Thresher: An Efficient Storage Manager for Copy-on-write Snapshots

Liuba Shrira*
Department of Computer Science
Brandeis University
Waltham, MA 02454

Hao Xu
Department of Computer Science
Brandeis University
Waltham, MA 02454

Abstract

A new generation of storage systems exploit decreasing storage costs to allow applications to take snapshots of past states and retain them for long durations. Over time, current snapshot techniques can produce large volumes of snapshots. Indiscriminately keeping all snapshots accessible is impractical, even if raw disk storage is cheap, because administering such large-volume storage is expensive over a long duration. Moreover, not all snapshots are equally valuable. Thresher is a new snapshot storage management system, based on novel copy-on-write snapshot techniques, that is the first to provide applications the ability to discriminate among snapshots efficiently. Valuable snapshots can remain accessible or stored with faster access while less valuable snapshots are discarded or moved off-line. Measurements of the Thresher prototype indicate that the new techniques are efficient and scalable, imposing minimal (4%) performance penalty on expected common workloads.

1 Introduction

A new generation of storage systems exploit decreasing storage costs and efficient versioning techniques to allow applications to take snapshots of past states and retain them for long durations. Snapshot analysis is becoming increasingly important. For example, an ICU monitoring system may analyze the information on patients' past response to treatment.

Over time, current snapshot techniques can produce large volumes of snapshots. Indiscriminately keeping all snapshots accessible is impractical, even if raw disk storage is cheap, because administering such large-volume storage is expensive over a long duration. Moreover, not all snapshots are equally valuable. Some are of value for a long time, some for a short time. Some may require faster access. For example, a patient monitoring system might retain readings showing an abnormal behavior. Recent snapshots may require faster access than older snapshots.

Current snapshot systems do not provide applications with the ability to *discriminate* efficiently among snapshots, so that valuable snapshots remain accessible while less valuable snapshots are discarded or moved off-line. The problem is that incremental copy-on-write, the basic technique that makes snapshot creation efficient, *entangles* on the disk successive snapshots. Separating entangled snapshots creates disk fragmentation that reduces snapshot system performance over time.

This paper describes Thresher, a new snapshot storage management system, based on a novel snapshot technique, that is the first to provide applications the ability to discriminate among snapshots efficiently. An application provides a discrimination policy that ranks snapshots. The policy can be specified when snapshots are taken, or later, after snapshots have been created. Thresher efficiently disentangles differently ranked snapshots, allowing valuable snapshots to be stored with faster access or to remain accessible for longer, and allowing less-valuable snapshots to be discarded, all without creating disk fragmentation.

Thresher is based on two key innovations. First, a novel technique called *ranked segregation* efficiently separates on disk the states of differently-ranked copy-on-write snapshots, enabling no-copy reclamation without fragmentation. Second, while most snapshot systems rely on a no-overwrite update approach, Thresher relies on a novel update-in-place technique that provides an efficient way to transform snapshot representation as snapshots are created.

The ranked segregation technique can be efficiently composed with different snapshot representations to lower the storage management costs for several useful discrimination policies. When applications need to defer

*This work was supported in part by NSF grant ITR-0428107 and Microsoft.

snapshot discrimination, for example until after examining one or more subsequent snapshots to identify abnormalities, Thresher segregates the normal and abnormal snapshots efficiently by composing ranked segregation with a compact diff-based representation to reduce the cost of copying. For applications that need faster access to recent snapshots, Thresher composes ranked segregation with a dual snapshot representation that is less compact but provides faster access.

A snapshot storage manager, like a garbage collector, must be designed with a concrete system in mind, and must perform well for different application workloads. To explore how the performance of our new techniques depends on the storage system workload, we prototyped Thresher in an experimental snapshot system [12] that allows flexible control of workload parameters. We identified two such parameters, update density and overwriting, as the key parameters that determine the performance of a snapshot storage manager. Measurements of the Thresher prototype indicate that our new techniques are efficient and scalable, imposing minimal (4%) performance penalty on common expected workloads.

2 Specification and context

In this section we specify Thresher, the snapshot storage management system that allows applications to discriminate among snapshots. We describe Thresher in the context of a concrete system but we believe our techniques are more general. Section 3 points out the snapshot system dependent features of Thresher.

Thresher has been designed for a snapshot system called SNAP [12]. SNAP assumes that applications are structured as sequences of *transactions* accessing a storage system. It supports *Back-in-time execution* (or, BITE), a capability of a storage system where applications running general code can run against read-only snapshots in addition to the current state. The snapshots reflect transactionally consistent historical states. An application can choose which snapshots it wants to access so that snapshots can reflect states meaningful to the application. Applications can take snapshots at unlimited "resolution" e.g. after each transaction, without disrupting access to the current state.

Thresher allows applications to discriminate among snapshots by incorporating a snapshot discrimination policy into the following three snapshot operations: a request to take a snapshot (*snapshot request, or declaration*) that provides a *discrimination policy*, or indicates lazy discrimination, a request to access a snapshot (*snapshot access*), and a request to specify a discrimination policy for a snapshot (*discrimination request*).

The operations have the following semantics. 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, e.g. to specify a discrimination policy for a snapshot. 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 wants 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 discrimination policy ranks snapshots. A rank is simply a numeric score assigned to a snapshot. Thresher interprets the ranking to determine the relative lifetimes of snapshots and the relative snapshot access latency.

A snapshot storage management system needs to be efficient and not unduly slow-down the snapshot system.

3 The snapshot system

Thresher is implemented in SNAP [12], the snapshot system that provides snapshots for the Thor [7] object storage system. This section reviews the baseline storage and snapshot systems, using Figure 3 to trace their execution within Thresher.

Our general approach to snapshot discrimination is applicable to snapshot systems that separate snapshots from the current storage system state. Such so-called *split* snapshot systems [16] rely on update-in-place storage and create snapshots by copying out the past states, unlike snapshot systems that rely on no-overwrite storage and do not separate snapshot and current states [13]. Split snapshots are attractive in long-lived systems because they allow creation of high-frequency snapshots without disrupting access to the current state while preserving the on-disk object clustering for the current state [12]. Our approach takes advantage of the separation between snapshot and current states to provide efficient snapshot discrimination. We create a specialized snapshot representation tailored to the discrimination policy while copying out the past states.

3.1 The storage system

Thor has a client/server architecture. Servers provide persistent storage (called database storage) for objects. Clients 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 a the context of transaction. A transaction commit causes all

modifications to become persistent, while an abort leaves no transaction changes in the persistent state. The system uses optimistic concurrency control [1]. A client sends its read and write object sets with modified object states to the server when the application asks to commit the transaction. If no conflicts were detected, the server commits the transaction.

An object belongs to a particular server. The object within a server is uniquely identified by an *object reference* (*Oref*). Objects are clustered into 8KB pages. Typically objects are small and there are many of them in a page. An object Oref is composed of a *PageID* and a *oid*. The PageID identifies the containing page and allows the lookup of an object location using a *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 us to move an object within a page without changing the references to it.

When an object is needed by a transaction, the client fetches the containing page from the server. Only modified objects are shipped back to the server when the transaction commits. Thor provides transaction durability using the ARIES no-force no-steal redo log protocol [5]. Since only modified objects are shipped back at commit time, the server may need to do an *installation read* (iread) [8] to obtain the containing page from disk. An in-memory, recoverable cache called the *modified object buffer*(*MOB*) stores the committed modifications allowing to defer ireads and increase write absorption [4, 8]. The modifications are propagated to the disk by a background *cleaner* thread that *cleans* the MOB. 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, writes the updated page back to disk, and removes the objects from the MOB.

