparent object management environment. Thus, the commercial success of the object-oriented database systems largely depends on how well they meet these stringent requirements.

Contemporary relational database systems consist of two main modules: a query processor and a storage system. A query processor returns the result of a given query by translating it into a series of internal storage system calls. The low-level storage system provides data persistency and transaction management with full control of physical devices. In object-oriented database management systems (OODBMSs), however, it is no longer adequate for upper layers, such as a query processor, to call a low-level storage system directly. This is because the upper layers of an OODBMS should be adapted to the rich and extensible nature of the object-oriented data model directly, while a traditional relational storage

system supports only record-oriented data abstraction. That is, upper layers (if built directly on top of the relational storage system) would have to implement object abstraction, resulting in poorer performance due to increased complexity (Bancilhon et al., 1992).

To overcome this problem, most OODBMSs employ a middle layer, which is called an object manager, on top of the storage system. The objective of an object manager is to reduce the impedance mismatch between upper layers (e.g., object query processor) and lower layers (e.g., storage system) by implementing object abstraction using the facilities of the underlying storage system. We summarize the basic functionalities of an object manager as follows (Bancilhon et al., 1992):

1. To generate object identifiers
2. To create and delete persistent objects
3. To support object access method
4. To support object naming service

Besides the above features, an object manager is also involved in method binding, object versioning and object clustering.

Along this line, an efficient object storage system, the object manager together with the storage system, is essential to ensure a reasonable performance of OODBMSs. In addition, object storage systems need to simplify the addition and modification of application-specific functions, since new database applications differ from traditional ones in their requirements of operations and storage structures. That is, the extensibility of object storage system is the key to flexible object management.

The extensibility of an object storage system heavily depends on the system architecture. First, the system must be based on the architecture that can easily support the addition of new operations and storage structures. Second, the system should provide uniform interfaces for operations and its facilities. In addition, many of the basic architectural and performance tradeoffs involved in its design should be well understood. Considering the great potential of object-oriented paradigm in extensibility and reusability, it is evident that these prerequisites for an extensible object storage system could be achieved by an object-oriented design and implementation.

In this paper, we develop a high-performance extensible object storage system for next-generation database applications, called Soprano (SNU Object Persistent Repository with Advance Novel Operations). One of the key design features is to provide an object abstraction of all facilities (like B+-tree index) as well as persistent data for various advanced database applications which demand high performance. By treating everything as an object, the system as well as application programs are simplified. Soprano also helps persistent programming through many useful persistent classes and full support of C++ features like virtual functions and virtual base classes[1]. Soprano supports efficient object management and pointer swizzling for fast object access.

[1] In order to support virtual functions and virtual base classes, schema information should be provided for the system.

A number of research prototype object managers have been developed. Examples include Mneme at Massachusetts University (Moss, 1990), E Persistent Virtual Machine (EPVM) (Schuh et al., 1990) implemented on top of Exodus storage system (EXODUS Project Group, 1991), and ObServer (Hornick & Zdonik, 1987) at Brown University. There are also many object managers of commercial OODBMS products including O2 (Bancilhon et al., 1992). However, these object managers do not consider the extensibility of itself. They also use a storage system of a relational DBMS for object persistency and thus, they cannot provide object concept uniformly over all around of the system and may cause unnecessary overhead.

The remainder of this paper is organized as follows. Section 2 through 4 present the overall architecture of Soprano and describe the features and details of implementation techniques for major components. In Section 5, we evaluate the performance of our system based on the OO1 benchmark (Cattell & Skeen, 1992) and conclusions are given in Section 6.

2 THE ARCHITECTURE OF SOPRANO

2.1 Architecture Overview

Figure 1 illustrates the overall (single-site) architecture of Soprano in terms of the major sub-modules that constitute the system. The lowest level is the objectbase that controls physical storage devices. The page cache and the object cache manage main memory buffers for pages and objects, respectively. The transaction manager, the lock manager, and the log manager coordinate concurrent object accesses and provide recovery capabilities.

Most existing object managers including that of O2 OODBMS implemented an object abstraction using the facilities of the relational storage systems. Although these systems made the most of existing storage systems, the two-layered architecture may suffer from performance degradation due to the complexity from unnecessary concept like a file. Instead, Soprano is one system, not two separate subsystems with arbitrary boundaries between them. Such tight integrated architecture can control consistently the flow of persistent objects from a physical device to an object cache. This also provides the extensibility such that a new type of system objects can be easily added, and leads to high performance by avoiding unneeded overhead. In addition, the uniformly designed system has good advantages in software engineering aspect like maintainability.

For example, Soprano, unlike the existing systems, does not support file concept directly. This is because a user should be able to access persistent objects without any intervention by file operations like ‘open’ and ‘close’. That is, a traditional file concept is required only for the means to cluster and sequentially access objects in object storage systems. Moreover, a system should be able to store objects

