SNAP: Efficient Snapshots for Back-in-Time Execution

Liuba Shrira
Microsoft Research
and
Department of Computer Science
Brandeis University
Waltham, MA 02454
liuba@cs.brandeis.edu

Hao Xu
Department of Computer Science
Brandeis University
Waltham, MA 02454
hxu@cs.brandeis.edu

Abstract

SNAP is a novel high-performance snapshot system for object storage systems. The goal is to provide a snapshot service that is efficient enough to permit "back-in-time" read-only activities to run against application-specified snapshots. Such activities are often impossible to run against rapidly evolving current state because of interference or because the required activity is determined in retrospect.

A key innovation in SNAP is that it provides snapshots that are transactionally consistent, yet non-disruptive. Unlike earlier systems, we use novel in-memory data structures to ensure that frequent snapshots do not block applications from accessing the storage system, and do not cause unnecessary disk operations. SNAP takes a novel approach to dealing with snapshot meta-data using a new technique that supports both incremental meta-data creation and efficient meta-data reconstruction.

We have implemented a SNAP prototype and analyzed its performance. Preliminary results show that providing snapshots for back-in-time activities has low impact on system performance even when snapshots are frequent.

1 Introduction

Low-cost disk storage makes it feasible for storage systems to retain and keep large on-line historical snapshots of data. Access to historical snapshots enables a range of new applications, based on what we call "back-in-time execution", where read-only applications run against chosen snapshots instead of the current state. These applications can perform activities that would be impossible to run against rapidly evolving current state because of interference, or because the required activity is determined only in retrospect. Examples include simulation replay and analysis, security audits, and data mining.

Many databases and file systems support snapshots for data recovery purposes, but they are poorly suited for back-in-time execution, because they use techniques that perform well only under the assumption that snapshots are infrequent. To enable new applications, persistent snapshots must be substantially more efficient than they are now.

SNAP is a novel high-performance snapshot system for object storage systems. It supports expressive back-in-time execution where applications can run general programs on selected snapshots. A key innovation is that, unlike in earlier systems, SNAP provides snapshots that are transactionally consistent, yet non-disruptive. They do not prevent applications from accessing the storage system even when snapshots are frequent; and they do not cause unnecessary disk update. In this way, SNAP avoids performance problems common to current snapshot systems. Aged snapshots are eventually discarded, or can be selectively retained at additional cost.

At any time, an application may ask SNAP to launch a snapshot. The storage system creates the snapshot, and names it for later reference. To construct the snapshot, the storage system lazily copies objects to an *archive*. Unlike in earlier approaches, novel in-memory data structures of SNAP make it possible to create a snapshot without interfering with ongoing activities that could potentially overwrite and lose snapshot states, and to build multiple snapshots simultaneously. This allows SNAP to support frequent consistent snapshots with minimal impact on system performance at the minimal cost of memory for the data structures.

To provide efficient access to objects, object storage systems cluster small related objects on disk pages, and use page tables to find those objects. To provide this performance benefit to application programs accessing snapshots, SNAP copies entire pages into the archive and uses snapshot page tables to find snapshot objects, different from systems that archive objects [14, 9]. Archiving snapshot pages instead of objects simplifies system design and allows general programs to access snapshots efficiently but it requires extra archiving space. Given current disk technology trends, this trade-off seems worthwhile.

A problem for snapshot archiving is that snapshot page tables change as pages migrate from the storage system to the archive. One challenge in designing an efficient snapshot system is how to manage this changing snapshot metadata efficiently. A key innovation behind SNAP is a new efficient technique, that supports both incremental page table updates and fast page table reconstruction.

SNAP has been implemented as a subsystem in the Thor object storage system [8]. We analyzed snapshot access performance, and how taking snapshots affects the storage system performance as the snapshot frequency varies under extended OO7 benchmark [2] workloads. Our performance results show that on most workloads, with additional inmemory data structures and a separate archive disk, SNAP provides efficient snapshot creation and storage with minimal impact on the latency of transactions even when snapshots are frequent. Furthermore, back-in-time execution is reasonably efficient on both recent and old snapshots.

The paper makes the following contributions. It describes the first high-performance consistent snapshot system based on new archiving techniques that support expressive back-in-time execution of general code. It describes a specific implementation in an object storage system, analyzes its workload-related performance trade-offs, and presents experimental evidence supporting our claims.

2 Back-in-time execution

We consider the requirements for back-in-time execution. Details on example applications that could potentially benefit from back-in-time execution in SNAP can be found in [16].

We assume that applications are structured as sequences of *transactions* accessing a storage system. *Back-in-time* execution is a capability of a storage system where the application runs transactional programs against read-only snapshots, instead of against the rapidly evolving current storage state. For back-in-time execution to work, snapshots must reflect transactionally consistent historical states satisfying the same invariants as the current storage state.

An application should be able to choose which snapshots to access. Providing snapshots at ”some time in the past” as is done in some systems does not offer sufficient time-accuracy desired by some applications.

In order to provide greater flexibility to snapshot accessing applications [16], applications should have the capability to run on a snapshot general programs rather than SQL-based access methods provided by versioned databases [14].

SNAP supports the requirements for back-in-time execution by providing two operations: a request to take a snapshot (*snapshot request, or declaration*), and a request to access a snapshot (*snapshot access*). Informally, an application takes a snapshot by asking for a snapshot ”now”. This snapshot request is serialized along with other transactions and other snapshots. That is, a snapshot reflects all state-modifications by transactions serialized before this request, but does not reflect modifications by transactions serialized after. A snapshot request returns a snapshot *name* that applications can use to refer to this snapshot later. For simplicity, we assume snapshots are assigned unique sequence numbers that correspond to the order in which they occur. A snapshot access request specifies which snapshot an application intends to use for back-in-time execution. The request returns a consistent set of object states, allowing the read-only transaction to run as if it were running against the current storage state.

A snapshot system must be efficient to be useful. It must not adversely affect the performance of the applications that request snapshots, and it must not disrupt transactions on current storage state. Back-in-time transactions should also run efficiently.

3 Base storage system

To perform well, a snapshot system must be integrated with its underlying storage system and therefore snapshot techniques must be designed with the storage system architecture in mind. SNAP has been designed to provide snapshots for objects in Thor [8] object storage system. Thor is a highly efficient system that provides competitive performance on standard benchmarks [8, 11]. Thor's efficiency comes from its architecture optimized to perform well for complex object data. This makes it a good basis of investigating high-performance snapshot techniques but it also raises the question of how to generalize the snapshot techniques developed for Thor. We will return to this question in Section 7 after we present SNAP.