The server also manages an in-memory page cache used to serve client fetch requests. Before returning a requested page to the client, the server updates the cache copy, installing all modifications in the MOB for that page so that the fetched page reflects 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 writes pages to disk.

3.2 Snapshots

SNAP creates snapshots by copying out the past storage system states onto a separate snapshot archive disk. A snapshot provides the same abstraction as the storage system, consisting of *snapshot pages* and a *snapshot page table*. This allows unmodified application code running in the storage system to run as BITE over a snapshot.

SNAP copies snapshot pages and snapshot page table mappings into the archive during cleaning. It uses an incremental copy-on-write technique specialized for split snapshots: a snapshot page is constructed and copied into the archive when a page on the database disk is about to be overwritten the first time after a snapshot is declared. Archiving a page creates a snapshot page table mapping for the archived page.

Consider the pages of snapshot v and page table mappings over the transaction history starting with the snapshot v declaration. At the declaration point, all snapshot v pages are in the database and all the snapshot v page table mappings point to the database. Later, after several update transactions have committed modifications, some of the snapshot v pages may have been copied into the archive, while the rest are still in the database. If a page P has not been modified since v was declared, snapshot page P is in the database. If P has been modified since v was declared, the snapshot v version of P is in the archive. The snapshot v page table mappings track this information, i.e. the archive or database address of each page in snapshot v.

Snapshot access. We now describe how BITE of unmodified application code running on a snapshot uses a snapshot page table to look up objects and transparently redirect object references within a snapshot between database and archive pages.

To request a snapshot v, a client application sends a *snapshot access request* to the server. The server constructs an archive page table (APT) for version v (APT_v) and "mounts" it for the client. APT_v maps each page in snapshot v into its archive address or indicates the page is in the database. Once APT_v is mounted, the server receiving a page fetch requests from the client looks up pages in APT_v and reads them from either archive or database. Since snapshots are accessed read-only, APT_v can be shared by all clients mounting snapshot v.

Figure 1 shows an example of how unmodified client application code accesses objects in snapshot v that includes both archived and database pages. For simplicity, the example assumes a server state where all committed modifications have been already propagated to the database and the archive disk. In the example, client code requests object y on page Q, the server looks up Q in APT_v, loads page Q_v from the archive and sends it 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 the server. The server looks up P in APT_v

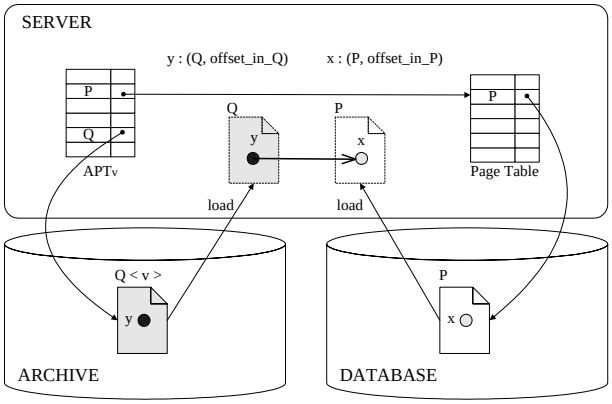

Figure 1: BITE: page-based representation

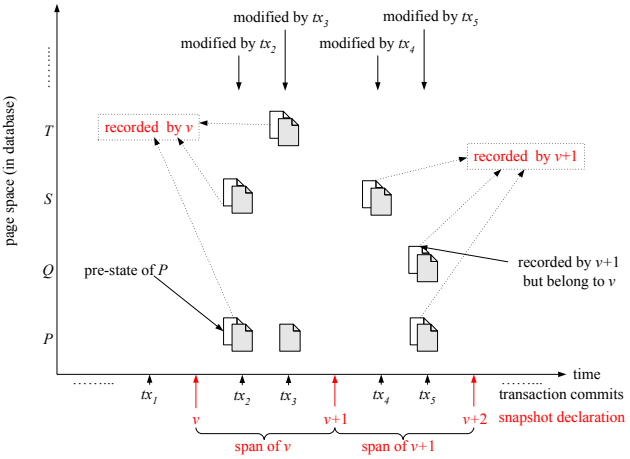

Figure 2: Split copy-on-write

and finds out that the page P for snapshot v is still in the database. The server reads P from the database and sends it to the client.

In SNAP, the archive representation for a snapshot page includes the complete storage system page. This representation is refered to as *page-based*. The following sections describe different snapshot page representations, specialized to various discrimination policies. For example, a snapshot page can have a more compact representation based on modified object diffs, or it can have two different representations. Such variations in snapshot representation are transparent to the application code running BITE, since the archive read operation reconstructs the snapshot page into storage system representation before sending it to the client.

Snapshot creation. The notions of a *snapshot span* and pages *recorded* by a snapshot capture the incremental copy-on-write manner by which SNAP archives snapshot pages and snapshot page tables. Snapshot declarations partition transaction history into *spans*. The span of a snapshot v starts with its declaration and ends with the declaration of the next snapshot ($v+1$). Consider the first modification of a page P in a span of a snapshot v. The pre-state of P belongs to snapshot v and has to be eventually copied into the archive. We say snapshot v *records* its version of P. In Figure 2, snapshot v records pages P and S (retaining the pre-states modified by transaction tx_2) and the page T (retaining the pre-state modified by transaction tx_3). Note that there is no need to retain the pre-state of page P modified by transaction tx_3 since it is not the first modification of P in the span.

If v does not *record* a version of page P, but P is modified after v is declared, in a span of a later snapshot, the later snapshot records v's version of P. In above example, v's version of page Q is recorded by a later snapshot $v+1$ who also records its own version of P.

Snapshot pages are constructed and copied into the archive during cleaning when the pre-states of modified pages about to be overwritten in the database are available in memory. Since the cleaner runs asynchronously with the snapshot declaration, the snapshot system needs to prevent snapshot states from being overwritten by the on-going transactions. For example, if several snapshots are declared between two successive cleaning rounds, and a page P gets modified after each snapshot declaration, the snapshot system has to retain a different version of P for each snapshot.

SNAP prevents snapshot state overwriting, without blocking the on-going transactions. It retains the pre-states needed for snapshot creation in an in-memory data structure called *versioned modified object buffer* (VMOB). VMOB contains a queue of buckets, one for each snapshot. Bucket v holds modifications committed in v's span. As transactions commit modifications, modified objects are added to the bucket of the latest snapshot (Step 1, Figure 3). The declaration of a new snapshot creates a new mutable bucket, and makes the preceding snapshot bucket immutable, preventing the overwriting of the needed snapshot states.

A cleaner updates the database by cleaning the modifications in the VMOB, and in the process of cleaning, constructs the snapshot pages for archiving. Steps 2-5 in Figure 3 trace this process. To clean a page P, the cleaner first obtains a database copy of P. The cleaner then uses P and the modifications in the buckets to create all the needed snapshot versions of P before updating P in the database. Let v be the first bucket containing modifications to P, i.e. snapshot v records its version of P. The cleaner constructs the version of P recorded by v simply by using the database copy of P. The cleaner then updates P by applying modifications in bucket v, removes the modifications from the bucket v, and pro-

ceeds to the following bucket. The updated P will be the version of P recorded by the snapshot that has the next modification to P in its bucket. This process is repeated for all pages with modifications in VMOB, constructing the recorded snapshot pages for the snapshots corresponding to the immutable VMOB buckets.

