RSnap: Recursive Writable Snapshots for Logical Volumes

by

Abhishek Gupta

B.E., Visvesvaraya National Institute of Technology, Nagpur, India, 2002

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

September, 2006

Abstract

Volume Snapshots provide instantaneous checkpoints of a complete file-system. Whereas previous work on developing snapshot utilities has focused on satiating the need for online backups, modern trends encourage their deployment in expedited clone construction scenarios. Unlike their read-only counterparts, these clones are expected to be mutable and to some extent independent of their base images.

We present RSnap, a volume manager that allows users to create mutable recursive snapshots of logical volumes. Storage is allocated to these snapshots in a Dedicate-on-Write fashion. Since unrestrained clone creation can quickly obscure sharing relationships between a set of volumes, RSnap includes an automatic garbage collector that can identify and reclaim unreferenced blocks.

RSnap uses the radix tree data structure to implement recursive snapshots. We capitalize on kernel infrastructure to overcome some of the performance and consistency problems in using the radix tree. Our evaluations show that an in kernel radix tree implementation gives satisfactory performance and scales with the number of snapshots.

Table of Contents

List of Tables

List of Figures

Acknowledgements

As I pen these acknowledgements as a final step in the realization of my thesis, I feel equally thrilled and sad. Thrilled because of the opportunities that await me in the outside world and sad because I am leaving so much unexplored in this academic realm. Yet, what gives me happiness is that no matter where life takes me I will continue to be a part of this department, this institution and this community. So, consider these words as a tribute to all of you who were involved in shaping the two most memorable years in my life.

A big thank you to Norm, not just for guiding my research but trusting me more than I trust myself. This work would never have been the way it is now without your insight, experience and support. Also, you are the best role model I have ever had. A big thank you to Buck for being my second reader, a terrific mentor and a very clever critic. Thank you for all the patient listening you gave to my vague random ideas throughout our interaction. I will be coming back for many more discussions hereafter. A big thank you to Andy Warfield for his useful comments on my thesis. The Parallax project will always remain a source of inspiration for me. A very special thanks to Mike Feeley for his inspiring class on Operating Systems.

A heartfelt thank you to the great folks at DSG. Thank you Jake for your intelligent help on all aspects of my thesis, your intriguing insights in to the classiest things in life and for being a great great buddy. Thank you Kan for all the sweet music, the fabulous Chinese snacks and for being a very caring friend. Thanks to Brendan for teaching me how to compile my very first kernel module and for solving the infinite configuration problems I showed up with. Also thanks to Anirban, Andre, Brad, Gang, Mike(Tsai), Camillo, Cuong and Geoffrey for all the the geeky talk and fun times.

A big thank you to Dave, Mike(Yurick), Clint, Jordan and Jennifer from the bullpen days. Your company actually made those grueling times enjoyable. A special thanks to Irene for showing me around SFU and introducing me to the "The Bridge Across Forever". A big thank you to Satyajit for being the first to befriend me in Vancouver. Finally, a huge thank you to Pavan for his never-ending support, friendship and encouragement.

Chapter 1

Introduction

Storage virtualization has always been a topic of immense interest. Typically, it is defined as the process of coalescing many different physical storage networks and devices, and making them appear as one virtual entity for purposes of management and administration. From Petal[16] in the 90s to the more contemporary Parallax[28], researchers and vendors have contemplated upon fascinating combinations of software and hardware techniques to virtualize the block layer. Some of these techniques are kernel based, whereas others are user space device drivers and some are even hardware implementations of the block layer[25].

A useful feature which usually comes gratis with block layer virtualization is the ability to take live snapshots of logical volumes. Volume snapshots can be described as instantaneous frozen images of a disk and are a natural choice for taking backups, doing disaster recovery and even for setting up file-system agnostic versioning. Recently, the need to support snapshot writability has become imperative and classical logical volume management systems have been upgraded with limited copy-on-write implementations.