3.1 Thor

Thor is an object storage system based on client/server architecture. Servers, also called *object repositories* (ORs), provide persistent storage (called database storage) for objects. Clients, also called *front ends* (FEs), cache copies of the objects and run applications that interact with the system by making calls to methods of cached objects. Method calls occur within the context of transaction. A transaction terminates by a commit or abort. A commit causes all modifications to become persistent, an abort leaves no transaction changes in the persistent state.

Thor uses optimistic concurrency control. FE keeps track of all objects read and modified by current transaction and sends the read and write object sets with modified object states to OR when the application asks to commit the transaction. If no stale objects were read by the transaction, OR commits the transaction.

An object in Thor belongs to a particular OR. The object within OR is uniquely identified by an *object reference* (*Oref*). Thor clusters objects into 8KB pages. Pages are further grouped into 32KB segments to increase the unit of disk transfer and improve disk efficiency. Typically objects are small and there are many of them in a page. Object oref is composed of a 22-bit *PageID* and a 9-bit *oid*. The PageID identifies the containing page and allows the lookup of object location using *page table*. The oid is an index into an offset table stored in the page. The offset table contains the object offsets within the page. This indirection allows to move an object within a page without changing the references to it.

When an object is needed by a client transaction, FE fetches the containing page from OR. Only modified objects are shipped back from FE to OR when the transaction commits. Therefore, in order to propagate a modification to disk, the OR may need to do an *installation read* (iread) [11] to obtain the containing page from disk. Reading containing pages as part of committing a transaction, would degrade transaction performance. Instead, OR uses a deferred iread approach [11] storing committed modifications into a recoverable *Modified Object Buffer* (MOB) [3] and propagating them to disk later, at convenient time. Importantly, this approach enables multiple modifications to the same object as well as to the same page to accumulate and be written to disk by a single write (write absorption), thus reducing disk activities and improving system performance [11, 3]. MOB is recoverable because at commit time modifications are recorded in a stable write-ahead-log [4]. The tail of the stable log is kept in memory sharing the modified objects with the MOB.

Entries are removed from the MOB as the corresponding modifications are installed in their containing pages which are written back to disk. Removal of MOB entries (*cleaning* the MOB) is done by a *cleaner thread* that runs in the background. The thread wakes up and runs when the MOB fills beyond a statically determined *high watermark*. It removes MOB entries until the MOB becomes small enough. The cleaner processes the MOB in transaction log order to facilitate the truncation of the transaction log. For each modified object encountered, it reads the page containing the object from disk (iread) if the page is not cached, installs all modifications in the MOB for objects in that page, and writes the updated page back to disk. When the cleaner finishes a round of cleaning the MOB, it removes log entries for all transactions that have been completely processed from the transaction log.

OR also manages an in-memory page cache used to serve FE fetch requests. Before returning a requested page to FE, OR updates the cache copy, installing all modifications in the MOB for that page so that pages fetched from OR reflect the up-to-date committed state. The page cache uses LRU replacement but discards old dirty pages (it depends on *ireads* to read them back during MOB cleaning) rather than writing them back to disk immediately. Therefore the cleaner thread is the only component of the system that write pages to disk.

4 SNAP design

4.1 Overview

Object storage systems cluster (small) objects on disk pages to exploit the spatial locality in the code accessing the objects, and update objects in-place to maintain the clustering. To provide snapshots in a system that updates objects in-place, an object's pre-state has to be archived before it is updated. Creating snapshots by archiving snapshot state on a per-object rather than per-page basis could reduce archive space. This approach, however, would make back-in-time execution prohibitively expensive for general code accessing a snapshot unless snapshot object clustering is preserved. Clustering is a well-known hard problem. Work in versioned systems [14] explored techniques for maintaining key-based object clustering for indexed access but they do not work for customized application code. To preserve object clustering, SNAP archives entire modified pages in a snapshot. This approach trades extra archive space for better snapshot access performance and the simplicity of system design. Snapshots are stored on a separate archive disk to provide high-performance disk access to both the current storage system and the archive.

A snapshot in SNAP may contain both current database pages and pages stored in the archive. The current page in the database contains the correct state for a page that has not been modified since the snapshot request. The archive contains the correct state for the page that has been modified after the snapshot. Our approach to archiving uses a copy-on-write scheme specialized for snapshots. It creates and archives a snapshot page when a page on the database disk is about to be overwritten the first time after a snapshot is declared. SNAP tracks the highest archived version number for each database page so that each page of each snapshot is archived only once.

To keep track of the pages in snapshots, SNAP manages snapshot page tables that point to the corresponding pages in the archive and in the database. In the next section we describe how code running on a snapshot uses a snapshot page table to lookup objects and transparently redirect object references in a snapshot between database and archive pages.

4.2 Accessing a snapshot

To request a snapshot v, a client application running at FE sends a *snapshot access request* to OR. The OR constructs an archive page table (APT) for version ”v” (APT_v) and ”mounts” it for the FE. APT_v maps each page in snapshot v into its archive address or it indicates the page is in the database. Since snapshots are accessed read-only, APT_v can be shared by all FEs mounting snapshot v. Once APT_v is mounted, OR serves page fetch requests from FE by looking up pages in APT_v and reading them from either archive or database. Figure 1 shows an example of

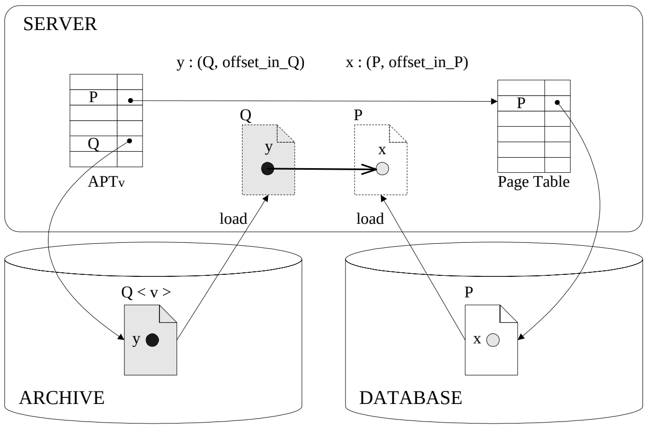

Figure 1. Accessing snapshot v through APT

how unmodified client application code accesses objects in snapshot v that includes both archived and database pages. For simplicity, the example assumes an OR state where all committed modifications have been already propagated to the database and archive disk. In the example, client code requests object y on page Q, SNAP looks up Q in APT_v, loads page $Q < v >$ from the archive and sends the page $Q < v >$ to the client. Later on client code follows a reference from y to x in the client cache, requesting object x in page P from OR. At OR side, SNAP looks up P in APT_v and finds out that page P of snapshot v is still in the database. SNAP reads P from database and sends P to the client.