The cleaner writes the recorded pages into the archive sequentially in snapshot order, thus creating *incremental* snapshots. The mappings for the archived snapshot pages are collected in versioned *incremental* snapshot page tables. VPT_v (versioned page table for snapshot v) is a data structure containing the mappings (from page id to archive address) for the pages recorded by snapshot v. As pages recorded by v are archived, mappings are inserted into VPT_v. After all pages recorded by v have been archived, VPT_v is archived as well.

The cleaner writes the VPTs sequentially, in snapshot order, into a separate archive data structure. This way, a forward sequential scan through the archived incremental page tables from VPT_v and onward finds the mappings for all the archived pages that belong to snapshot v. Namely, the mapping for $v's$ version of page P is found either in VPT_v, or, if not there, in the VPT of the first subsequently declared snapshot that records P. SNAP efficiently bounds the length of the scan [12]. For brevity, we do not review the bounded scan protocol here.

To construct a snapshot page table for snapshot v for BITE, SNAP needs to identify the snapshot v pages that are in the current database. HAV is an auxiliary data structure that tracks the highest archived version for each page. If $\text{HAV}(P) < v$, the snapshot v page P is in the database.

4 Snapshot discrimination

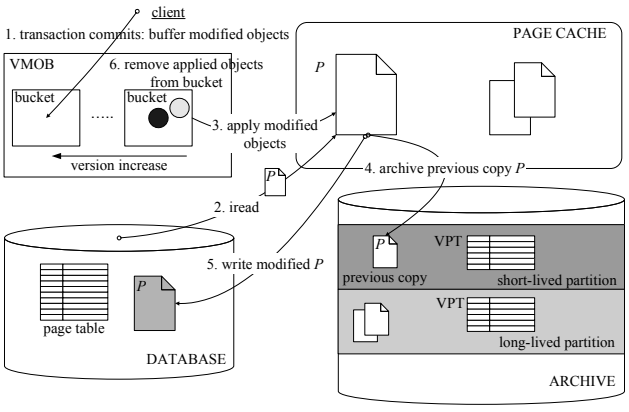

Figure 3: Archiving split snapshots

A snapshot discrimination policy may specify that older snapshots outlive more recently declared snapshots. Since snapshots are archived incrementally, managing storage for snapshots according to such a discrimination policy can be costly. Pages that belong to a longer-lived snapshot may be recorded by a later short-lived snapshot thus entangling short-lived and long-lived pages. When different lifetime pages are entangled, discarding shorter-lived pages creates archive storage fragmentation. For example, consider two consecutive snapshots v and $v+1$ in Figure 2, with v recording pages versions P_v, and S_v, and $v+1$ recording pages P_{v+1}, Q_{v+1} and S_{v+1}. The page Q_{v+1} recorded by $v+1$ belongs to both snapshots v and $v+1$. If the discrimination policy specifies that v is long-lived but $v+1$ is transient, reclaiming $v+1$ before v creates disk fragmentation. This is because we need to reclaim P_{v+1} and S_{v+1} but not Q_{v+1} since Q_{v+1} is needed by the long-lived v.

In a long-lived system, disk fragmentation degrades archive performance causing non-sequential archive disk writes. The alternative approach that copies out the pages of the long-lived snapshots, incurs the high cost of random disk reads. But to remain non-disruptive, the snapshot system needs to keep the archiving costs low, i.e. limit the amount of archiving I/O and rely on low-cost sequential archive writes. The challenge is to support snapshot discrimination efficiently.

Our approach exploits the copying of past states in a split snapshot system. When the application provides a snapshot discrimination policy that determines the lifetimes of snapshots, we *segregate* the long-lived and the short-lived snapshot pages and copy different lifetime pages into different archive areas. When no long-lived pages are stored in short-lived areas, reclamation creates no fragmentation. In the example above, if the archive system knows at snapshot $v+1$ creation time that it is shorter-lived than v, it can store the long-lived snapshot pages P_v, S_v and Q_{v+1} in a long-lived archive area, and the transient P_{v+1}, S_{v+1} pages in a short-lived area, so that the shorter-lived pages can be reclaimed without fragmentation.

Our approach therefore, combines a discrimination policy and a discrimination mechanism. Below we characterize the discrimination policies supported in Thresher. The subsequent sections describe the discrimination mechanisms for different policies.

Discrimination policies. A snapshot discrimination policy conveys to the snapshot storage management system the importance of snapshots so that more important snapshots can have longer lifetimes, or can be stored with faster access. Thresher supports a class of flexible discrimination policies described below using an example. An application specifies a discrimination policy by providing a relative snapshot ranking. Higher-ranked snapshots are deemed more important. By default, every

snapshot is created with a lowest rank. An application can "bump up" the importance of a snapshot by assigning it a higher rank. In a hospital ICU patient database, a policy may assign the lowest rank to snapshots corresponding to minute by minute vital signs monitor readings, a higher rank to the monitor readings that correspond to hourly nurses' checkups, yet a higher rank to the readings viewed in doctors' rounds. Within a given rank level, more recent snapshots are considered more important. The discrimination policy assigns longer lifetimes to more important snapshots, defining a 3-level sliding window hierarchy of snapshot lifetimes.

The above policy is a representative of a general class of discrimination policies we call *rank-tree*. More precisely, a k-level rank-tree policy has the following properties, assuming rank levels are given integer values 1 through k:

- RT1: A snapshot ranked as level i, $i > 1$, corresponds to a snapshot at each lower rank level from 1 to $(i-1)$.
- RT2: Ranking a snapshot at a higher rank level increases its lifetime.
- RT2: Within a rank level, more recent snapshots outlive older snapshots.

Figure 4 depicts a 3-level rank-tree policy for the hospital example, where snapshot number 1, ranked at level 3, corresponds to a monitor reading that was sent for inspection to both the nurse and the doctor, but snapshot number 4 was only sent to the nurse.

An application can specify a rank-tree policy *eagerly* by providing a snapshot rank at snapshot declaration time, or *lazily*, by providing the rank after declaring a snapshot. An application can also ask to store recent snapshots with faster access. In the hospital example above, the importance and the relative lifetimes of the snapshots associated with routine procedure are likely to be known in advance, so the hospital application can specify a snapshot discrimination policy eagerly.

4.1 Eager ranked segregation

The eager *ranked segregation* protocol provides efficient discrimination for eager rank-tree policies. The protocol assigns a separate archive region to hold the snapshot pages ($volumes^i$) and snapshot page tables (VPT^i) for snapshots at level i. During snapshot creation, the protocol segregates the different lifetime pages and copies them into the corresponding regions. This way, each region contains pages and page tables with the same lifetime and temporal reclamation of snapshots (satisfying policy property RT2) within a region does not create disk fragmentation. Figure 3 shows a segregated archive.

At each rank level i, snapshots ranked at level i are archived in the same incremental manner as in SNAP and at the same low sequential cost. The cost is low because by using sufficiently large write buffers (one for each volume), archiving to multiple volumes can be as efficient as strictly sequential archiving into one volume. Since we expect the rank-tree to be quite shallow the total amount of memory allocated to write buffers is small.