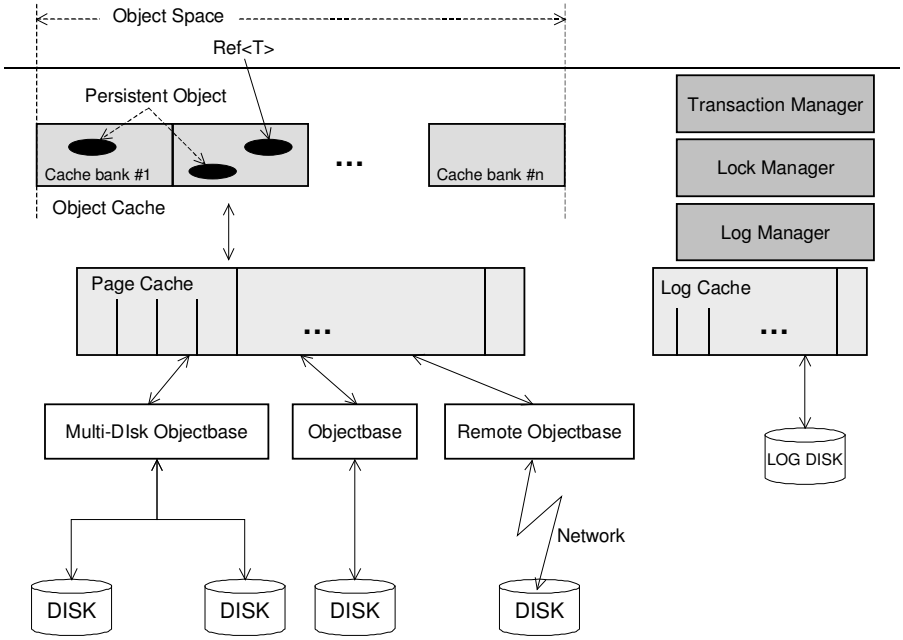

Figure 1: Soprano Architecture

of various types in a file for their clustering. However, this is a contrast to a general file concept where a file typically has data of the same type. As such, Soprano provides *object groups* for clustering and supports a traditional file concept through a persistent class library on top of the system.

In Soprano, as mentioned above, everything is an object including supporting facilities like a BLOB structure for multimedia data, a B+-tree index, a file, and even directories for object naming service as well as user persistent data. With Soprano, therefore, the system itself is viewed as a collection of objects, allowing all facilities to be accessed in the same manner as persistent objects. This kind of uniformity of object concept provides a user with easy persistent programming without any impedance mismatch between a programming language and an object storage system.

2.2 Client-Server Architecture

Soprano follows a multi-process architecture where each client process runs with its own server process that services requests from the client. Figure 2 illustrates the client-server architecture of Soprano, where each client and server process has the internal structure as shown in Figure 1. The granularity of data shipping between a client and a server is a page or a set of pages. The page server architecture avoids network overhead and allows clients direct (shared) access to data pages. Consequently, indexing and BLOB operations can be executed at a client. The Soprano server consists of several request brokers: a page cache request broker, an objectbase request broker, and a lock request broker which are tied to the

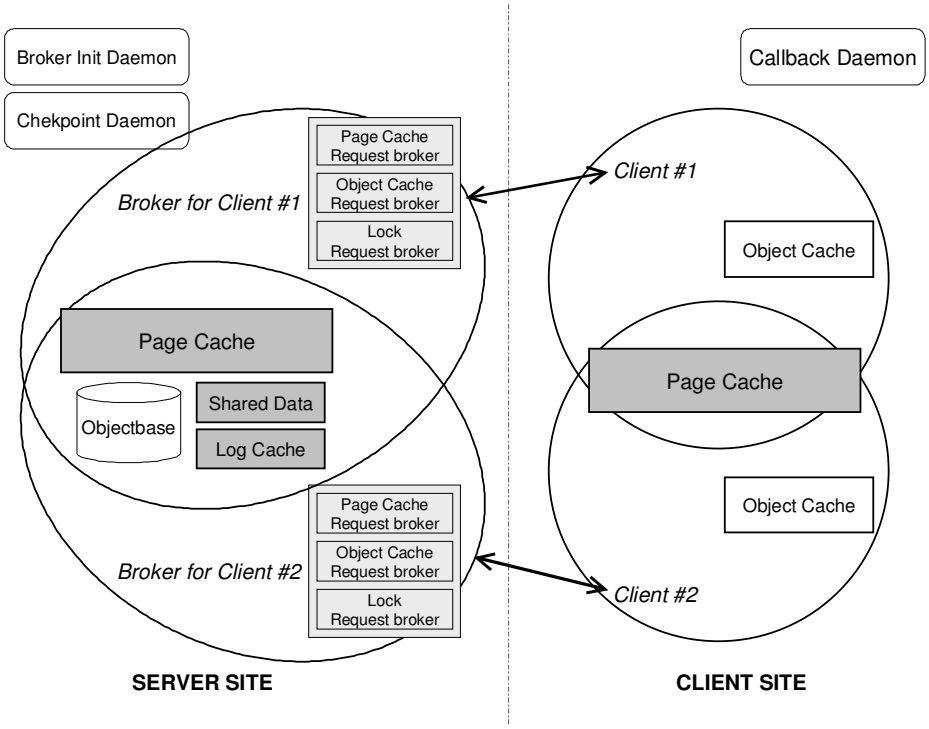

Figure 2: Clients and Servers of Soprano

functions of a page cache, an objectbase, and lock management, respectively. When a client process starts up, it establishes connections with its own request brokers spawned by a `Broker Init Daemon` and handles service requests, which cannot be carried out by itself, through brokers. That is, a server operates page allocation/deallocate, read/write, and lock request/release on behalf of its client and returns results to the client. Server processes share system resources like a page cache, a log cache, objectbases as well as global system objects including directories. Likewise, all client processes at the same node share system objects including a page cache. A callback daemon helps Soprano support inter-transaction caching, which together with data sharing between clients improves system performance by reducing data traffic over a network.