4.3 Versioned modified object buffer (VMOB)

4.3.1 VMOB data structure

In workloads with overwriting, write-absorption improves performance significantly [3, 11] but poses a problem for snapshot systems because it can lead to loss of snapshot state. To avoid loss of snapshot states, incoming transactions could be blocked after snapshot declaration and the entire MOB could be cleaned. This approach however creates a serious performance problem. It disrupts applications and reduces database write-absorption when snapshots are frequent.

To support frequent snapshots, SNAP uses a simple new approach that stores old snapshot states in a versioned modified object cache called *VMOB*. At the cost of extra memory for VMOB, the approach avoids loss of snapshot states by overwriting without blocking application transactions and without sacrificing write-absorption benefits for database updates.

The VMOB holds the immutable snapshot states in a compact way on a per-object basis, and propagates these snapshot states on a per-page basis into the archive when the database is updated. VMOB consists of a queue of buckets in ascending order of snapshot sequence number. A bucket v in the queue contains the immutable object states created for snapshot v by updates committed after snapshot v was declared. The head bucket is mutable and holds modifications to the current database state. Declaring a new snapshot makes the head VMOB bucket immutable and creates a new empty mutable bucket.

The head VMOB bucket serves as the write-absorption buffer between snapshot declarations, i.e., a new modification to object o overwrites the old object state of o in bucket v after snapshot v is declared but before snapshot $v+1$ is declared. This write-absorption is desirable because it allows to only preserve the states needed for the snapshots.

The modifications in VMOB buckets are propagated into the archive when database is updated. The update process, similar to MOB cleaning in Thor, is, likewise, called VMOB cleaning. The cleaner maintains two invariants. Invariant C1 requires that on-disk copy of a database page that needs to be archived, is written to the archive before it is overwritten in the database. This invariant simplifies archive recovery.

Invariant C2 requires that VMOB buckets are cleaned and removed in snapshot order. Therefore the oldest bucket is always cleaned first. After all modifications in the oldest bucket are archived and propagated into the database, the bucket is removed. At this point, the corresponding log entries for the objects in the removed bucket are also removed from in-memory log tail (as in Thor cleaning). In addition, the snapshot meta-data is updated to reflect the newly archived pages . We will see later that C2 makes it efficient to update persistent snapshot meta-data and to reconstruct APT.

At the time of cleaning, for a given object o, multiple VMOB buckets may contain different versions of object states of o. Furthermore, for a given page P, multiple

VMOB buckets may contain different modifications to P. The cleaner updates a dirty database page once during a cleaning round after all the different snapshot versions of this page corresponding to different buckets are archived. This enforces the invariant C1.

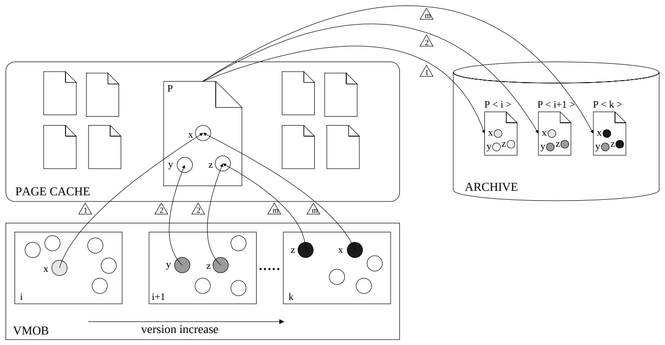

Figure 2. Cleaning a bucket with VMOB

Figure 2 shows the VMOB structure during cleaning. The oldest bucket was created after snapshot version i, the head bucket after snapshot version k. Page P has three modified objects, x, y and z in several buckets. During bucket i cleaning, when P is processed, modifications to P in buckets $i+1$ and k are processed in sequence. First x in bucket i is installed into P and P is archived as $P < i >$ - the P in snapshot i. Then y and z in bucket $i+1$ are installed and archived, and so on. After versions of P have been archived for all needed snapshots from i to k, the current version of P is written to the database.

4.3.2 Performance trade-offs

VMOB affects system performance in several ways. Due to VMOB, snapshot declaration has minimal impact on storage system performance. Declaration is handled as a transaction serialized with other concurrent transactions. The transaction writes a *declare-snapshot* log record to the stable log. At the same time, SNAP simply initializes some in-memory data structures such as a new VMOB bucket.

VMOB affects the number of database updates. In the absence of snapshots, a system with VMOB propagates updates like the base Thor storage system. The incoming modification hit rate in the VMOB (object write-absorption) determines the frequency of VMOB cleaning and, consequently, the number of database page updates. The modification hit rate depends on VMOB size and degree of overwriting (denoted by α) in the workload. When the hit rate is low, VMOB fills fast causing frequent cleaning.

In the presence of snapshots, VMOB holds past snapshot states. In workloads with overwriting, these immutable object states cause VMOB to fill up faster than without snapshots because write-absorption is limited to the mutable head bucket of VMOB. The resulting more frequent cleaning will produce extra database disk updates compared to a system without snapshots.

To avoid the extra database updates, VMOB can be augmented with extra memory to hold snapshot states. The amount of extra memory needed depends on the amount of overwriting in the workload and snapshot frequency. An important question is: how much extra memory is needed? We will show empirically in Section 5 that even for workloads with very high overwriting ($\alpha = 70\%$), an augmented VMOB with double the amount of memory in a system without snapshots (e.g. MOB in Thor) can support high-frequency snapshots without extra database updates. Much less extra memory is needed for workloads with less extreme overwriting. Given the trend of falling memory cost, we consider this cost worthwhile.

When VMOB is cleaned, SNAP uses multiple I/O slave threads to perform page processing in batches parallelly on database and on archive, which can partially hide the cost of sequential I/O on archive disk behind the cost of random I/O on database.

4.4 Archive meta-data

4.4.1 Persistent meta-data: VPT

SNAP meta-data consists of snapshot page tables that enable access to snapshots as explained in Section 4.2. The challenge in managing the meta-data is that with frequent snapshots the total snapshot page table volume grows fast, and, moreover, these page tables are dynamic. They change as VMOB cleaner copies groups of pages from the database into the archive.

The requirements for meta-data management are therefore two-fold: To minimize the performance penalty for the cleaner, SNAP needs to avoid writing large amounts of data. To improve the performance of snapshot access, SNAP needs to support efficient APT mounting.