Snapshots are particularly useful in environments running virtual machines (VMs), wherein administrators prepare an initial golden root image and later export a writable snapshot to every newly instantiated guest operating system. Not only does this cut down upon the time taken to prepare entire root file system images but is also a great way to conserve disk space. The task of implementing writable snapshots is further complicated with the availability of fork in popular VM technology. Forking a VM requires sophisticated copy-on-write mechanisms to replicate both incore memory and secondary storage. Whereas forking incore memory has been studied in depth under the Potemkin[27] project, one of our intentions through this work is to present a detailed analysis of forking complete logical volumes. It is likely that the VM fork facility will be soon extended into a recursive fork feature. A facility to recursively fork the file system will be of good advantage then.

Network emulation environments such as Emulab can also benefit from writable recursive snapshots. In these setups, remote users create their testbed environments for varying periods of time. One can imagine the incremental creation of test environments by always adding software to writable snapshots of some base image. The newly modified snapshots can then become future candidate base images. In this way, if some configuration becomes unusable, a user can start again by taking a writable snapshot of some intermediate snapshot of her choice. Also, unrelated users can rapidly construct their environments by taking writable snapshots of disk images prepared by their peers. Emulab's current mechanism for imaging disks[8] is quite brute force and can enjoy significant savings in storage and time if used in conjunction with RSnap.

Mutable snapshots can also emulate thin provisioning technology. A recent Gartner report[18] suggests that applications are often allocated storage which is much higher than their actual needs, thus leading to underutilized storage. In these scenarios, a thin provisioning system provides a client with an illusion of large sized virtual volumes that have no physical storage allocated to them. Storage is physically allocated only when the client tries to write to these volumes. One can imagine a situation where an administrator prepares one fixed size null device image and then distributes arbitrarily sized snapshots of that image to satisfy application requests. Later, actual space gets allocated as writes are made to the snapshot. Major storage vendors such as Network Applicance, 3-Par and VMWare have incorporated such techniques in their popular products.

A more recent use of snapshots has been observed in snapshotting live databases. Database snapshots facilitate the maintainance of multiple copies of the database for reporting or archival purposes with minimal storage overhead. The writability property of such snapshots can be useful to administrators in selectively dropping or adding users, adding new indexes for reporting, developing indexed views, changing stored procedures, or modifying the database in ways similar to the creation of a DSS/Reporting database out of an OLTP one.

Snapshots are also useful in performing activities related to intrusion detection[14] and computer forensics[29]. The principle requirement there is to be able to capture and replay through the historical states of a system. Volume snapshots are used to checkpoint the storage state of the system under audit. Recent work on intrusion detection has also leveraged virtual machine cloning[27] to create large scale honey-farm environments. The rate of virtual machine deployment in these honeyfarms is expected to be as high as the rate of packet arrival over the network. Although the system prototype in [27] capitalized on a RAM disk for fast virtual machine deployment, a utility for fast disk backed file-system cloning will be of immense value.

Keeping the versatility of snapshots in perspective, we have developed RSnap, a block layer virtualization infrastructure designed specifically for supporting high performance writable recursive snapshots of some original volume. The keyword *recursive* implies that RSnap allows a user to create snapshots of snapshots, whereas, the keyword *writable* implies that snapshots can be mutable. Most volume managers implement snapshots as a special read-only case, whereas, RSnap considers each snapshot as a distinct first class logical volume. Treating each snapshot as a volume which happens to share copy-on-write blocks with its parent grants it considerable freedom with regard to growth and lifetime but results in an inflexible storage reclamation process. RSnap counters such inflexibility through a fast fault tolerant garbage collector that can continue in parallel to regular I/O operations on the system.

The rest of this thesis is organized as follows: Chapter 2 is a survey of previous work and highlights our specific contributions. Chapter 3 is a detailed description of our design. Chapter 4 presents the implementation details of RSnap. Chapter 5 covers some interesting performance measurements and finally we conclude in Chapter 6.

Chapter 2

Related Work and Contributions

This chapter is a brief survey of existing systems that implement snapshot technology. Broadly, they can be categorized into volume managers and file-systems. Whereas volume managers checkpoint the instantaneous image of the whole disk, file-systems implement snapshots at the granularity of files. We explore how earlier systems used snapshots as a mechanism to support physical backups and file-system level undo operations, whereas, modern systems use it as a tool for quick file-system replication. Towards the end, we compare and contrast these systems with RSnap.