2.3 The Objectbase

The objectbase of Soprano provides the abstraction of a set of sequential pages over a physical device. The objectbase is responsible of allocating and deallocating pages. It also supports a segment group as one of the units of clustering. A segment group is a set of segments and a segment is a set of pages that lie sequentially on a disk. The Soprano objectbases provide the same abstraction and operations independently of the types of physical devices and thus, improve the extensibility of the system. That is, a new device type can be added to Soprano by just deriving the subclass for the new device from the `ObjectBase` class (or its already-existing subclasses) and redefining operations to handle the device.

For example, Soprano provides a `RemoteBase` class that handles a remote disk.

While the objectbase provides the abstraction of a set of pages and operations on it, users need the abstraction of a set of persistent objects on a database. The `OBase` class of Soprano implements operations on persistent objects on top of an objectbase and provides the abstraction of a set of persistent objects for users.

The obase is responsible for the creation and deletion of objects in a database and supports an object group for clustering related objects. An object group of an obase is a set of related objects and provides a user-level view of a segment group of an objectbase. Users can store related objects together by specifying an object group where the objects should be placed at their creation time. An object group also allows a user to sequentially access persistent objects in an object group.

3 OBJECT ACCESS

In this section, we present the design and implementational features concerned with object access in Soprano.

3.1 Pointer Swizzling

Soprano employs a physical OID representation where an OID encodes a physical location of the object referred to by it. Although the physical OID scheme has a disadvantage of object migration due to the lack of location independency, it allows faster access of persistent objects than a logical OID policy which requires mapping between logical OIDs and their physical addresses (Khoshafian & Copeland, 1986). As more objects are moved, logical OID schemes may degrade the overall performance of the system more due to the loss of caching effects of mapping table (Eickler et al., 1995). In addition, a typed logical OID, which has a class information, makes schema evolution difficult. Soprano allows persistent objects to be moved around by forwarding marks (Moss, 1990; Bancilhon et al., 1992), where a new physical address of the moved object is stored in the original location and thus, the object can be accessed indirectly with the new address.

Although OID provides efficient navigational access by direct representation of the relationship between objects, the objects brought into the main memory can be accessed only after the OID translations into the corresponding virtual memory addresses. To help the translation, object storage systems usually maintain mapping tables to locate objects cached in main memory. However, it is a performance penalty to check the residency and compute the in-memory address on every access. To solve this problem, many object storage systems replace in-memory OID references with virtual addresses. This concept is known as pointer swizzling (White, 1994). Pointer swizzling can improve the

performance of object access by skipping the lookup-table search, particularly in CPU-intensive applications. It can also give transparent access to persistent objects just as for transient objects. Soprano employs a software swizzling scheme (precisely edge marking method) (White, 1994) to check whether a pointer is swizzled or not and follows a lazy swizzling policy to avoid unnecessary swizzling. Although this software checking induces a little overhead to access objects, it does not impose a limit on the size and the structure of an object identifier. Soprano also unswizzles swizzled pointers to OIDs when a pointer to an object is no longer used or when the object is displaced from the object cache. This unswizzling allows Soprano to deliberately replace recently-unused objects in the cache at any time.

3.2 Object Access Interface

Soprano provides a smart pointer interface, the `Ref` handler of ODMG 2.0 (Cattel et al., 1997) standard as an object access interface. The object handler of Soprano has two address fields, an OID (disk address) and a `memptr` (in-memory address) of an object referred to by the handler. That is, the `Ref` handler of Soprano maintains both information separately while a traditional implementation stores a swizzled address in the OID field. Thus, the object handler of Soprano requires a little more memory space. However, the space overhead of our method is relatively smaller with the current hardware technology and the advantages of our implementation can be elaborated as follows.

First, Soprano may efficiently perform swizzling check. Usually, software swizzling is done by tagging reference bits and thus, the check requires bit operations. However, Soprano can perform this check just by comparing the `memptr` with `NULL` value without any additional bit operations. Especially on a RISC architecture system, this comparison can be just done with two machine operations and the `memptr` field is used again to access an in-memory object right after the comparison. Therefore, the overhead of software swizzling check is as small as a few machine cycles[2].

Second, Soprano can unswizzle - that is, restore a virtual address to an OID - easily using the OID field for managing a reverse reference list. As shown in Figure 3, every cached object maintains the swizzled `Ref` handlers that refer to the object itself with a doubly linked list. When a `Ref` handler is swizzled, the handler is inserted into the reverse reference list of the object addressed by the handler and when the handler is unswizzled, it is deleted from the list. Also, the OID field of a `Ref` handler is set to its corresponding node of the reverse reference list and reversely, the node pointer is set to the handler. This structure makes the deletion of a handler from the reverse reference list easy when the handler is to be unswizzled. Also, an object can be displaced easily from a cache, since all `Ref` handlers that refer the object can be unswizzled simply by traversing its reverse reference list.

---

[2] We performed experiments to evaluate the performance of the software swizzling check method on 40MHz Sparc machine with Soprano. In this experiment, the average overhead of checking was 4.3 machine cycles.