The eager ranked segregation works as follows. The declaration of a snapshot v with a rank specified at level $k(k \geq 1)$, creates a separate incremental snapshot page table, VPT^i_v for every rank level $i(i \leq k)$. The incremental page table VPT^i_v collects the mappings for the pages recorded by snapshot v at level i. Since the incremental tables in VPT^i map the pages recorded by *all* the snapshots at level i, the basic snapshot page table reconstruction protocol based on a forward scan through VPT^i (Section 3.2) can be used in region i to reconstruct snapshot tables for level i snapshots.

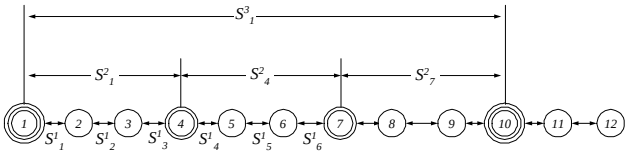

Figure 4: Example rank-tree policy

The recorded pages contain the pre-state before the first page modification in the snapshot span. Since the span for snapshot v at level i (denoted S^i_v) includes the spans of all the lower level snapshots declared during S^i_v, pages recorded by a level i snapshot v are also recorded by some of these lower-ranked snapshots. In Figure 4, the span of snapshot 4 ranked at level 2 includes the spans of snapshots (4), 5 and 6 at level 1. Therefore, a page recorded by the snapshot 4 at level 2 is also recorded by one of the snapshots (4), 5, or 6 at level 1.

A page P recorded by snapshots at multiple levels is archived in the volume of the highest-ranked snapshot that records P. We say that the highest recorder *captures* P. Segregating archived pages this way guarantees that a volume of the shorter-lived snapshots contains no longer-lived pages and therefore temporal reclamation within a volume creates no fragmentation.

The mappings in a snapshot page table VPT^i_v in area i point to the pages recorded by snapshot v in whatever area these pages are archived. Snapshot reclamation needs to insure that the snapshot page table mappings are safe, that is, they do not point to reclaimed pages. The segregation protocol guarantees the safety of the snapshot page table mappings by enforcing the following invariant I that constrains the intra-level and inter-level reclamation order for snapshot pages and page tables:

1. VPT_v^i and the pages recorded by snapshot v that are *captured* in $volume^i$ are reclaimed together, in temporal snapshot order.

2. Pages recorded by snapshot v at level $k(k > 1)$, *captured* in $volume^k$, are reclaimed *after* the pages recorded by all level $i(i < k)$ snapshots declared in the span of snapshot v at level k.

I(1) insures that in each given rank-tree level, the snapshot page table mappings are safe when they point to pages captured in volumes within the same level. I(2) insures that the snapshot page table mappings are safe when they point to pages captured in volumes above their level. Note that the rank-tree policy property RT2 only requires that "bumping up" a lower-ranked snapshot v to level k extends its lifetime but it does not constrain the lifetimes of the lower-level snapshots declared in the span of v at level k. I(2) insures the safety of the snapshot table mappings for these later lower-level snapshots.

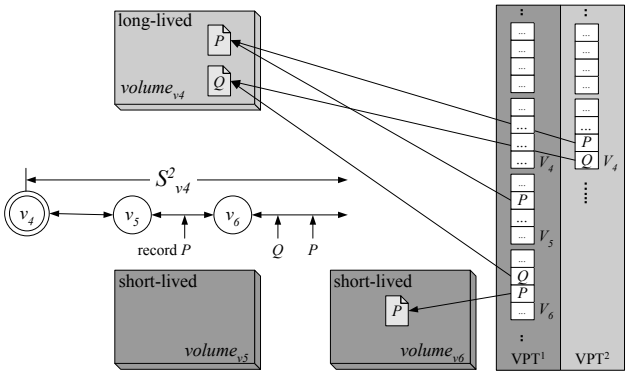

Figure 5: Eager ranked segregation

Figure 5 depicts the eager segregation protocol for a two-level rank-tree policy shown in the figure. Snapshot v_4, specified at level 2, has a snapshot page table at both level 1 and level 2. The archived page P modified within the span of snapshot v_5, is recorded by snapshot v_5, and also by the level 2 snapshot v_4. This version of P is archived in the volume of the highest recording snapshot (denoted $volume_{v_4}$). The snapshot page tables of both recording snapshots $VPT^1_{v_5}$ and $VPT^2_{v_4}$ contain this mapping for P. Similarly, the pre-state of page Q modified within the span of v_6 is also captured in $volume_{v_4}$. P is modified again within the span of snapshot v_6. This later version of P is not recorded by snapshot v_4 at level 2 since v_4 has already recorded its version of P. This later version of P is archived in $volume_{v_6}$ and its mapping is inserted into $\text{VPT}^1_{v_6}$. Invariant $I(1)$ guarantees that in $\text{VPT}^1_{v_6}$ mappings for page P in $volume_{v_6}$ is safe. Invariant $I(2)$ guarantees that in $\text{VPT}^1_{v_6}$ the mapping for page Q in volume v_4 is safe.

4.2 Lazy segregation

Some applications may need to defer snapshot ranking to after the snapshot has already been declared (use a *lazy* rank-tree policy). When snapshots are archived first and ranked later, snapshot discrimination can be costly because it requires copying. The lazy segregation protocol provides efficient lazy discimination by combining two techniques to reduce the cost of copying. First, it uses a more compact diff-based representation for snapshot pages so that there is less to copy. Second, the diff-based representation (as explained below) includes a component that has a page-based snapshot representation. This page-based component is segregated without copying using the eager segregation protocol.

Diff-based snapshots. The compact diff-based representation implements the same abstraction of snapshot pages and snapshot page tables, as the page-based snapshot representation. It is similar to database redo recovery log consisting of sequential repetitions of two types of components, checkpoints and diffs. The checkpoints are incremental page-based snapshots declared periodically by the storage management system. The diffs are versioned page diffs, consisting of versioned object modifications clustered by page. Since typically only a small fraction of objects in a page is modified by a transaction, and moreover, many attributes do not change, we expect the diffs to be compact.

The log repetitions containing the diffs and the checkpoints are archived sequentially, with diffs and checkpoints written into different archive data structures. Like in SNAP, the incremental snapshot page tables collect the archived page mappings for the checkpoint snapshots. A simple page index structure keeps track of page-diffs in each log repetition (the diffs in one log repetition are referred to as *diff extent*).

To create the diff-based representation, the cleaner sorts the diffs in an in-memory buffer, assembling the page-based diffs for the diff extents. The available sorting buffer size determines the length of diff extents. Since frequent checkpoints decrease the compactness of the diff-based representation, to get better compactness, the cleaner may create several diff extents in a single log repetition. Increasing the number of diff extents slows down BITE. This trade-off is similar to the recovery log. For brevity, we omit the details of how the diff-based representation is constructed. The details can be found in [16]. The performance section discusses some of the issues related to the diff-based representation compactness that are relevant to the snapshot storage management performance.