To manage the meta-data, SNAP keeps a global page table called VPT (version page table). VPT maintains the following mapping for each archived page: $< version, pageid > \longrightarrow archive\_addr$

VPT is both an in-memory and on-disk data structure. When a snapshot v is declared, a new table VPT_v is created in memory. VPT_v keeps the mapping from pageid to archive address of pages archived for snapshot v. It is a partial page table in the sense that not every database page has an entry in VPT_v.

If a page P in snapshot v is in database, P may be modified later on. Before it is modified, however, P is archived and becomes part of a later snapshot w. The following invariant holds for any page P that has no entry in VPT_v:

1. $\exists w, w > v$ and there is an entry for P in VPT_w, or

2. P is in database

VPT is also a persistent data structure on database disk. Persistent VPT is log-structured and append-only. Figure 3 shows the database disk layout including VPT. When the VMOB bucket v is emptied and removed from VMOB as part of the cleaning, VPT_v is appended to the on-disk VPT. The VPTs are appended in the order of snapshot sequence number because VMOB buckets are removed in order (Invariant C2).

Once VPT_v is written out to database, the in-memory VPT_v is evicted. Therefore, the number of in-memory VPTs is very small. These in-memory VPTs correspond to the "live" buckets in VMOB.

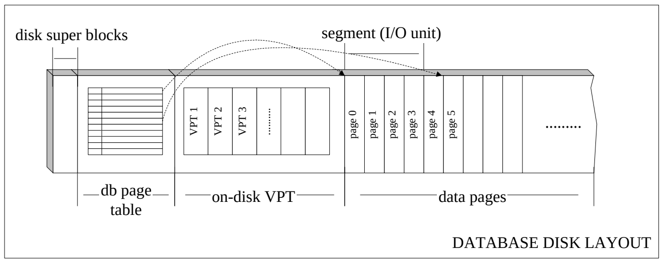

Figure 3. Database layout with VPT

4.4.2 DPT: Dangling page table

We define a dangling page in a snapshot v as a page that has no entry in VPT_v. DPT_v (dangling page table) keeps the following binding list for version v:

$$\{ < P_1, archive\_addr_1 >, \ldots , < P_n, archive\_addr_n > \}$$

,where P_1 ... P_n are dangling pages in snapshot v. Only special versions have corresponding DPTs.

Suppose P is in DPT_v and at time t_1, DPT_v indicates that P is in database in snapshot v. At a later time t_2, P is archived to address $archive\_addr_w$ as part of snapshot w, where $w > v$. In addition to archiving P for snapshot w, SNAP updates DPT_v, binding P to $archive\_addr_w$. In fact, the cleaner updates every DPT for versions smaller than w, if the DPT indicates that P in that snapshot is in the database.

At a later time t_3, when APT_v is built, the archive address for entry P is obtained as follows:

1. If there is an entry for P in VPT_v, then take the $archive\_address$ in VPT_v

2. Otherwise, check DPT_v and take the non-nil binding of P

3. If the binding of P is nil in DPT_v, then P is in database.

Obviously, DPT is a volatile in-memory data structure, reconstructible by a full scan of the on-disk VPT.

4.4.3 VPT checkpoints

DPT can not possibly keep track of dangling pages for every version, growing too large as new snapshots are created. Instead, DPT only keeps track of a limited number of versions called checkpoints.

A version v is a checkpoint if SNAP maintains a DPT_v that keeps track of all dangling pages for snapshot v. DPT keeps track of only a small number (n) of versions.

The checkpoints are spaced at approximately equal intervals in the on-disk VPT and partition the on-disk VPT into n areas.

Suppose the on-disk VPT is 512 MB in size, and the checkpoints are created every 4MB. Then when the log-structured on-disk VPT grows 4MB (roughly) in size, SNAP creates and maintains a DPT_x for version x. (the intervals are approximate because persisting VPT_x may cause on-disk VPT to grow more than 4MB beyond VPT_c, where c is the last checkpoint version before x).

When snapshot v is "mounted" upon client request, SNAP builds APT_v for version v by consulting both VPT and DPT: If v is a checkpoint version, SNAP reads in VPT_v from on-disk VPT and applies all DPT bindings for v. If a page is still dangling in APT_v, it is still in database.

If v is not a checkpoint version, SNAP reads in (Invariant C2 makes it a sequential scan) VPT_v, VPT_{v+1}, VPT_{v+2} ... VPT_x, where either version x is a checkpoint version or VPT_x is the latest persistent VPT on disk. The location of P can be determined as follows:

1. check if P is an entry of VPT_v;

2. If not: check if P is an entry of $VPT_v, VPT_{v+1} ... VPT_x$, in order;

3. If not: if x is a checkpoint version, check if P has a binding in DPT_x;

4. If x is not a checkpoint version: check if P is an entry of any in-memory VPTs that aren't persistent yet, starting from the lowest version to the highest in order;

5. If not: P is in database

In other words, to mount APT_v, SNAP needs to read forward all VPTs on disk until it reaches a checkpoint version x. Together with the in-memory VPTs and DPT_x, APT_v can be constructed. Figure 4 shows the relationship among DPT, VPT and on-disk VPT.

To summarize, checkpointing enables SNAP to maintain a small in-memory DPT and bound the snapshot mount time by the frequency of version checkpointing.

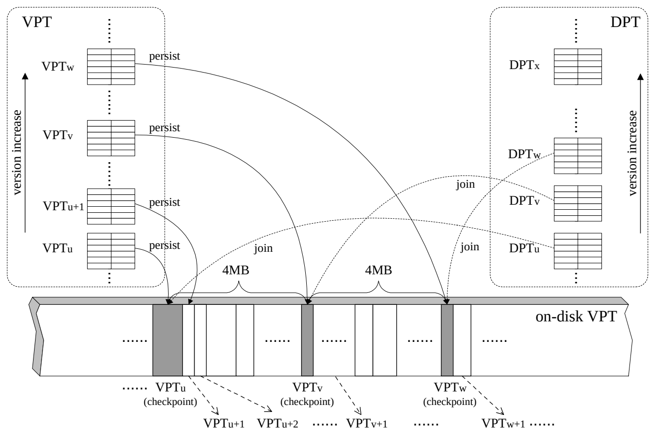

Figure 4. VPT and DPT

4.5 Garbage collection policies

Suppose on average each snapshot version deposits 256MB (32K pages) of data to the archive. Then given 256MB RAM dedicated to checkpoint DPTs, SNAP can support taking a snapshot every minute for 10 days, with 5460MB of on-disk VPT and an archive of 3640GB in size. This shows that with a reasonable amount of in-memory DPT, SNAP can support a long time of history preservation at a high frequency for many of today's applications. However, it may still be unrealistic to keep the entire version history on the hard disk regardless of the snapshot frequency. For some applications, keeping every snapshot is feasible given that all snapshots will be used and discarded shortly after they are taken. The default garbage collection policy in SNAP maintains an active archive window so that disk space of snapshots beyond this window is automatically reclaimed as free space.