2.1 Volume Managers

Weibren et. al. first proposed the notion of a Logical Disk[2]. The motivation was to separate the responsibilities of managing files and disk blocks by introducing an interface between the two. Under this arrangement a file-system can only address logical blocks of a virtual disk or a volume. Logical block numbers are transparently converted into physical block numbers by the storage management infrastructure. Other than adding modularity to the operating system code, an advantage of this mechanism is that the layout of blocks on the disk can be transparently reorganized to reduce disk access times. Although, their particular interest was in demonstrating how such a system can be used to provide LFS[21] equivalent write performance in a file-system agnostic manner, they set the stage for stackable file-systems research[7].

The Petal[16] system took the logical disk concept to the next level by implementing a distributed storage service. Petal servers comprise of multiple machines hosting several commodity disks which provide a client with sparse 64-bit byte volumes. Physical storage corresponding to a Petal virtual disk is resilient to single component failures and can be geographically distributed. These virtual disks can also be expanded in performance and capacity as more servers and disks are added to the pool of resources. One of the distinguishing features of the Petal system was its ability to take live snapshots of virtual disks. The intent was to assist a client in automating the task of creating volume backups.

In order to manage snapshots, Petal maintains an explicit epoch number in its data structures. The epoch-number is incremented on every snapshot event so as to distinguish between data written to the same virtual disk at different instances of time. After a snapshot is taken, all accesses to the virtual disk are translated by using tuples associated with the new epoch-number, whereas the newly created snapshot uses the entries containing the older epoch-number. Copy-on-write is triggered whenever writes occur on the snapshotted virtual disk which results in entries containing the most recent epoch-number being added to the various data structures. Petal does not allow the modification of snapshots as their purpose is merely to facilitate backups.

Peabody[12] is a prototype volume management system that proposes continuous versioning of disk state. The primary motivation behind Peabody was to develop an ability to "undo" any change to the file-system. The authors note that implementing such a facility in the file-system itself can be complex and must be thought out during the initial stages of its development. Therefore, it would be worthwhile to implement it at the block layer instead. They further state that even though equivalent facilities are available in several commercial volume managers most of them are only used at enterprise scale and are an overkill for common desktop environments. Peabody is implemented by extending the Intel iSCSI target and is mountable by any iSCSI compatible initiator. The iSCSI protocol mandates synchronous completion of I/O requests before a response can be sent to the requestor. Since deploying Peabody entails updating additional metadata for every write, clients can expect some performance degradation.

Whereas most volume managers involve an explicit snapshot creation operation, Peabody transparently maintains all versions of disk blocks by keeping a write log data structure. One of the problems in versioning all states of the disk is that several of them would be inconsistent. Hence, Peabody relies on the assumption that its host file-system will have a consistency restore mechanism. A secondary problem is that it puts tremendous pressure on storage resources and therefore the authors propose the use of MD5 based content hashing to silence duplicate writes. In this context the authors demonstrated that for some workloads up to 84% of disk writes were actually duplicates.

Clotho[4] is another volume versioning system designed for desktop environments. Unlike Peabody, Clotho is not intended to manage multiple volumes and is targeted more towards single disk laptop environments. Also, Clotho does not automatically preserve all versions of a volume but instead allows the user to customize the frequency of snapshots. Users can configure Clotho to create a snapshot of the volume every hour, or whenever all files to the device are closed or on every single write to the device. Further, instead of using content hashing for coalescing duplicate requests, Clotho uses off-line differential compression to reduce the disk space overhead for archived versions of the volume.

Clotho makes use of the Linux stackable file-system architecture and plugs itself below existing file-systems. Clients are provided with a block device interface and a set of primitives which can be used for various management tasks such as snapshot creation, deletion, listings and compaction. Clotho operates by partitioning the disk into primary and backup data segments. The primary data segment contains the data corresponding to the latest version of the volume whereas the backup data segment holds all the data of archived versions or snapshots. When the backup data segment is completely filled up, the user is expected to create more space by either deleting or compacting older versions.