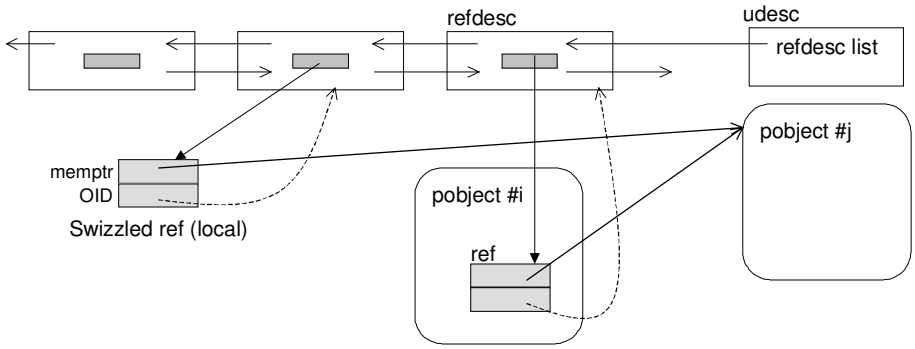

Figure 3: The Reverse Reference List

Soprano uses class definitions to support the virtual function and the virtual base class of C++ language. First, Soprano initializes hidden pointers (Ellis & Stroustrup, 1990) of an object by calling its constructor when fetched into memory from a storage device[3]. The hidden pointers include a pointer to a virtual function pointer table and a pointer to a virtual base class part. However, the initialization of hidden pointers does not solve the entire virtual base class problem. For instance, when a class `A` is the virtual base class of `B`, the `Ref` handler to an instance of `A` cannot refer to an object of `B`. This is because the `A` part of the object of `B` is located in the middle of the object rather than at the beginning of the object. That is, the `A` portion of an object of `B` cannot be accessed correctly through the `memptr` of the handler since the `memptr` field points the start of the object. To solve this problem, Soprano provides a `VRef` handler that caches the address of the part of a virtual base class in it.

3.3 In-memory Object Management

Soprano manages in-memory objects in the object cache, which consists of one or more object cache banks. The object cache coordinates the fix and unfix of persistent objects in memory with object cache banks that controls the allocation and deallocation of memory space. We split the object cache into several cache banks due to the following reasons. The object cache should handle fragmentation as well as heavy memory allocation and deallocation since it should be able to manipulate objects of various sizes. Therefore, it is not efficient to allocate and free a space in one large cache space. The problem of fragmentation becomes more critical when the cache space is shared by clients. In that case, this approach can give more opportunity for adjacent memory spaces to be freed together by assigning different cache banks to each client in a round-robin form.

The object cache maintains the descriptors of all used memory slots. As shown in Figure 4, the

[3] A constructor for initialization of hidden pointers is generated automatically when the class is registered.

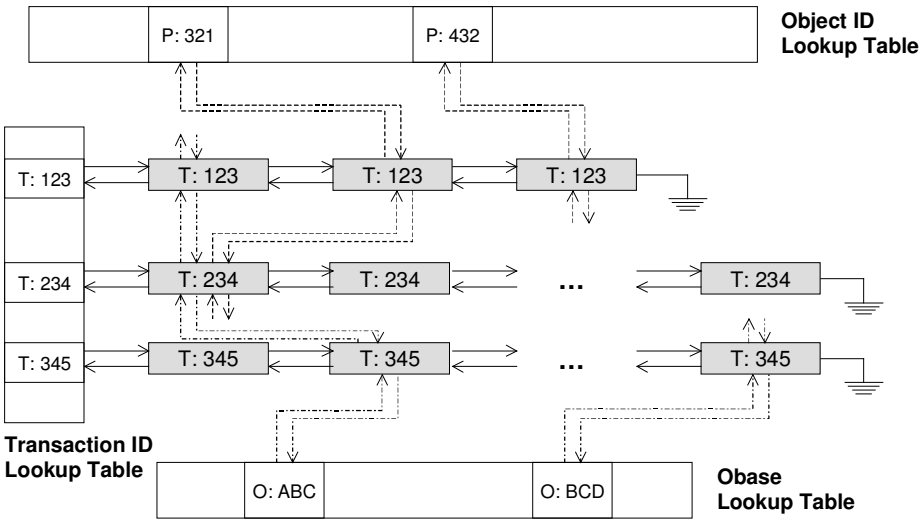

Figure 4: The management of descriptors of used memory slots

descriptors whose objects are accessed by the same transaction are connected with a doubly linked list. Similarly, each descriptor is inserted into extra two doubly linked lists, one on an obase where it is stored, one on its OID (actually on the hashed value of the OID). As mentioned above, the object cache bank handles the allocation and deallocation of memory space - after this, we refer it `RBlock` - into which persistent objects are read from disk. Here, the `RBlock` is allocated over multiples of the minimum block size[4].

Every free `RBlock` has its own `FreeBlkDesc` and all valid objects in the free block are connected with a doublely linked list. This list allows unfixed objects to be reused later if the block is not reassigned to other objects yet (see Figure 5 and Figure 6). The `BoundaryTags` of a `RBlock` have `NULL` value when the `RBlock` is used. When the block is not used, the boundary tags points the `FreeBlkDesc` of the block. Checking the boundary tags of neighbor blocks, a free `RBlock` can be easily coalesced with its adjacent blocks whenever possible. Similarly, a used block keeps the `UsedBlkDesc` field to point its `UsedBlkDesc` and the next two fields are used to link valid objects within the block when it is free. As shown in Figure 5, the `FreeBlkDescPool` manages lists of free `FreeBlkDescs` according to the size of free space in order to quickly find a best-fit space and to minimize fragmentation.

3.4 Method Call Handling

The interface of Soprano is expressed as a C++ class library and object codes are linked with the run-time library to produce an executable application. That is, user-defined classes (including their methods) associated with the application as well as the system classes are linked in the executable and the

[4] This parameter can be configured when building the Soprano system.