For other applications, it is desirable to retain a small subset of the snapshots for long term analysis while discarding the rest fairly soon. SNAP also supports a selective snapshot retaining policy using a simple design that copies out (through sequential I/O) pages and meta-data for selected versions. We do not describe the design and performance evaluation of this subsystem due to lack of space. In general the cleaning cost increases under this policy because of additional I/O on archive disk. Readers may refer to [16] for details.

4.6 Crash recovery

The archive and its meta-data are normally updated as part of cleaning the VMOB. SNAP ensures that snapshot pages are recoverable by writing a record, called *archive-clone* record to stable log every time after a group of pages get archived for a snapshot but before the modifications in VMOB are installed to these pages in the database (Invariant C1). The *archive-clone* log record keeps track of where each page in this group is cloned in the archive for this snapshot.

To ensure easy recovery of the meta-data, SNAP writes a *write-vpt* record into the stable log each time an in-memory VPT is stored on disk. During the recovery, the *write-vpt* records and the *archive-clone* records together allow to easily recover the archive meta-data consisting of persistent VPTs and volatile VPTs that have not been yet propagated to disk before crash.

Lastly, the recovery procedure rebuilds the in-memory DPT data structure by scanning the on-disk VPT. The recovery protocol has been fully designed but has not been implemented yet. For brevity, we omit the description here.

5 Performance evaluation

We implemented SNAP as a snapshot subsystem in Thor object storage system, and evaluated the performance of read-only transactions on snapshots and the impact of snapshots on the storage system. The evaluation considers how performance of SNAP with frequent snapshots compares to the baseline Thor system. Thor is a highly efficient system [8] which makes it a good base for the comparison. SNAP implements the complete design we have described, including the support for recovery during normal operation, allowing us to evaluate system performance in the absence of failures. The failure recovery procedure has not been implemented yet.

5.1 Experiment setup

Our transaction workloads are based on the multiuser medium OO7 benchmark [2]. A transaction includes a read-only traversal (T1), or a read-write traversal. The OO7 read-write traversals T2a and T2b do not allow to generate workloads with varying degree of object overwriting since they always update a fixed set of objects, creating 100% write-absorbtion in the modified object caches in the server. To evaluate SNAP on workloads with varying (low and high) degree of overwriting, we implemented a family of variant traversals T2a' (α) by modifying the traversal T2a. The variant traversals update a randomly selected *AtomicPart* object of a *CompositePart* instead of always modifying the *root* in T2a. We control the amount of overwriting α by adjusting the object update history in a sequence of T2a' traversals. Like T2a, each T2a' traversal modifies 500 objects.

The system includes a single server configuration with a medium multi-user (3 user) OO7 database. The database size is 185MB. Medium OO7 is smaller than a potential SNAP database but easier to manage than a large OO7 database in our long-running experiments. The choice is conservative for our evaluation since faster database I/O (less seek time) makes the archive I/O cost more prominent. The server runs on a Redhat 7.2 workstation with one P4 2Ghz CPU, 512M RAM and 2 Western Digital Caviar 120GB EIDE hard disks. The database resides on a raw hard disk and the archive resides on another raw hard disk. SNAP uses Linux raw devices and direct I/O to bypass file system cache. Clients run on workstations with the same configuration. Unless specified otherwise the server is configured with 50 MB of page cache, and a 2 MB MOB in Thor.

5.2 Snapshot performance

5.2.1 Declaration

A snapshot request in SNAP is a light-weight operation. Without VMOB, a versioning storage system needs to block incoming modifications until the dirty data is flushed (unless no-overwrite storage [17] is used) to avoid the overwriting of previous states by subsequent transactions. A snapshot declaration in SNAP takes on average 0.19 milliseconds to complete without blocking any concurrent transactions.

5.2.2 Read-only transactions on snapshots

Read-only transactions on immutable snapshots have no validation and commit cost. The performance is determined by how many archive and database pages are fetched and the cost of these fetches.

We create an archive by running 1200 T2a' traversals while requesting a snapshot every 1 transaction. The resulting archive of about 5GB contains 1200 snapshots. We run OO7 T1 traversals on snapshot 5, 205, 405, 605 and 805 respectively. The average archive disk page fetch time varies from 6.28ms to 6.76ms. The average database disk fetch time is 3.32ms. The database fetch cost is unrealistically low in our experiments because our database is small and occupies a small disk region. In a larger database we expect fetch cost for general transaction code accessing the database to be closer to random disk access, resulting in moderate snapshot execution degradation due to loss of inter-page locality.

5.2.3 Mounting a snapshot

To access a snapshot v its archive page table (APT) has to be reconstructed. OR reads on-disk VPTs VPT_v, VPT_{v+1}, VPT_{v+2}, ..., VPT_w, where version w is the next checkpoint version ahead of v.

We use the archive built by the previous experiment. The archive was generated with VPT checkpointing set to a default of 4MB i.e., a VPT checkpoint is written to database whenever 4MB of VPT entries have accumulated since the previous checkpoint. Figure 5 shows the experiment results for versions [10,700]. The seesaw pattern in the APT mounting time of a version reflects its distance from the closest checkpoint forward. The mounting time varies between 200ms and 500ms, indicating the cost is independent of the archive size and bounded by the frequency of checkpoints.

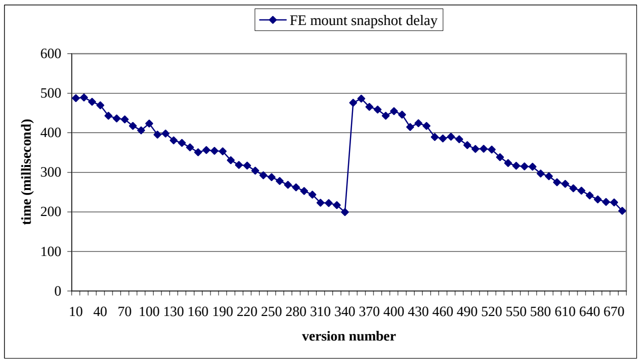

Figure 5. Mounting snapshot overhead

5.3 VMOB memory augmentation

We now consider the question raised in section 4.3: how much extra memory is needed by VMOB so it can avoid the extra database I/O caused by reduced write-absorption in workloads with overwriting. To answer the question we consider workloads with a varying degree of overwriting and a range of snapshot frequencies, and run SNAP and Thor systems side by side under the same workloads, experimentally adding extra memory for VMOB in SNAP until SNAP performs the same number of cleaning database I/O as Thor. The baseline Thor is configured with 2000KB of MOB space.