The snapshots declared between checkpoints are reconstructed by first mounting the snapshot page table for

the closest (referred to as *base*) checkpoint and the corresponding diff-page index. This allows BITE to access the checkpoint pages, and the corresponding page-diffs. To reconstruct P_v, the version of P in snapshot v, the server reads page P from the checkpoint, and then reads in order, the diff-pages for P from all the needed diff extents and applies them to the checkpoint P in order. Figure 6 shows an example of reconstructing a page P in

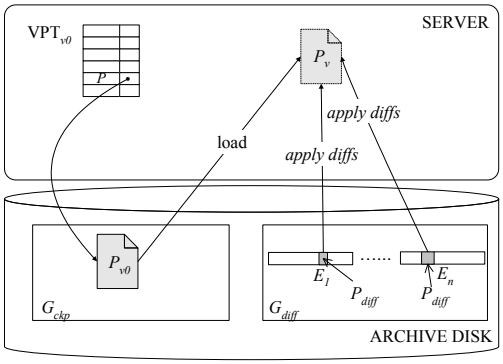

Figure 6: BITE: diff-based representation

a diff-based snapshot v from a checkpoint page P_{v_0} and diff-pages contained in several diff extents.

Segregation. When an application eventually provides a snapshot ranking, the system simply reads back the archived diff extents, assembles the diff extents for the longer-lived snapshots, creates the corresponding long-lived base checkpoints, and archives the retained snapshots sequentially into a longer-lived area. If diffs are compact, the cost of copying is low.

The long-lived base checkpoints are created without copying by separating the long-lived and short-lived checkpoint pages using eager segregation. Since checkpoints are simply page-based snapshots declared periodically by the system, the system can derive the ranks for the base checkpoints once the application specifies the snapshot ranks. Knowing ranks at checkpoint declaration time enables eager segregation.

Consider two adjacent log repetitions L_i, L_{i+1} for level-1 snapshots, with corresponding base checkpoints B_i, and B_{i+1}. Suppose the base checkpoint B_{i+1} is to be reclaimed when the adjacent level-1 diff extents are merged into one level 2 diff extent. Declaring the base checkpoint B_i a level-2 *rank tree* snapshot, and base checkpoint B_{i+1} as level-1 *rank tree* snapshot, allows to reclaim the pages of B_{i+1} without fragmentation or copying.

Figure 7 shows an example eager rank-tree policy for checkpoints in lazy segregation. A representation for level-1 snapshots has the diff extents E_1, E_2 and E_3 (in the archive region G^1_{diffs}) associated with the base

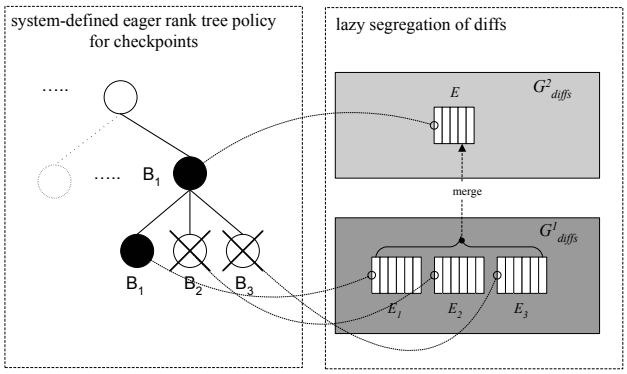

Figure 7: Lazy segregation

checkpoints B_1, B_2 and B_3. To create the level-2 snapshots, E_1, E_2 and E_3 are merged into extent E (in region G^2_{diffs}). This extent E has a base checkpoint B_1. Eventually, extents E_1, E_2, E_3 and checkpoints B_2, B_3 are reclaimed. Since B_1 was ranked at declaration time as rank-2 longer-lived snapshot, the eager segregation protocol lets B_1 capture all the checkpoint pages it records, allowing to reclaim the shorter-lived pages of B_2 and B_3 without fragmentation.

Our lazy segregation protocol is optimized for the case where the application specifies snapshot rank within a limited time period after snapshot declaration, which we expect to be the common case. If the limit is exceeded, the system reclaims shorter-lived base checkpoints by copying out longer-lived pages at a much higher cost. The same approach can also be used if the application needs to change the discrimination policy.

4.3 Faster BITE

The diff-based representation is more compact but has a slower BITE than the page-based representation. Some applications require lazy discrimination but also need low-latency BITE on a recent window of snapshots. For example, to examine the recent snapshots and identify the ones to be retained. The eager segregation protocol allows efficient composition of diff-based and page-based representations to provide fast BITE on recent snapshots, and lazy snapshot discrimination. The composed representation, called *hybrid*, works as follows. When an application declares a snapshot, hybrid creates two snapshot representations. A page-based representation is created in a separate archive region that maintains a sliding window of W recent snapshots, reclaimed temporally. BITE on snapshots within W runs on the fast page-based representation. In addition, to enable efficient lazy discrimination, hybrid creates for the snapshots a diff-based representation. BITE on snapshots outside W runs on the slower diff-based representation.

Snapshots within W therefore have two representations (page-based and diff-based).

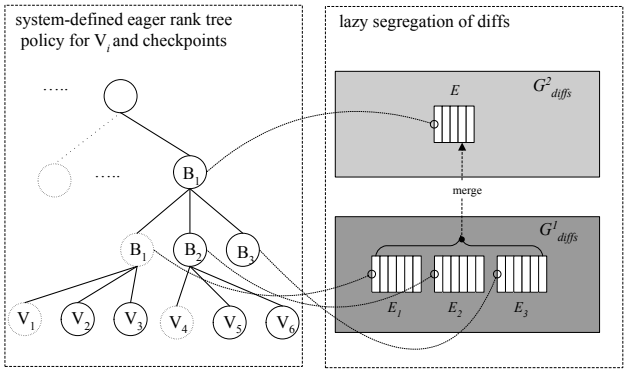

Figure 8: Reclamation in Hybrid

The eager segregation protocol can be used to efficiently compose the two representations and provide efficient reclamation. To achieve the efficient composition, the system specifies an *eager* rank-tree policy that ranks the page-based snapshots as lowest-rank (level-0) rank-tree snapshots, but specifies the ones that correspond to the system-declared checkpoints in the diff-based representation, as level-1. As in lazy segregation, the checkpoints can be further discriminated by bumping up the rank of the longer-lived checkpoints. With such eager policy, the eager segregation protocol can retain the snapshots declared by the system as checkpoints without copying, and can reclaim the aged snapshots in the page-based window W without fragmentation. The cost of checkpoint creation and segregation is completely absorbed into the cost of creating the page-based snapshots, resulting in lower archiving cost than the simple sum of the two representations.

Figure 8 shows reclamation in the hybrid system that adds faster BITE to the snapshots in Figure 7. The system creates the page-based snapshots V_i and uses them to run fast BITE on recent snapshots. Snapshots V_1 and V_4 are used as base checkpoints B_1 and B_2 for the diff-based representation, and checkpoint B_1 is retained as a longer-lived chekpoint. The system specifies an eager rank-tree policy, ranking snapshots V_i at level-0, and bumping up V_1 to level-2 and V_4 to level-1. This allows the eager segregation protocol to create the checkpoints B_1, and B_2, and eventually reclaim B_2, V_5 and V_6 without copying.

5 Performance

Efficient discrimination should not increase significantly the cost of snapshots. We analyze our discrimination techniques under a range of workloads and show they have minimal impact on the snapshot system performance. Section 6 presents the results of our experiments. This section explains our evaluation approach.

Cost of discrimination. The metric λ_{dp} [12] captures the non-disruptiveness of an I/O-bound storage system. We use this metric to gauge the impact of snapshot discrimination. Let r_{pour} be the "pouring rate" – average object cache (MOB/VMOB) 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 MOB/VMOB cleaning. We define:

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

λ_{dp} indicates how well the draining keeps up with the pouring. If $\lambda_{dp} \geq 1$, the system operates within its capacity and the foreground transaction performance is not be affected by background cleaning activities. If $\lambda_{dp} < 1$, the system is overloaded, transaction commits eventually block on free object cache space, and clients experience commit delay.