Clotho snapshots are read-only and are accessible through a virtual block device created in the file-system. Clotho uses a Logical Extent Table (LXT) to translate all virtual block numbers into physical addresses. The LXT is conceptually an array indexed by logical block numbers. Additionally, entries for previous versions of a block are chained to the most recent LXT entry, so as to form a linked list of versions. This minimizes the number of accesses required to translate the block addresses belonging to the most recent version of the volume. However, accessing earlier versions is expensive and might entail the traversal of the entire linked list.

The Linux Volume Manager (LVM2)[26] is another volume management system, which is used extensively in both home and production environments. LVM2 allows a user to create a volume group by coalescing multiple physical devices together. These volume groups can be expanded by adding more physical devices later. Users can carve out individual logical volumes out of existing volume groups. Space within each volume group is managed with a set of user space utilities provided by LVM2. In addition to providing conventional volume management support, LVM2 allows users to tag their volumes as encrypted, mirrored or redundant as well.

For each logical volume, LVM2 creates a virtual device in the user's file-system. The driver corresponding to these virtual devices is the LVM2 kernel component, which is also known as the device-mapper. The device-mapper transparently redirects I/O requests on a virtual device to its corresponding physical device. In most cases, it is a simple 1-1 mapping between logical and physical sector numbers, however, if logical blocks in the volume traverse multiple physical devices, then a B-Tree is used to efficiently calculate the corresponding physical device and offset.

LVM2 implements snapshots by reserving a fixed amount of contiguous space within the volume group. After a snapshot of the volume is taken, LVM2 maintains a table of exceptions corresponding to sectors overwritten either in the original volume or the snapshot. In case sectors are written on the original volume before they have been written in the snapshot, the device-mapper synchronously copies old sectors to the space reserved for the snapshot and then updates the table of sector exceptions.

The Federated Array of Bricks (FAB)[22] project required a more sophisticated snapshot implementation. FAB is an attempt to construct a distributed storage system using *bricks*-small rack-mounted computers built using commodity disks, CPU and NVRAM-connected over the ethernet. FAB capitalizes upon the economies of scale inherent in mass production. As an illustration, the authors cite the possibility of constructing a brick built out of 12 SATA disks and 1GB of NVRAM for under $2000. Even with triple-replication the cost of such a system can be kept at 20-80% of modern enterprise scale solutions. The authors realize the fact that commodity systems are much more prone to failure and hence suggest the use of voting based replication and erasure code schemes to provide reliability and continuity in service. Like most volume managers, FAB offers a block device interface based on the Intel iSCSI protocol.

Since FAB replicates the physical blocks belonging to a volume over multiple bricks, the task of creating a snapshot is slightly more complicated. The snapshot process must ensure that distributed segments of the volume are in sync at the instant of snapshot creation. Typically, taking a snapshot involves suspending the applications using the volume for a brief interval of time. Due to clock skews and latency delays in a cluster environment, it is hard to keep this interval brief. FAB follows a two phase approach. In the first phase the system co-ordinator sends a snapshot creation request along with a recent time-stamp. All bricks respond to the request by creating a new tentative mapping structure and sending a positive response. The co-ordinator then tries to establish a quorum based on these responses. Depending upon whether a quorum is established or not, the co-ordinator sends out a commit or rollback request in the second phase.

Frisbee[8] is a disk imaging system, which forms an integral part of the Emulab software. Frisbee enables fast and convenient snapshot, transfer and installation operations on entire disk images. The Emulab testbed provides experimenters with privileged access to remote machines. In such a scenario it is important to roll back the machine to a stable state after an experiment switch has been made. Also, in case the users corrupt their file-system or wish to install a customized file-system, Frisbee allows them to automatically reload their images at experiment initialization time.