We designed three workloads: T2a'(α=20%), T2a'(α=40%) and T2a'(α=70%) to represent workloads with low, medium and high object overwriting rates.

SNAP-n denotes a SNAP system creating snapshots at a frequency where a new snapshot is declared every n transaction commits. We study SNAP-n where $n = 1..5, 20, 50$. Since SNAP-1 represents the highest possible snapshot frequency where every committed update to the database is preserved, we refer to SNAP-1 to SNAP-5 as high-frequency snapshot systems, and consider SNAP-20 and SNAP-50 as moderate-frequency snapshot systems.

Figure 6 shows extra object cache memory required for VMOB relative to MOB in Thor, as a function of snapshot frequency. The results indicate that on a workload with low

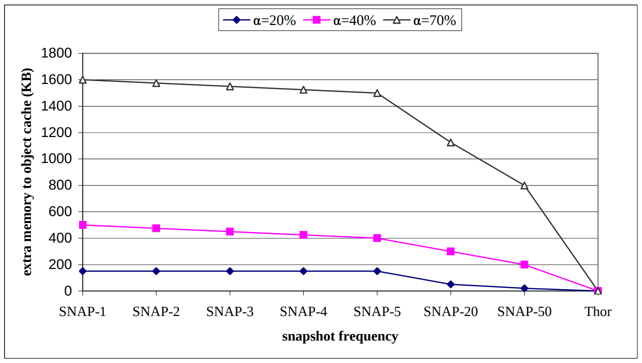

Figure 6. Extra object cache sizes

overwriting rate (α = 20%), regardless of the snapshot frequency, SNAP needs to be given only slightly more extra memory. In SNAP-1, the extra is 150K, only a 7.5% increase relative to Thor. On a workload with medium degree of overwriting (α = 40%), the extra memory size increases as the snapshot frequency increases. 500KB of extra memory is assigned to SNAP-1 – a 25% increase. On a workload with very high overwriting (α = 70%) the extra memory size increase is more significant at high frequencies. At the maximum frequency SNAP-1, the extra memory size is 1600KB – a 80% increase. Note, the amount of extra memory does not vary much from SNAP-1 to SNAP-5 under all workloads. This is because in the very high snapshot frequency range even SNAP-5 already reduces the object-level write absorption in SNAP to near-zero, limiting further increase.

Since object cache is much smaller than the page cache, We think that it is acceptable to configure SNAP with a reasonably larger object cache considering today's hardware costs. Therefore, the rest of the experiments study SNAP and Thor configurations where SNAP is given extra memory for VMOB so that both systems perform the same number of database updates.

5.4 Cleaning cost

Snapshots are built during cleaning. Inevitably, the cleaning process may slow down compared to Thor. However, because during cleaning, pages are cloned to archive disk sequentially in parallel to random database disk ireads and writes, page cloning cost can partially hide behind the database I/O, especially when snapshot frequency is not very high.

Workload (characterized by its object overwriting rate α) on the other hand also has an impact on cleaning performance. For a given a snapshot frequency, higher α causes fewer database page I/Os during cleaning, yet the number of pages to be archived may be high, e.g., SNAP-1 completely eliminates write-absorption across transactions, resulting in the same number of archived pages regardless of α. As a result, higher α in a workload causes SNAP to clone more pages for each dirty database page, making the parallel I/O hiding effect weaker.

Given above analysis, we now consider the cleaning overhead in SNAP. Let t_{page_clean} be the average cleaning time for one dirty database page in Thor (and in SNAP). Since our SNAP configurations perform the same number of database I/O as Thor due to memory augmentation, t_{page_clean} accurately captures the cleaning performance degradation caused by snapshot building.

We run 500 T2a' (α) traversals (where α is 20%, 40% and 70%) in Thor and SNAP-n (where $n = 1..5, 20, 50$). The server had a 50MB page cache across configurations, 2M MOB space in Thor and augmented VMOB (refer to figure 6) in SNAP-n (where $n = 1..5, 20, 50$). We conservatively choose a page cache of 50MB that provides high iread cache hit rate (50%) in Thor (and in SNAP) during cleaning. With a higher iread cache hit rate, there are less database ireads I/O during cleaning, and consequently the hiding effect of parallel I/O is less beneficial to SNAP.

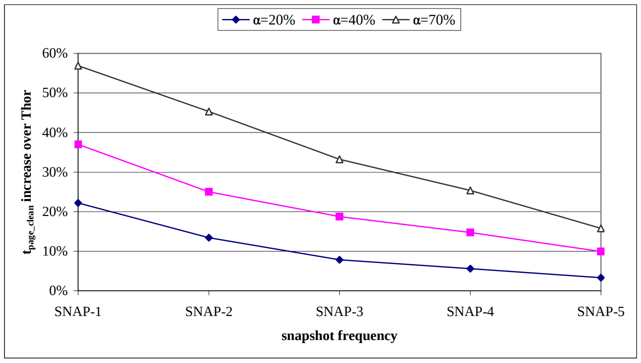

Figure 7. t_{page_clean} increase over Thor under different workloads

Figure 7 shows the increase of t_{page_clean} in SNAP over t_{page_clean} in Thor. The experiment confirms, that as analyzed above, cleaning under higher-α workload introduces more overhead in SNAP. Clearly, the overhead of t_{page_clean} drops as the snapshot frequency decreases from SNAP-1 to SNAP-5, because there are less page cloning to do when snapshot frequency is lower.

Figure 7 does not show data points for SNAP-20 and SNAP-50 because at these snapshot frequencies there was no increase observed in t_{page_clean} in SNAP for all workloads.

With low overwriting (α = 20%), the t_{page_clean} has minimal increase when snapshot frequency is high(SNAP-n, $n = 1..5$). In SNAP-5, t_{page_clean} is only 3.3% higher than t_{page_clean} in Thor. Page-cloning in this configuration is almost completely hidden behind database

I/Os. For each dirty database page written to database, there are on average 3.5 pages cloned in SNAP-5 and 4.5 pages cloned in SNAP-1. With high overwriting ($\alpha = 70\%$), t_{page_clean} is 15.76% higher than Thor in SNAP-5 and 56.9% higher than Thor in SNAP-1. For each dirty database page, there are on average 5.2 pages cloned in SNAP-5 and there are 7 pages cloned in SNAP-1. With moderate overwriting ($alpha = 40\%$), t_{page_clean} is 9.9% higher than Thor in SNAP-5 and 37% higher than Thor in SNAP-1. The overhead of SNAP on cleaning is zero, at moderate frequencies (SNAP-20) and (SNAP-50).