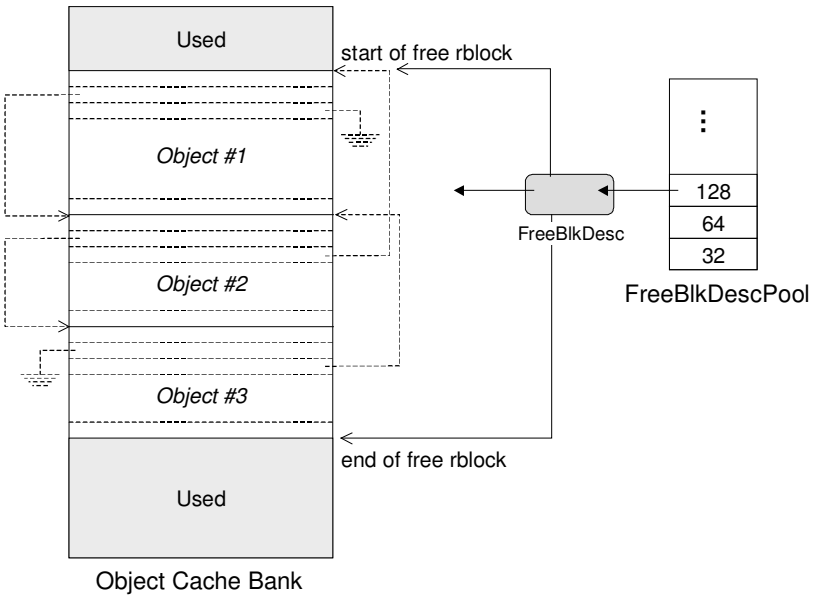

Figure 5: The Object Cache Bank

application can freely apply methods of user classes to persistent objects.

However, when a method is used in a query, the object storage system takes the responsibility on method execution since all of the persistent classes of databases cannot be linked with a query processor. That is, a query processor should be able to bind a method dynamically. Soprano supports the dynamic method binding by using a dynamic linking loader of the underlying operating system. For this, every newly defined class and its methods are compiled into a position independent code (PIC) and then inserted into a dynamic library of persistent classes. When a method is called dynamically, Soprano finds the address where that method is loaded and executes it. Although this allows a dynamic method binding, it is still difficult to pass arguments of various types dynamically. To solve this problem, Soprano uses an agent function that passes arguments of the `void*` type to its corresponding method with appropriate type conversions. An agent function is generated automatically when a new class is imported to a database. For example, given a method `int f(int, double)` of a class A, an agent function, `void f'(void*, void*, void*, void*)` is generated and compiled into the dynamic library. The first two arguments of the agent function are pointers to an object and to a return value, respectively. The last two arguments are the first and second arguments of the method `f`[5].

3.5 The Persistent Object Class Hierarchy

A persistent object of Soprano should be an instance of a persistence-capable class and a class

[5] Actually, Soprano passes arguments as an array of `void*` terminated by a `NULL` pointer to easily handle a varying number of arguments.

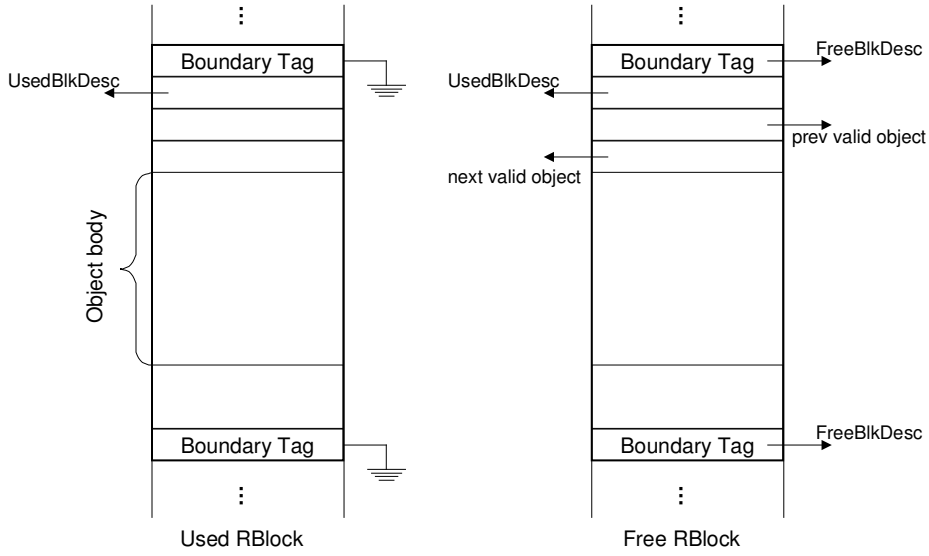

Figure 6: The `Rblock` structure

becomes persistence-capable by tracing its ancestry to a persistent base class, `PObject`. However, the lifetime of an object is determined at the time the object is created. Soprano allows a user to specify the lifetime of the object by giving a special location as an additional argument of the `new` operator, which is overloaded to accept additional arguments specifying where the newly created object should be. The `TransientBase` database is used to create objects of transient lifetime. Transient objects are destroyed when the process in which the objects are created terminates.

Figure 7 shows the built-in class hierarchy of Soprano, from which a user can derive his or her own classes. The class `FileObject` and `IndexObject` provide the abstraction of a traditional file and index, respectively. The `FileObject` has a subclass, `SequentialFile` that stores data sequentially. Within `SequentialFiles`, Soprano supports sequential files for both variable size and fixed size data.

Soprano defines a B+-tree index class below IndexObject. However, B+-trees cannot efficiently support queries on class hierarchies since their answer should be restricted to target classes as well as search predicates. For class indexing, the `CHIndex` class implements the class-hierarchy index (Kim et al., 1989). In another research (Ahn et al., 2000), we develop a new indexing framework, Index Set that gives us maximal performance gain with minimal space and update overhead using `CHIndex`.