Frisbee's operations are divided into three phases: image creation, image distribution and image installation. When a user decides to create an image out of their existing file-system, their machine is rebooted with a RAM disk based version of UNIX. This satisfies the need for disk quiescence. Next, the file-system hosted upon a partition of interest is identified and a corresponding format-aware module is invoked to create a list of free blocks. In the second stage only allocated blocks are read and compressed to produce 1MB chunks. Slicing the disk into chunks allows the distribution of the disk image out of order. Also, during the installation stage it allows overlap of decompression with disk I/O. Frisbee performs compression/decompression operations on the clients so as to reduce backend server load. Once created, images are stored as regular files on a Frisbee server and can be transferred over *scp* or NFS. However, since their application scenario is LAN based, they are able to ensure the scalability of the transfer using an UDP on IP-multicast protocol. Image installation is very similar to image creation, except for the fact that after decompression only selective blocks on the file-system may be written. The authors observe that this might result in data of previous users being leaked to current users and hence provide an alternate option to zero out free blocks in the file-system.

With the exception of Frisbee, most of the systems discussed above have a snapshot feature to facilitate online backups. Most of them categorize snapshots as read-only volumes which will be removed from the system once a backup on tertiary storage has been taken. In other words, these systems handle snapshots as a special case. A growing class of industrial and academic volume managers seem to blur the line between a snapshot and a volume. These systems treat both snapshots and regular volumes as persistent virtual disks that are allocated storage on demand. This technology known as thin provisioning allocates storage to an application only when it writes to the file-system. Proponents of thin provisionining[10][15] claim that the conventional "Dedicate-on-Allocation" philosophy forces customers to buy storage upfront and are wasteful in terms of both maintenance and energy consumption. Snapshots based on thin provisioning are typically used for debugging, testing and intrusion detection scenarios where a modifiable image of a file-system can be quite useful.

The 3PAR thin provisioning model partitions physical storage in to 256MB chunks. These chunks are grouped together to make Logical Disks(LDs) with specific RAID characteristics. LDs are organized into groups and then space is "Dedicated-on-Write" to a thin provisioned volume. Note that a thin provisioned volume can have an arbitrary size. However, storage is only allocated when the application really needs it. If storage within the group is exhausted then more LDs can be added on-demand. 3PAR's Virtual Copy software takes this even further by allowing the creation of writable snapshots of a virtual volume. Just like regular volumes, snapshots can grow on-demand and can be recursively snapshotted.

The 3PAR architects realize that at some point an application may not get its virtually guaranteed storage. In order to handle such situations they have implemented multiple categories of alerts to notify the system administrators. As an example, a user may specify the logical capacity threshold for a volume at the time of its creation. System administrators are notified when a volume's actual physical consumption is close to its logical threshold. This also protects against a "run away" application which is in an abnormal state and is continuously writing data to the storage device. Another type of alert can be set on the amount of physical capacity used across the volume management system. If the effective physical capacity gets exhausted then new write requests are failed until additional storage becomes available.

The Parallax[28] system is built along the same lines. Parallax is a distributed storage manager designed to support millions of Xen[3] based virtual machines running in a cluster environment. Parallax runs as a user space device driver in an isolated Xen domain, thus providing security and commendable reductions in programming effort. Virtual machines, using Parallax as a backend, partition their local storage resources into two parts. One of these is used as a persistent cache for locally hosted virtual machines and the other is a contribution to a pool of distributed storage shared by its peers in the cluster. The shared pool is used for purposes of data redundancy and availability. Parallax provides each virtual machine with a virtual disk abstraction and transparently converts logical block requests into cluster wide physical block addresses.

The primary motivation behind Parallax was to provide the ability to frequently snapshot a virtual machine's disk while keeping the infrastructure costs much lower than commercial SAN products. Parallax snapshots are designed to serve two distinct scenarios. The first scenario is one in which frequent checkpointing of system state is required for debugging purposes. Intrusion detection and sandboxing environments fall in to this category.

Secondly, snapshots can be used as backend file systems for newly instantiated virtual machines. Parallax designers realized this dual design goal by using a hierarchical tree data structure known as the radix tree. A radix tree allows for a very simple snapshot construction process and also facilitates the on-demand growth of the snapshot. In contrast to Petal their scheme is much simpler, efficient and does not require distributed locking or lease-based persistent caching techniques.

2.2 File-Systems

Unlike volume managers, file-systems do not have an explicit snapshot mechanism. As an alternative, some file-systems provide fine-grained mechanisms for versioning files. Although a complete file-system agnostic, file level, snapshot can be obtained by using the dump, cpio or tar utilities, timing constraints remain a hindrance. For example, in a scenario where files are frequently changing, the dump utility might fail to capture a file's instantaneous state of interest. Whereas file-system versioning has been explored extensively in Elephant[23] and the more recent Ext3Cow[19] project, there happen to be a few intriguing examples which provide entire file-system snapshots.