As we show in next section, except for the highest snapshot frequency (SNAP-1) where every update to database is captured, the overhead of SNAP on cleaning translates into only minimal impact on system performance even at a very high snapshot frequency.

The experiment results also show that the meta-data (VPT and DPT) maintenance cost together with the snapshot-related logging time to the stable log is a very small fraction of the overall snapshot-building cost (from 3% to 5%), which is consistent with SNAP's design goal of efficient meta-data. In addition, the client-side performance does not vary across all SNAP configurations as well as Thor.

5.5 Impact on foreground performance

Slow-down of background cleaning at the server side does not necessarily impact the client side performance determined by the foreground transaction commit cost. Transaction commit cost at server side consists of transaction validation cost and the cost of inserting modified objects into object cache. Validation is independent to snapshot building. Inserting modified objects into object cache however, would block a commit attempt if the "draining" of object cache by cleaning falls behind transaction commits that "pour" objects into object cache.

Let R_{pour} be the "pouring rate" – average object cache free-space consumption speed due to incoming transaction commits, which insert modified objects. Let R_{drain} be the "draining rate" – the average growth rate of object cache free space produced by cleaning. We define:

$$\lambda_{dp} = \frac{R_{drain}}{R_{pour}}$$

Clearly, λ_{dp} indicates how well the draining keeps up with the pouring. As long as $\lambda_{dp} \geq 1$, foreground transaction commit performance will not be affected by background cleaning activities. A system is considered to operate within its capacity when $\lambda_{dp} \geq 1$. When $\lambda_{dp} < 1$, transaction commits block on free object cache space and clients experience commit delay. A system is considered overloaded when $\lambda_{dp} < 1$.

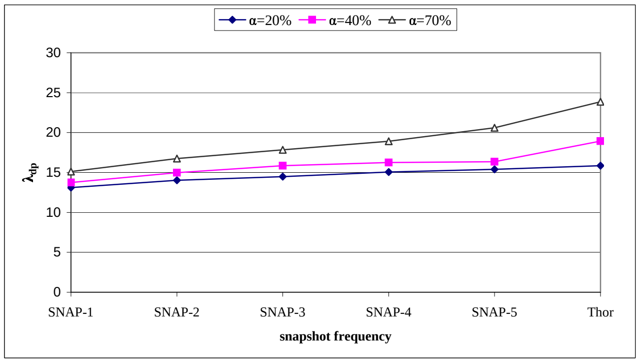

Figure 8. λ_{dp} under different workloads

SNAP decreases λ_{dp} compared to Thor under the same workload because it slows down the cleaning thus reduces R_{drain}. We present in figure 8 different λ_{dp} measured and calculated on output produced by the experiment in section 5.4. The goal is to understand how well SNAP cleaning keeps up under various workloads and at various snapshot frequencies. The three curves represent T2a' workloads with 20%, 40% and 70% object overwriting rates.

First, note that λ_{dp} increases as α increases in the baseline (Thor). This is because in Thor a workload with higher α reduces R_{pour} while keeping R_{drain} unchanged . Second, in all workloads, λ_{dp} always decreases as the frequency of snapshot increases. This is because a higher snapshot frequency entails more snapshot building activities, which in turn reduces R_{drain}. We don't present data points for SNAP-20 and SNAP-50 because we don't observe a decrease of λ_{dp} in these configurations compared to Thor.

In section 5.4 we show that the increase of t_{page_clean} is relatively high when α is very high. Figure 8 tells us on the other hand that when α is very high, λ_{dp} is also high in Thor, which effectively leaves more opportunity for SNAP to keep up. This implies that a workload with high α is not necessarily harder for SNAP to keep up compared to a workload with relatively low α.

Notably, in all configurations and workloads, figure 8 shows a λ_{dp} far greater than 1, which explains why the client-side performance remains constant across all configurations. This implies that in a lightly-loaded database system, SNAP can always support frequent (up to SNAP-1) snapshots without foreground performance degradation.

We have also conducted experiments with higher R_{pour} using multiple concurrent clients against a single server. In our experiments, SNAP's overhead (increase of t_{page_clean}) over Thor decreased compared to a single client setup. With a single client, during cleaning, database I/O primarily includes ireads and writes of dirty pages. With multiple clients, the significant increase in amount of client fetches causes misses in the shared server cache resulting in extra random database disk reads. During cleaning, the random disk reads increase the cost of average database I/O creating a greater opportunity for hiding the sequential archive I/O done in parallel on archive disk. Due to the lack of space, we do not present the experiment results for multiple clients.

6 Related work

Many storage systems provide backup snapshots for failure recovery rather than back-in-time execution and are not concerned with low-cost frequent snapshots. Chervenak [7] provides a survey of backup system techniques for file systems. Lindsey [1] introduces transactionally consistent snapshots in a relational database. Fuzzy dump [4] is a technique to build a transactionally consistent snapshot without blocking concurrent transaction execution. Mohan et al. [10] describe an incremental fuzzy dump using an approach similar to SNAP. Copy-on-write archives pre-images of modified pages, placing latches on dirty pages to block transactions from overwriting data that needs to be archived. Versions are not supported as the latest archive copy of same data page overwrites a previous copy. Copy-on-write is used in persistent object systems [6] to provide consistent checkpoints for recovery but not versions.

Multi-version concurrency control systems [9] manage consistent snapshot objects on disk pages. The goal is to enable read-only queries to run against snapshots instead of current state to avoid contention. These systems do not allow the application to choose the snapshot, and, typically manage only few versions that are not retained for longer term storage.

Postgres [19] is the first transactional system to use a redo log to materialize old and current database versions on demand. The version creation scheme based on no-overwrite update approach is low-cost but the approach introduces a high performance penalty for access to the current version [20].

Temporal database systems [18, 12] provide access to versioned snapshots by extending objects (tuples) with a time attribute. Salzberg et al. [14] describe a general framework for access methods in versioned storage systems. Like in SNAP, efficient access to versions is based on clustering of versioned data (and index meta-data) on disk pages. The difference is they focus on efficient version access for key-range queries rather than general code and evaluate it using theoretical analysis.