Unconventional data - text, graphics, images, video, and so forth - are handled in Soprano as objects of class `LargeObj`. The class `LargeObj` defines the operations for storing and retrieving very large (virtually unlimited) size data. Soprano also provides collection that can contain an arbitrary number of elements. A set of collection classes, such as a set, a bag, a list, and an array, can be derived from them. As an application, several classes of multimedia objects were implemented using `LargeObj` and

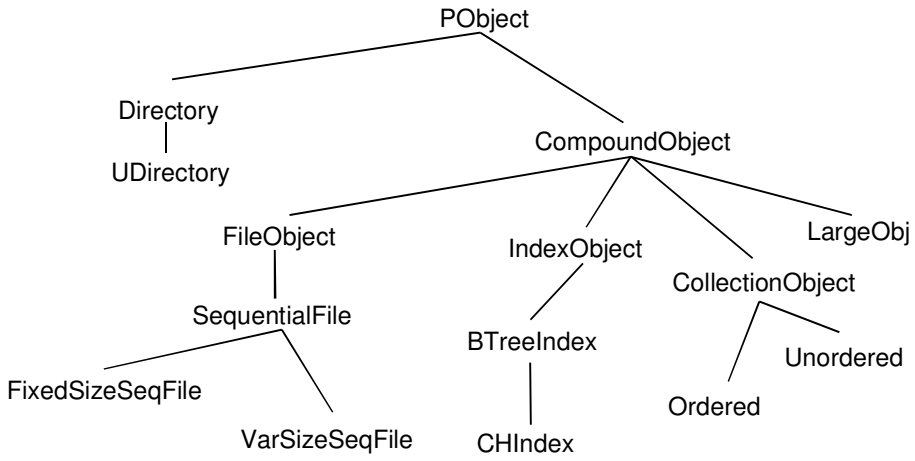

Figure 7: The `PObject` Class Hierarchy

collection classes in order to handle text, 2D spatial data, image and video efficiently (Park & Kim, 1998). These classes could be easily incorporated into Soprano, since the built-in classes can be used and extended in the same way as user-defined classes without the detailed knowledge of Soprano.

Object-oriented database applications usually will begin processing by accessing one or more critical objects, which anchor webs of objects that are then fetched into the applications as they are dereferenced. Therefore, the naming of these 'root' objects and their later retrieval by name is necessary for this startup (Cattel et al., 1997). Soprano provides two, flat name spaces per objectbase through a system directory: one is for objects and the other is for object groups. All names in a particular name space should be unique and an object (or object group) can have a maximum of one name. A name associated with an object or an object group is automatically deleted when the named object or the object group has been destroyed. A user can define his or her own name space using a directory class, `UDirectory`. Unlike the system directory, the `Udirectory` allows an object (or object group) to have more than one name. However, the removal of a dangling name to an object (or object group) which has been deleted is an application responsibility.

Soprano also support object versioning in an ODMG C++ OML (Object Manipulation Language) environment through the SOP object version system (Lee & Kim, 1999). Our object version system provides object version facility in ODMG standard and gives a solution for dynamic binding in strongly typed languages like C++. It also minimizes the performance overhead on the Soprano.

4 TRANSACTION PROCESSING

Soprano provides page-level, two-phase locking and page-level, redo-only physical logging for access to the database. The page cache follows ¬steal/¬force cache policy (Haerder & Reuter, 1983) in order to

support redo-only logging. The log and lock managers of Soprano are embedded in the page cache. Therefore, upper layers, that access a database via the page cache, do not have to do additional jobs for data consistency and durability since locking and logging are transparently supported by the page cache.

Although the page-level locking may reduce concurrency, this criticism, however, may no longer be as valid as in object-oriented database applications where data sharing among applications is relatively lower than that of traditional applications. Rather, the page-level locking policy can reduce locking overhead by decreasing the number of locking requests made to a server. A page-level lock request may also be piggy-backed on the page request to avoid additional message passings for locking since the granularity of locking is the same as the unit of data shipping (Rahm, 1991). It is also easy to apply the page-level locking to a page-server architecture like Soprano (Dewitt & Maier, 1990).

In addition, the redo-only recovery scheme through ¬steal/¬force cache management allows a client to abort a transaction efficiently and provides fast recovery from a system failure especially in a page-server architecture. The redo-only logging makes the recovery process more efficient by accumulating repeated modifications on the same page (White & Dewitt, 1995). This scheme also speeds up the execution of transactions and allows high utilization of I/O bandwidths of network and log devices reducing the commit time[6], since all log records generated at a client are transferred to a server when committed. Soprano allows the page cache to swap out uncommitted dirty pages to a swap device in order to overcome the problem that a ¬steal/¬force cache policy may make a page cache full and degrade system performance (Haerder & Reuter, 1983). This page swapping does not affect pointer swizzling, since the swizzling occurs in an object cache and not in a page cache.

Objectbases are also concurrency controlled and recoverable. The `ObjectBase` class uses physical latches for their concurrency control rather than locks to improve concurrency on hot spot data such as a page allocation table. Soprano employs an operational logging and a redo/undo scheme for the recovery of objectbases. The transaction manager employs a conventional database recovery algorithm, the ARIES (Mohan et al., 1992) for a redo/undo logging. Although Soprano uses two distinct recovery algorithms, it is not problematic since the algorithms are applied to two different data.

The transaction manager supports a pending action (Mohan et al., 1992) for operations that are not easy to recover such as freeing pages. In addition, a compensation action is used to rollback the updates by a nested top-level action (Mohan et al., 1992) in case the parent transaction aborts. For example, when a transaction in which a new object name is inserted into the system directory is aborted, the name should be deleted again by issuing the compensation action since the directory operation is wrapped into a nested top-level action. Soprano increases concurrency on hot spot data (like a system directory object or a schema manager) by applying nested-top actions.