One of the earlier file-systems to do so was developed for the Plan 9 operating system. This file-system known as the Cached WORM file-system[20] used a write-once-read-many(WORM) optical disk to store read-only file-system snapshots. These disks were popular backup media which had a price-per-bit similar to magnetic tapes. This provided users with an opportunity to couple their secondary and tertiary storage. The benefits were that remote tapes were no longer required and the availability of local snapshot images allowed for greater sharing at block-level. For operational purposes, the system required that the logical address space of the file-system be much smaller than the WORM disk's physical address space.

The Plan 9 system operated by maintaining a normal magnetic disk as a cache for the WORM device. This had a dual purpose: firstly, it allowed for a data block to be written several times before it was actually committed to the optical store and secondly, the cache served to hide the latency inherent in optical devices. Snapshots were created by executing the dump utility, as a result of which the file-system would pause all future I/O activity until the dump is completed. The dump operation involved flushing all dirty blocks from the magnetic disk cache to the WORM device. The file-system maintained the state of each block in the cache which eventually facilitated the process of identifying dirty blocks during the dump. Once a block was committed to the optical store, its state was changed to read-only and any future write attempts were served by allocating a new block. The entire dump activity could take up to 10 seconds and the author does not recommends its use during peak hours.

In a similar vein, NetApp's WAFL(Write Anywhere File Layout)[9] file-system provides both logical and physical backup[11] utilities. The logical backup utility is similar to the conventional dump command with the exception that it is built in to the kernel. The kernel level dump is optimized to take advantage of internal file-system data structures and backup oriented read-ahead policies. WAFL's physical backup utility is based on snapshotting the file-system at regular intervals. These intervals can be defined by the user with the restriction that only 20 snapshots can be maintained at any given instant. Like most backup systems, WAFL's snapshots are read-only.

On a slightly different note, the XenFS project aims to provide file-system sharing between multiple domains running on top of Xen. Although the project is still in its early stages, the proposal is to implement a NFS like file-system for guest domains. The file-system is not true NFS and is designed to work on a single physical machine only. Since guest domains execute in a tightly coupled virtualized environment, there are several optimizations that can be incorporated into the generic NFS design. One such optimization is the use of a universal interdomain page cache. This would serve the usual goals of reducing disk I/O and reducing memory footprint. Applications using XenFS will be able to take advantage of interdomain memory sharing using the standard mmap API. Additionally, XenFS allows copy-on-write file-system functionality enabling multiple domains to share a common base file-system. All of this will be supported by CoW mechanisms at the memory level.

2.3 Contributions

Volume snapshots in themselves are a classic topic. In the past snapshots were perceived as read-only ephemeral volumes maintained solely to facilitate online backups and therefore not much attention was paid to their maintenance and access costs. However, contemporary snapshot utilities are targetted more towards an expedited clone construction mechanism where each clone of some base image might get exposed to any general purpose I/O operations. This shift in requirements demands a more sophisticated maintainance process and some predictability in I/O performance. RSnap is an affordable and versatile volume manager which meets these requirements while maintaining efficiency in data access. In the following paragraphs we briefly highlight the similarities and differences between RSnap and previously cited systems and conclude by summarizing our specific contributions.

Unlike conventional volume managers, RSnap allows volume snapshots to be mutable. Additionally, it allows the creation of snapshots of snapshots, which in turn may be mutable. RSnap's kernel level implementation facilitates performance and scalability, while avoiding expensive context switching and redundant copying inherent in typical user space drivers. Further, RSnap's implementation extensively benefits from existing kernel infrastructure such as the page cache and the journalling layer. Unlike Peabody, RSnap does not provide continuous preservation of a file-system's state. Like Clotho, RSnap allows a user to configure the frequency of snapshots.