Let t_{clean} be the average cleaning time per dirty database page. Apparently, t_{clean} determines r_{drain}. In Thresher, t_{clean} reflects, in addition to the database ireads and writes, the cost of snapshot creation and snapshot discrimination. Since snapshots are created on a separate disk in parallel with the database cleaning, the cost of snapshot-related activity can be partially "hidden" behind database cleaning. Both the update workload, and the compactness of snapshot representation affect r_{pour}, and determine how much can be hidden, i.e. non-disruptiveness.

Overwriting (α) is an update workload parameter, defined as the percentage of repeated modifications to the same object or page. α affects both r_{pour} and r_{drain}. When overwriting increases, updates cause less cleaning in the storage system because the object cache (MOB/VMOB) absorbs repeated modifications, but high frequency snapshots may need to archive most of the repeated modifications. With less cleaning, it may be harder to hide archiving behind cleaning, so snapshots may become more disruptive. On the other hand, workloads with repeated modifications reduce the amount of copying when lazy discrimination copies diffs. For example, for a two-level discrimination policy that retains one snapshot out of every hundred, of all the repeated modifications to a given object o, archived for the short-lived level-1 snapshots, only one (last) modification gets retained in the level-2 snapshots. To gage the impact of discrimination on the non-disruptiveness, we measure r_{pour} and r_{drain} experimentally in a system with and without discrimination for a range of workloads with

low, medium and high degree of overwriting, and analyze the resulting λ_{dp}.

λ_{dp} determines the maximum throughput of an I/O bound storage system. Measuring the maximum throughput in a system with and without discrimination could provide an end-to-end metric for gauging the impact of discrimination. We focus on λ_{dp} because it allows us to explain better the complex dependency between workload parameters and cost of discrimination.

Compactness of representation. The effectiveness of diff-based representation in reducing copying cost depends on the *compactness* of the representation. We characterize compactness by a relative snapshot retention metric R, defined as the size of snapshot state written into the archive for a given snapshot history length H, relative to the size of the snapshot state for H captured in full snapshot pages. $R = 1$ for the page-based representation. R of the diff-based representation has two contributing components, R_{ckp} for the checkpoints, and R_{diff} for the diffs. *Density* (β), a workload parameter defined as the fraction of the page that gets modified by an update, determines R_{diff}. For example, in a static update workload where any time a page is updated, the same half of the page gets modified, $R_{diff} = 0.5$. R_{ckp} depends on the frequency of checkpoints, determined by L – the number of snapshots declared in the history interval corresponding to one log repetition. In workloads with overwriting, increasing L decreases R_{ckp} since checkpoints are page-based snapshots that record the first pre-state for each page modified in the log repetition. Increasing L by increasing d, the number of diff extents in a log repetition, raises the snapshot page reconstruction cost for BITE. Increasing L without increasing d requires additional server memory for the cleaner to sort diffs when assembling diff pages.

Diff-based representation will not be compact if transactions modify all the objects in a page. Common update workloads have sparse modifications because most applications modify far fewer objects than they read. We determine the compactness of the diff-based representation by measuring R_{diff} and R_{ckp} for workloads with expected medium and low update density.

6 Experimental evaluation

Thresher implements in SNAP [12] the techniques we have described, and also support for recovery during normal operation without the failure recovery procedure. This allows us to evaluate system performance in the absence of failures. Comparing the performance of Thresher and SNAP reveals a meaningful snapshot discrimination cost because SNAP is very efficient: even at high snapshot frequencies it has low impact on the storage system [12].

Workloads. To study the impact of the workload we use the standard multiuser OO7 benchmark [2] for object storage systems. We omit the benchmark definition for lack of space. An OO7 transaction includes a read-only traversal (T1), or a read-write traversal (T2a or T2b). The traversals T2a and T2b generate workloads with fixed amount of object overwriting and density. We have implemented extended traversal summarized below that allow us to control these parameters. To control the degree of overwriting, we use a variant traversal T2a' [12], that extends T2a to update a randomly selected *AtomicPart* object of a *CompositePart* instead of always modifying the same (*root*) object in T2a. Like T2a, each T2a' traversal modifies 500 objects. The desired amount of overwriting is achieved by adjusting the object update history in a sequence of T2a' traversals. Workload parameter α controls the amount of overwriting. Our experiments use three settings for α, corresponding to low (0.08), medium (0.30) and very high (0.50) degree of overwriting.

To control density, we developed a variant of traversal T2a', called T2f (also modifies 500 objects), that allows to determine β, the average number of modified *AtomicPart* objects on a dirty page when the dirty page is written back to database (on average, a page in OO7 has 27 such objects). Unlike T2a' which modifies one *AtomicPart* in the *CompositePart*, T2f modifies a group of *AtomicPart* objects around the chosen one. Denote by T2f-*g* the workload with group of size *g*. T2f-1 is essentially T2a'.

The workload density β is controlled by specifying the size of the group. In addition, since repeated T2f-g traversals update multiple objects on each data page due to write-absorption provided by MOB, T2f-g, like T2a', also controls the overwriting between traversals. We specify the size of the group, and the desired overwriting, and experimentally determine β in the resulting workload. For example, given 2MB of VMOB (the standard configuration in Thor and SNAP for single-client workload), the measured β of multiple T2f-1 is 7.6 (medium α, transaction 50% on private module, 50% on public module). T2f-180 that modifies almost every *AtomicPart* in a module, has β = 26, yielding almost the highest possible workload density for OO7 benchmark. Our experiments use workloads corresponding to three settings of density β, low (T2f-1,β=7.6), medium (T2f-26,β=16) and very high (T2f-180,β=26) Unless otherwise specified, a medium overwriting rate is being used.

Experimental configuration. We use two experimental system configurations. The single-client experiments run with snapshot frequency 1, declaring a snapshot after each transaction, in a 3-user OO7 database (185MB

in size). The multi-client scalability experiments run with snapshot frequency 10 in a large database (140GB in size). The size of a single private client module is the same in both configurations. All the reported results show the mean of at least three trials with maximum standard deviation at 3%.

The storage system server runs on a Linux (kernel 2.4.20) workstation with dual 64-bit Xeon 3Ghz CPU, 1GB RAM. Two Seagate Cheetah disks (model ST3146707LC, 10000 rpm, 4.7ms avg seek, Ultra320 SCSI) directly attach to the server via LSI Fusion MPT adapter. The database and the archive reside on separate raw hard disks. The implementation uses Linux raw devices and direct I/O to bypass file system cache. The client(s) run on workstations with single P3 850Mhz CPU and 512MB of RAM. The clients and server are inter-connected via a 100Mbps switched network. In single-client experiments, the server is configured with 18 MB of page cache (10% of the database size), and a 2MB MOB in Thor. In multi-client experiments, the server is configured with 30MB of page cache and 8-11MB of MOB in Thor. The snapshot systems are configured with slightly more memory [12] for VMOB so that the same number of dirty database pages is generated in all snapshot systems, normalizing the r_{drain} comparison to Thor.

6.1 Experimental results

We analyze in turn, the performance of eager segregation, lazy segregation, hybrid representation, and BITE under a single-client workload, and then evaluate system scalability under a multiple concurrent client workload.