A commit and abort of a transaction in Soprano are accomplished as follows. At commit time, the

---

[6] Generally, network and disk devices suffer from the frequency of usages rather than the amount of usage data.

transaction manager forces the page cache to log all pages modified by the transaction and update the `commitLSN`[7] of each page to the LSN (log sequence number) of the corresponding log record. Therefore, a page always keeps the LSN - the unique key of the log record that describes the last update applied to that page. This ensures efficient recovery. A page should be re-read into memory before logging if the page is swapped out. After finishing the commit operations of the page cache, the transaction manager writes the pending operations in a commit log record and flushes out all log records of the transaction. The last step of the commit procedure is to execute the pending operations and release all locks held by the transaction. In the case of a nested top-level transaction, the state is taken by its parent transaction. The transaction manager does not generate a commit log record if the transaction does not generate any logs so that a read-only transaction can commit quickly.

To abort a transaction, the transaction manager first discards all pending operations and causes the page cache to undo modifications. The page cache restores the old values by reading the log records designated by the `commitLSN` of each page. If the `commitLSN` is -1, the page cache throws away the dirty page, since it means that the old page is stored in a storage device. For recovery in objectbases, the transaction manager reads logs backwards to extract the operations done by the transaction and undoes them. Finally, all logs including an abort log record are written to a log device and all locks are released.

The transaction manager uses fuzzy checkpoints so as not to disrupt services during checkpoints. The transaction manager wakes up the checkpoint daemon every time the log grows by a predefined number of records. For checkpoint, Soprano employs a variation of the ARIES algorithm (Mohan et al., 1992). During checkpointing, the page cache flushes out pages that live changed in it long time to further save log space.

A recovery process of Soprano consists of the following sequence of three steps. In the first analysis step, the transaction manager searches the list of all transactions that have not completed and identifies changes made by committed transactions that are not reflected in the disk copy of the page. After the first step, then, the transaction manager performs 'redo's for these changes and repeats histories (Mohan et al., 1992) for operation logs of objectbases. The last undo step is needed to reverse the changes by objectbases in aborting transactions. The detailed algorithm is similar to the ARIES.

The transaction management at clients is essentially the same with that of servers, but we can highlight some of the differences as follows. First, at commit time, a client transfers all pages updated by the transaction to a server piggy-backed on log records. However, the client caches locks rather than releases them to minimize its re-requesting of the same lock. To abort a transaction, the client throws away dirty pages and performs compensation actions after informing its actions to the server.

---

[7] Here, the `commitLSN` field is not the same as that of (Mohan, 1990).

5 PERFORMANCE EVALUATION

In this section, we present the results of a performance study. The OO1 Benchmark (Cattell & Skeen, 1992) was chosen for our performance study. The OO1 object operation benchmark, which was developed to evaluate scientific and engineering applications, exhibits the aspect of navigational object access in general object-oriented database applications. The OO1 benchmark database consists of `Part` and `Connection` objects, where every `Part` is connected to three other `Parts` via `Connections`. The connections between `Parts` are selected randomly to produce 90%-1% clustering factor: 90% of the connections are to the closest 1% of `Part` objects.

We used the `traversal` and `insert` operations of the OO1 benchmark as workload and the small database of 20,000 `Parts` and 60,000 `Connections` is used. The `traversal` operation of the OO1 benchmark recursively accesses all `Parts` connected to a randomly selected `Part` object, up to 7 hops. The insert operation creates and inserts 100 new `Part` objects connecting each new `Part` to three other `Parts` with the same notion of closeness mentioned before.

All of the benchmarks were run on a pair of Sun workstations on an Ethernet. A Sun Sparc2 with 32 Mbytes of memory was used as a server and a client process was run on a Sun Sparc IPC configured with 16 Mbytes of memory and 200 Mbytes disk drive. A single 1 Gbytes disk drive in the server was used to hold the OO1 benchmark database. The benchmarks were run on SunOS 4.1.3. The page cache was set to 400 Kbytes in both the server and client nodes and the object cache of a client was set to 5 Mbytes.

We ran the `traversal` operation 30 times, and Figure 8 shows the response times and the number of object faults for each traversal. The solid line in this graph represents total running time for each benchmark and the dotted line represents the number of object faults (i.e., the number of object misses in the object cache). The numbers are given in 1/1000.

As the figure shows, it is clear that the performance is largely a function of the number of objects misses in the cache. It therefore comes as no surprise that the performance at cold time was relatively bad due to the high object fault ratio. However, once much of objects needed by each traversal had been cached in the client, a traversal took less than 3 seconds. Considering a `traversal` operation accesses 3,280 connected `Parts`, this result indicates that Soprano can access more than 1,000 objects in one second. This performance result satisfies the requirement of most scientific and engineering applications (Cattell & Skeen, 1992). The bumpiness seen in the graph is largely due to the object clustering and irregular traversal paths. The results of the traversal also confirm the fact that efficient object caching is the key to the performance of object access.

Turning to the insert operation, there is much less difference between the cold or warm results, since the client cannot cache many objects enough to improve the hit ratio during the experiment. Table 1 presents the results for the insert operation.