When used in conjunction with Xen, RSnap can be hosted in domain 0 and serve as a backend storage manager for other guest domains. This is similar in effect to using XenFS except that instead of a file-system interface, RSnap provides the illusion of a physical device which can be imported by a guest domain. Further, RSnap's smart caching strategies expedite the process of logical to physical address translation and can potentially avoid duplicate read requests from peer domains.

RSnap belongs to the class of systems which observe the on-demand allocation or "Dedicate-on-Write" philosophy. To this extent RSnap allows the creation of both static and sparse volumes. RSnap snapshots are not ephemeral and their lifetime can exceed that of the parent they were based upon. In order to keep snapshots light-weight and growable, RSnap uses Parallax's radix tree data structure. We realize that 3PAR and other vendors provide commercial products with similar characteristics but we believe that their cost/benefit ratio is best realized in environments requiring mass scale storage management based on SANs, whereas, RSnap shares the spirit of FAB and Parallax in being a common man's filer.

RSnap's specific contributions can be summarized as follows:

1. We improve upon LVM2's current snapshot functionality by allowing a user to create snapshots of snapshots. Previously such functionality was only available in commercial systems, whereas RSnap is envisioned to become an open source filer.

2. Like Parallax, we use the radix tree data structure to provide light-weight mutable snapshots. However, due to our kernel space implementation we are able to utilize a trusted and well scrutinized code

base for caching and operational consistency purposes and are therefore able to present a much simpler implementation.

3. We present evaluations to show that our journalled metadata approach outperforms LVM2's copy-on-write implementation by a factor of 6 to 12 under heavy workloads and performs up to 55% better for realistic usage scenarios.

4. Lastly, we motivate the need to reclaim unreferenced blocks out of snapshotted volumes without reclaiming their dependent children and present the design, implementation and evaluation of a suitable garbage collector. Our garbage collector checkpoints it state to survive failures and can run in parallel to regular I/O operations on the system.

Chapter 3

System Design

This chapter provides a detailed description of RSnap's design. We begin by enumerating the design goals and then describe salient aspects of the design together with a discussion of the motivation behind our choices.

3.1 Design Goals

Our primary motivation behind this work was to develop a high performance, functionally rich block layer virtualization infrastructure that allows a user to take recursive writable snapshots of logical volumes. The semantics of recursive writable snapshots can be observed from Figure 3.1. Here, Volume A is the first volume that was created in the system. The arrows indicate the dependency of a snapshot on its parent. Volumes B, C and D are writable snapshots of Volume A. Volumes E and F are writable snapshots of Volume B. Similarly, Volumes G and H are writable snapshots of volume D. Note that Volumes A, B and D are frozen images and cannot be modified anymore. One can use them only to create more writable snapshots. The system does allow them to be mounted as read-only.

In some sense, the semantics of RSnap can be described as those of a volume versioning system. Typically, versioning systems allow modifications to the terminal nodes in a branch only, whereas RSnap can be perceived as a conscious effort to allow arbitrary branching within the version tree of a volume. Unlike some versioning systems, RSnap does not automatically preserve all possible versions of a volume. A user has to explicitly take a snapshot of some live device in order to create a point in time checkpoint. We have strived to keep the snapshot creation process simple and efficient so that a user may even automate it by creating a cron job.

At a higher level, RSnap can also be thought of as a file-system for maintaining volumes. Each volume in RSnap can be considered as a file. The distinction is that statically created volumes have a fixed size, whereas, snapshots acquire data blocks on demand. Several issues which came up during the development of RSnap can be observed in any regular file-system.