6.1.1 Snapshot discrimination

Eager segregation. Compared to SNAP, the cost of eager discrimination in Thresher includes the cost of creating VPTs for higher-level snapshots. Table 1 shows t_{clean} in Thresher for a two-level eager rank-tree with inter-level retention fraction fraction f set to one snapshot in 200, 400, 800, and 1600. The t_{clean} in SNAP is 5.07ms. Not surprisingly, the results show no noticeable change, regardless of retention fraction. The small incremental page tables contribute a very small fraction (0.02% to 0.14%) of the overall archiving cost even for the lowest-level snapshots, rendering eager segregation essentially free of cost. This result is important, because eager segregation is used to reduce the cost of lazy segregation and hybrid representation.

Table 1: t_{clean}: eager segregation

f	200	400	800	1600
t_{clean}	5.08ms	5.07ms	5.10ms	5.08ms

Lazy segregation. We analyzed the cost of lazy segregation for a 2-level rank-tree by comparing the cleaning costs, and the resulting λ_{dp} in four different system configurations, Thresher with lazily segregated diff-based snapshots ("Lazy"), Thresher with unsegregated diff-based snapshots ("Diff"), page-based (unsegregated) snapshots("SNAP"), and storage system without snapshots ("Thor"), under workloads with a wide range of density and overwriting parameters. The complete results, omitted for lack of space, can be found in [16]. Here we focus on the low and medium overwriting and density parameter values we expect to be more common.

Table 2: Lazy segregation and overwriting

α		t_{clean}	t_{diff}	λ_{dp}
low				
	Lazy	5.30ms	0.13ms	2.24
	Diff	5.28ms	0.08ms	2.26
	SNAP	5.37ms		2.24
	Thor	5.22ms		2.30
medium				
	Lazy	4.98ms	0.15ms	3.67
	Diff	5.02ms	0.10ms	3.69
	SNAP	5.07ms		3.72
	Thor	4.98ms		3.79
high				
	Lazy	4.80ms	0.21ms	4.58
	Diff	4.80ms	0.14ms	4.66
	SNAP	4.87ms		4.61
	Thor	4.61ms		4.83

A key factor affecting the cleaning costs in the diff-based systems is the compactness of the diff-based representation. A diff-based system configured with a 4MB sorting buffer, with medium overwriting, has a very low R_{ckp} (0.5% - 2%) for the low density workload (R_{diff} is 0.3%). For medium density workload (R_{diff} is 3.7%), the larger diffs fill sorting buffer faster but R_{ckp} decreases from 10.1% to 4.8% when d increases from 2 to 4 diff extents. These results point to the space saving benefits offered by the diff-based representation.

Table 2 shows the cleaning costs and λ_{dp} for all four systems for medium density workload with low, medium, and high overwriting. The t_{clean} measured in the Lazy and Diff systems includes the database iread and write cost, the CPU cost for processing VMOB, the page

archiving and checkpointing cost via parallel I/O, snapshot page table archiving, and the cost for sorting diffs and creating diff-extents but does not directly include the cost of reading and archiving diffs, since this activity is performed asynchronously with cleaning. The measured t_{diff} reflects these diff related costs (including I/O on diff extents, and diff page index maintenance) per dirty database page. The t_{clean} measured for SNAP and Thor includes the (obvious) relevant cost components.

Compared to Diff, Lazy has a higher t_{diff} reflecting the diff copying overhead. This overhead decreases as overwriting rate increases. t_{diff} does not drop proportionally to the overwriting increase because the dominant cost component of creating higher level extents, reading back the extents in the lowest level, is insensitive to the overwriting rate. Lazy pays no checkpoint segregation cost because it uses the eager protocol.

Next consider non-disruptiveness. We measure r_{pour} and conservatively compute λ_{dp} for Diff and Lazy by adding t_{diff} to t_{clean}, approximating a very busy system where diff I/O is forced to run synchronously with the cleaning. When overwriting is low, λ_{dp} in all snapshot systems is close to Thor. When overwriting is high, all systems have high λ_{dp} because there is very little cleaning in the storage system, and R is low in Diff and Lazy. Importantly, even with the conservative adjustment, λ_{dp} in both diff-based systems is very close to SNAP, while providing significantly more compact snapshots. Notice, all snapshot systems declare snapshots after each traversal transaction. [12] shows that λ_{dp} increases quickly as snapshot frequency decreases.

Hybrid. The Hybrid system incurs the archiving costs of a page-based snapshot system, plus the costs of diff extent creation and segregation, deeming it the costliest of Thresher configurations. Workload density impacts the diff-related costs. Figure 9 shows how the non-disruptiveness λ_{dp} of Hybrid decreases relative to Thor for workloads with low, medium and high density and a fixed medium overwriting. The denser workload implies more diff I/O. Under the densest possible workload, included here for comparison, the drop of λ_{dp} of hybrid over Thor is 13.6%, where for the common expected medium and low workloads the drop is 3.2% and 1.8% respectively. Note in all configurations, because system's λ_{dp} is greater than 1, there is no client-side performance difference observed between Hybrid and Thor. As a result, the metric λ_{dp} directly reflects the server's "cleaning" speed (t_{clean}). The results in Figure 9 indicate that *Hybrid* is a feasible solution for systems that need fast BITE and lazy discrimination (or snapshot compactness).

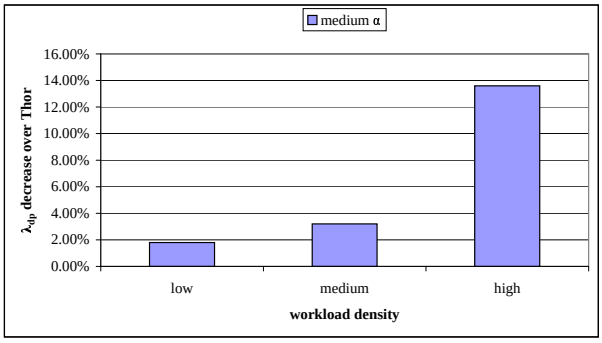

Figure 9: Hybrid: λ_{dp} relative to Thor

6.1.2 Back-in-time execution

We compare BITE in Diff and SNAP to Thor. Our experiment creates Diff and SNAP archives by running 16000 medium density, medium overwriting traversals declaring a snapshot after each traversal. The incremental VPT protocol [12] checkpoints VPTs at 4MB intervals to bound reconstruction scan cost. The APT mounting time, depending on the distance from the VPT checkpoint is of a seesaw pattern, between 21.05ms and 48.77ms. The latency of BITE is determined by the average fetch cost via APT (4.10ms per page).

Diff mounts snapshot by mounting the closest checkpoint, i.e. reconstructing the checkpoint page table (same cost as VPT in SNAP), and mounting the involved page index structures at average mounting time of page index at 7.61ms. To access a page P, Diff reads the checkpoint page and the d diff-pages of P. The average cost to fetch a checkpoint page is 5.80ms, to fetch a diff-page from one extent is 5.42ms. The cost of constructing the requested page version by applying the diff-pages back to the checkpoint page is negligible.