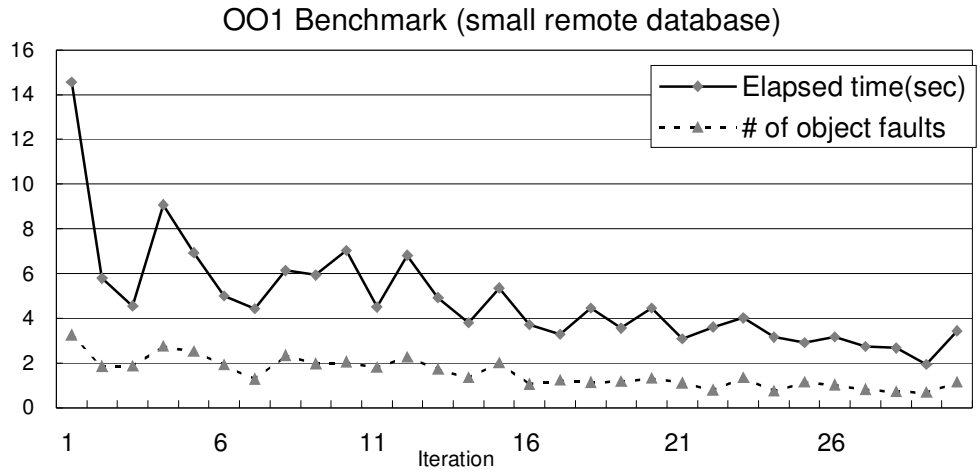

Figure 8: The result of `Traversal` operation

	Iterations									
Elapsed Time	1	2	3	4	5	6	7	8	9	10
(sec)	5.83	5.38	5.01	5.40	4.99	5.01	5.00	5.19	4.80	4.81

Table 1: The results of `insert` operation

6 Conclusions

Soprano is a high-performance object storage system for various advanced database applications. Soprano provides a uniform object abstraction and supports efficient object cache management and pointer swizzling/unswizzling for fast object access. By treating everything as an object, Soprano allows us the ease of persistent programming without any impedance mismatch between a programming language and an object storage system. In addition, Soprano has extensibility from object-oriented design and implementation.

Soprano follows the multi-process and page server architecture. Every client of Soprano is connected to its own request brokers, which forward requests from the client to an appropriate server module. A callback daemon helps Soprano support inter-transaction caching which, together with data sharing between clients, improves system throughput by reducing data traffic over a network significantly.

Soprano provides page-level, two-phase locking and page-level, redo-only physical logging for access to a database. The page cache of Soprano supports page swapping to overcome the disadvantage of redo-only recovery scheme. Soprano also employs an operational logging and a redo/undo scheme for the recovery of objectbases.

In the performance study, Soprano offered high performance enough to meet the performance requirement of engineering applications such as CAD. This result shows the feasibility of using an object-oriented design and implementation in building an object storage system that should have both extensibility and high performance.

In the future, we would like to extend Soprano to a distributed object storage systems.

References

Ahn, J.-H., Song, H.-J., & Kim, H.-J. (2000). Index Set: A Practical Indexing Scheme for Object Database Systems. *Data & Knowledge Engineering*, 33(3).

Bancilhon, F., Delobel, C., & Kanellakis, P., editors (1992). *Building an Object-Oriented Database System: The Story of O2*. Morgan Kaufmann Publishers.

Cattel, R. G. G. et al., editors (1997). *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers.

Cattell, R. G. G. & Skeen, J. (1992). Object Operations Benchmark. *ACM Transactions on Database Systems*, 17(1).

Dewitt, D. J. & Maier, D. (1990). A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems. In *Proceedings of the International Conference on Very Large Data Bases*, Brisbane, Australia.

Eickler, A., Gerlhof, C. A., & Kossmann, D. (1995). A Performance Evaluation of OID Mapping Techniques. In *Proceedings of the International Conference on Very Large Data Bases*, Zurich, Switzerland.

Ellis, M. A. & Stroustrup, B. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company.

EXODUS Project Group (1991). EXODUS Storage Manager Architectural Overview.

Haerder, T. & Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4).

Hornick, M. F. & Zdonik, S. B. (1987). A Shared, Segmented Memory System for an Object- Oriented Database. *ACM Transactions on Office Information Systems*, 5(1).

Khoshafian, S. N. & Copeland, G. P. (1986). Object Identity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).*

Kim, W., Kim, K.-C., & Dale, A. (1989). Indexing techniques for object-oriented databases. In *Object-oriented Concepts, Databases, and Applications*. Addison-Wesley Publishing Company.

Lee, S.-W. & Kim, H.-J. (1999). Object Versioning in an ODMG-compliant Object Databases System. *Software Practice and Experience*, 20(3).

Mohan, C. (1990). Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. In *Proceedings of the International Conference on Very Large Data Bases*, Brisbane, Australia.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., & Schwarz, P. (1992). ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1).

Moss, J. E. B. (1990). Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems*, 8(2).

Park, D.-J. & Kim, H.-J. (1998). Design and Implementation of Class Libraries Supporting Multimedia Data in SOP OODBMS. *Journal of KISS (B)*, 25(8).

Rahm, E. (1991). Concurrency and Coherency Control in Database Sharing Systems. Technical Report ZRI 3/91, Universit"at Kaiserslautern.

Schuh, D., Carey, M. J., & DeWitt, D. J. (1990). Persistence in E Revisited_Implementation Experiences. Technical Report #957, Computer Science Dept., University of Wisconsin-Madison.

White, S. J. (1994). *Pointer Swizzling Techniques for Object-Oriented Database Systems*. PhD thesis, Computer Science Dept., University of Wisconsin-Madison.

White, S. J. & Dewitt, D. J. (1995). Implementing Crash Recovery in QuickStore: A Performance Study. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, CA USA.