There has been a substantial amount of work on archiving snapshots in file systems that although has not been designed to support transactions, uses techniques relevant to our work in SNAP. The Write Anywhere File Layout (WAFL) [5] provides a file system snapshot operation using a block level copy-on-write to archive pre-images of blocks modified after the snapshot operation is requested. The cache is flushed to the archive asynchronously and only the root inode block needs to be written synchronously when snapshot starts. Flushing a large cache is costly because of loss of write absorption. Moreover, where SNAP relies on VMOB to prevent overwriting snapshot states, the WAFL system blocks application requests that would modify these blocks.

Elephant [15] is a versioned file system. Timestamps are used to provide a consistent snapshot of a file created by the updates between open and close operations on the file. Elephant uses a copy-on-write approach to store file versions and supports both on-demand versions (landmarks) and automatic versions to support undo.

Log-structured file systems [13] manage meta-data in a way similar to SNAP but do not support versions. CVFS [17] is a versioned file system built over a log-structured file system [13] that investigates meta-data management issues. The size of versioned meta-data is reduced by incremental meta-data logging and checkpointing. The difference is in what is optimized. CVFS uses multi-version tree technique to keep compact meta-data for directories but is not concerned with efficient back-in-time execution. As a result, back-in-time execution that needs to access state in multiple files (e.g. to run a make program) could be very expensive.

7 Conclusion

This paper describes SNAP, a novel consistent snapshot service for object storage systems. Our goal is to provide a snapshot service that is so efficient that it opens up a whole new range of applications based on back-in-time execution.

SNAP supports back-in-time execution that is more expressive than in other systems. Application can choose the snapshots they access and they can run on snapshots general code.

A key innovation in SNAP is that unlike in earlier systems, SNAP provides snapshots that are transactionally consistent, yet non-disruptive. They do not prevent applications from accessing the storage system even when snapshots are frequent, and they do not cause unnecessary disk update. In this way, SNAP avoids performance problems common to current snapshot systems.

SNAP performs well because of several reasons. Its design trades extra system resources for better performance; it uses novel data structures; and it is tightly integrated with a unique storage system architecture – Thor.

Earlier in Section 3 we raised the question of how applicable SNAP techniques and design decisions are to other systems. We believe that the techniques and design decisions in SNAP are general and apply to other systems. The decision to archive pages rather than objects and to access versions through versioned page tables is based on the requirement to support general code for back-in-time execution with reasonable performance, the technology trend toward inexpensive and plentiful disk storage, and the desire for archive design simplicity. These are Thor-independent reasons that would apply to other systems as well. These systems could benefit from the incremental page table maintenance techniques VPT and DPT.

The VMOB technique is Thor specific. However, the approach of retaining old snapshot states in versioned cache to support non-disruptive snapshots at the cost of extra memory seems to be more general and worthwhile given the falling memory costs. Future work includes extending the technique to locking systems and STEAL buffer management policy.

We implemented SNAP as a subsystem in Thor [8] and evaluated performance considering two important performance parameters, snapshot frequency and amount of overwriting in the workload. The results, based on extended OO7 benchmark [2], indicate that SNAP supports back-in-time execution efficiently, and, except with workloads of extreme overwriting, supports frequent snapshots with minimal impact on the performance of the storage system.

This paper makes the following contributions. It describes SNAP, the first efficient non-disruptive consistent snapshot system for object storage systems and describes a specific implementation of SNAP. It presents new snapshot techniques that are general and can be applied to other systems. It presents the performance evaluation of the new techniques and presents experimental evidence that supports our claims.

8 Acknowledgments

We thank Paaras Kumar for contributions to the initial design of SNAP, Jim Gray for useful comments, and Maurice Herlihy for help in improving the presentation.

References

[1] M. Adiba and B. Lindsay. Database snapshots. In *Proceedings of the 6th VLDB Conference*, pages 86–91, 1980.

[2] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 Benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington D.C., May 1993.

[3] S. Ghemawat. *The Modifi ed Object Buffer: A Storage Management Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, September 1995.

[4] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

[5] D. Hitz, J. Lau, and M. Malcom. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, San Francisco, CA, USA, January 1994.

[6] B. Koch, T. Schunke, A. Dearle, F. Vaughan, C. Marlin, R. Fazakerley, and C. Barter. Cache coherency and storage management in a persistent object system. In *Proceedings of the Fourth International Workshop on Persistent Object Systems Design, Implementation and Use*, pages 255–266, Martha's Vineyard, MA, Sept. 1990.

[7] Z. Kurmas and A. Chervenak. Evaluating Backup Algorithms. In *Proceedings of the IEEE Symposium on Mass Storage System*, 1998.

[8] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, Lisbon, Portugal, June 1999.

[9] P. M.Bober and M. J. Carey. On Mixing Queries and Transactions via Multiversion Locking. 1992.

[10] C. Mohan and I. Narang. An Effi cient and Flexible Method for Archiving a Data Base. In *Proceedings of the ACM SIGMOD Conference*, Washington, D.C., USA, May 1993.

[11] J. O'Toole and L. Shrira. Opportunistic Log: Effi cient Installation Reads in a Reliable Storage Server. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, USA, November 1994.

[12] G. Ozsoyoglu and R. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knoweldge and Data Engineering*, 7(4):513–532, August 1995.

[13] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log structured fi le system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, Pacifi c Grove, Ca, Oct. 1991. ACM.

[14] B. Salzberg, L. Jiang, D. Lomet, M. Barrena, J. Shan, and E. Kanoulas. A Framework for Access Methods for Versioned Data. *EDBT*, 2004.

[15] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofi r. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Charleston, SC, USA, December 1999.

[16] L. Shrira and H. Xu. SNAP: A Technical Report. *http://www.cs.brandeis.edu/˜hxu/tr-snap.pdf*, Mar. 2004.

[17] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata Effi ciency in Versioning File Systems. In *Proceedings of the 2nd Conference on File and Storage Technologies (FAST)*, San Francisco, CA, USA, March 2003.

[18] A. Steiner and M. C. Norrie. Implementing Temporal Databases in Object-Oriented Systems. In *In Proceedings of the 9th Int'l Conference on Advanced Information Systems Engineering*, Barcelona, Spain, June 1997.

[19] M. Stonebraker. The Design of the POSTGRES Storage System. In *Proceedings of the 13th International Conference on Very-Large Data Bases*, Brighton, England, UK, September 1987.

[20] M. Stonebraker and J. Hellerstein, editors. *Readings in Datbase Systems*, chapter 3. Morgan Kaufmann Publishers, Inc., 3rd edition, 1998.