Table 3: End-to-end BITE performance

	current db	page-based	diff-based
T1 traversal	17.53s	27.06s	42.11s

Table 3 shows the average end-to-end BITE cost measured at the client side by running one standard OO7 T1 traversal against Thor, SNAP and Diff respectively. Hybrid has the latency of SNAP for recent snapshots, and latency of Diff otherwise. The end-to-end BITE latency (page fetch cost) increases over time as pages are archived. Table 3 lists the numbers corresponding to a particular point in system execution history with the intention of providing general indication of BITE performance on different representations compared to the performance of accessing the current database. The perfor-

mance gap between page-based and diff-based BITE motivates the hybrid representation.

6.1.3 Scalability

To show the impact of discrimination in a heavily loaded system we compare Thresher (hybrid) and Thor as the storage system load increases, for single-client, 4-client and 8-client loads, for medium density and medium overwriting workload. (An 8-client load saturates the capacity of the storage system).

The database size is 140GB, which virtually contains over 3000 OO7 modules. Each client accesses its private module (45MB in size) in the database. The private modules of the testing clients are evenly allocated within the space of 140GB. Under 1-client, 4-client and 8-client workloads, the λ_{dp} of Thor is 2.85, 1.64 and 1.30 respectively. These λ_{dp} values indicate, that Thor is heavily loaded under multi-client workloads. Figure

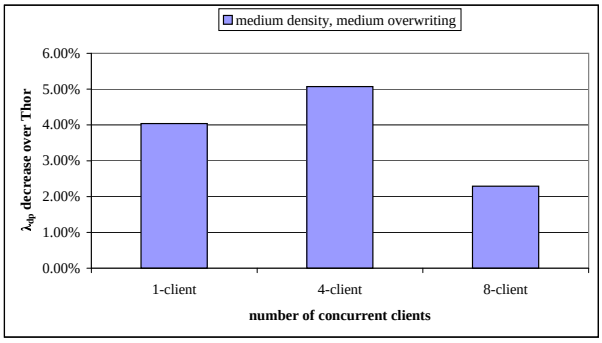

Figure 10: multiple clients: λ_{dp} relative to Thor

10 shows the decrease of λ_{dp} in Hybrid relative to Thor when load increases. Note, that adding more concurrent clients doesn't cause Hybrid to perform worse. In fact, with 8-client concurrent workload, Hybrid performs better than single-client workload. This is because with private modules evenly allocated across the large database, the database random read costs increase compared to the single-client workload, hiding the cost of sequential archiving during cleaning more effectively. Under all concurrent client workloads, Hybrid, the costliest Thresher configurations, is non-disruptive.

7 Related work

Most storage systems that retain snapshots use incremental copy-on-write techniques. To the best of our knowledge none of the earlier systems provide snapshot storage management or snapshot discrimination policies beyond aging or compression.

Versioned storage systems built on top of log-structured file systems and databses [13, 14], and write-anywhere storage [6], provide a low-cost way to retain past state by using no-overwrite updates. These systems does not distinguish between current and past states and use same representation for both. Recent work in ext3cow system [9], separates past and present meta-data states to preserve clustering, but uses no-overwrite updates for data.

Elephant [11] is an early versioned file system that provides consistent snapshots of a file system, allows faster access to recent versions, and provides a sliding window of snapshots but does not support lazy discrimination or different time-scale snapshots.

Compact diff-based representation for versions is used in the CVS source control system. Large-scale storage systems for archiving past state(e.g. [10, 17]) improve the compactness of storage representation (and reduce archiving bandwidth) by eliminating redundant blocks in the archive. These techniques, based on content hashes [10], and differential compression [17], incur high cost at version creation time and do not seem suited for non-disruptive creation of snapshots. However, these systems may benefit from snapshot discrimination.

Generational garbage collectors [15] use efficient storage reclamation techniques that reduce fragmentation by grouping together objects with similar lifetimes. The *rank tree* technique adopts a similar idea for immutable past states shared by snapshots with different lifetimes.

8 Conclusions

We have described new efficient storage management techniques for discriminating copy-on-write snapshots. The ranked segregation technique, borrowing from generational garbage collection, provides no-copy reclamation when the application specifies a snapshot discrimination policy eagerly at snapshot declaration time. Combining ranked segregation with a compact diff-based representation enables efficient reclamation when the application specifies the discrimination policy lazily, after snapshot declaration. Hybrid, an efficient composition of two representations, provides faster access to recent snapshots and supports lazy discrimination at low additional cost.

We have prototyped the new discrimination techniques and evaluated the effect of workload parameters on the efficiency of discrimination. The results indicate that our techniques are very efficient. Eager discrimination incurs no performance penalty. Lazy discrimination incurs a low 3% storage system performance penalty on expected common workoads. The diff-based representation provides more than ten-fold reduction in snapshot storage that can be further reduced with discrimination. Further-

more, the hybrid system that provides lazy discrimination and fast BITE incurs a 10% penalty to the storage system in the worst case of extremely dense update workload, and a low 4% penalty in the expected common case.

Snapshot discrimination could become an attractive feature in future storage systems. The paper has described the first step in this direction. Our prototype is based on a transactional object storage system, although we believe our techniques are more general. We have already applied them to a more general ARIES [5] STEAL system. A file system prototype would be especially worthwhile. It would require modifications to the file system interface along the lines of a recent proposal [3] to enable more efficient capture of updates.

References

[1] ADYA, A., GRUBER, R., LISKOV, B., AND MAHESHWARI, U. Efficient optimistic concurrencty control using loosely synchronized clocks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1995).

[2] CAREY, M. J., DEWITT, D. J., AND NAUGHTON, J. F. The OO7 Benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (Washington D.C., May 1993), pp. 12–21.

[3] DE LOS REYES, A., FROST, C., KOHLER, E., MAMMARELLA, M., AND ZHANG, L. The KudOS Architecture for File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP), WIP Session* (Brighton,UK, October 2005).

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

[5] GRAY, J. N., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

[6] HITZ, D., LAU, J., AND MALCOM, M. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference* (San Francisco, CA, January 1994).

[7] LISKOV, B., CASTRO, M., SHRIRA, L., AND ADYA, A. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)* (Lisbon, Portugal, June 1999).

[8] O'TOOLE, J., AND SHRIRA, L. Opportunistic Log: Efficient Installation Reads in a Reliable Storage Server. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, CA, November 1994).

[9] PETERSON, Z. N., AND BURNS, R. C. The Design, Implementation and Analysis of Metadata for Time Shifting File-system. *Technical Report HSSL-2003-03, Computer Science Department, The John Hopkins University* (Mar. 2003).

[10] QUINLAN, S., AND DORWARD, S. Venti: A New Approach to Archival Data Storage. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST)* (Monterey, CA, USA, January 2002).

[11] SANTRY, D., FEELEY, M., HUTCHINSON, N., VEITCH, A., CARTON, R., AND OFIR, J. 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).

[12] SHRIRA, L., AND XU, H. Snap: Efficient snapshots for back-in-time execution. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)* (Tokyo, Japan, Apr. 2005).

[13] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata Efficiency in Versioning File Systems. In *Proceedings of the 2nd Conference on File and Storage Technologies (FAST)* (San Francisco, CA, USA, March 2003).

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

[15] UNGAR, D., AND JACKSON, F. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems 14*, 1 (Mar. 1992), 1–27.

[16] XU, H. *Timebox: A High Performance Archive for Split Snapshots*. PhD thesis, Brandeis University, Dec. 2005.

[17] YOU, L., AND KARAMANOLIS, C. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st IEEE Symposium on Mass Storage Systems and Technologies (MSST)* (College Park, MD, Apr. 2004).