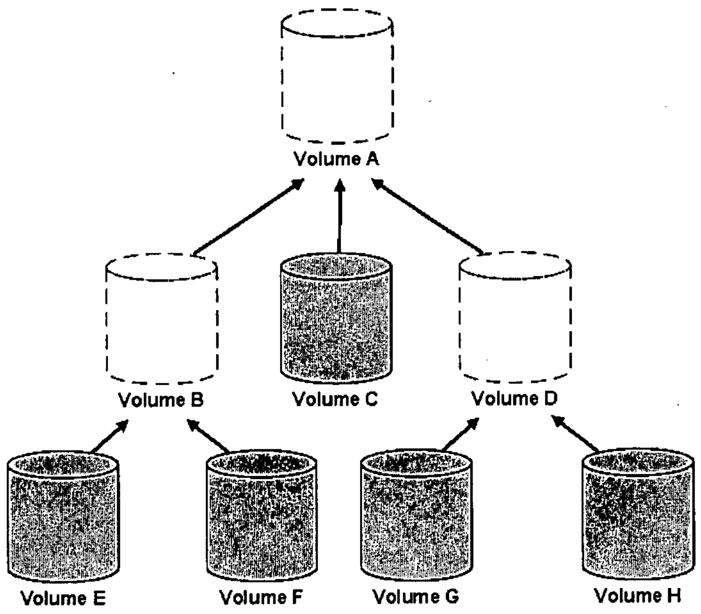


Figure 3.1: Family tree for a set of volumes

Another distinguishing feature of RSnap is that it allows deletion of any arbitrary volume from the system. Storage allocated to a leaf in a family tree of volumes can be immediately reclaimed, whereas, storage allocated to internal volumes is lazily reclaimed by a background garbage collector thread. Thus, in Figure 3.1, any of volumes C, E, F, G and H can be immediately reclaimed, whereas, the reclamation of A, B and D can be automated with assistance from the garbage collector.

From the above discussion, RSnap's design goals can be summarized as follows:

- Provide a simple interface for system administration.
- Provide light-weight mutable snapshots.
- Allow recursive snapshots.
- Allocate storage to snapshots in a "Dedicate-on-write" fashion.
- Allow reclamation of any arbitrary volume in the system.
- Maintain efficiency in data and metadata access.

In the next section we discuss different aspects of RSnap's design which enable us to meet the above motioned objectives. In particular we focus on the following:

1. Flexibility and Transparency: In this section, we describe RSnap's simple interface and its position within the kernel stack.
2. Volume Metadata: In this section, we discuss RSnap's implementation of the radix tree data structure and show how it can be used to provide light-weight recursive snapshots.
3. Consistency Maintenance: In this section, we describe how we provide consistency guarantees on RSnap's metadata.
4. Block Reclamation: In this section we describe our block reclamation algorithms.

3.2 Design Aspects

3.2.1 Flexibility and Transparency

The two entities of interest in RSnap are mutable volumes and read-only snapshots. Read-only Snapshots can be created out of mutable volumes and mutable volumes can be created out of Read-only snapshots. Each of these entities is a part of a larger entity called the blockstore which is a large collection of disk blocks. Each volume (or snapshot) within the blockstore is identified by a unique user defined 64-bit number.

RSnap uses the Linux device-mapper interface for user interaction. The device mapper interface provides primitives for device creation, deletion and messaging. Messaging is a primitive for implementing arbitrary device functionality and is a direct analogue of the ioctl() system call. RSnap extends these primitives in order to provide the following set of administrative operations:

- `Create()` allows a user to construct an arbitrary sized volume within a blockstore. The process of volume creation involves recording of the volume's metadata to the blockstore.
- `CreateSnapshot-RO()` creates an instantaneous frozen image or a read-only snapshot of a live device. Future writes to a snapshotted device will result in the invocation of the copy of write process in order to preserve the snapshot. A read-only snapshot must be created before a user can create mutable snapshots.

- `CreateSnapshot-RW()` allows a user to create a new volume out of a read-only snapshot. Essentially the new volume is a clone of the read-only snapshot and is expected to diverge from it as writes are performed on it.

- `Remove()` allows a user to temporarily disable the volume by removing its corresponding device file from the file-system. The volume lives passively in the blockstore and can be reinstated whenever necessary.

- `Reinstate()` allows a user to instantiate a volume residing dormantly in the blockstore. This involves the creation of a device file corresponding to the volume. The operations `Create()` and `CreateSnapshot-RW()` result in the automatic creation of the device file, whereas, `CreateSnapshot-RO()` results in the creation of a dormant read-only snapshot. The `Reinstate()` operation facilitates the mounts of these read-only snapshot volumes. Further, a reinstate is also useful in selectively instantiating volumes after a system reboot.

- `Reclaim()` facilitates a user in reclaiming the space occupied by a live volume or a snapshot in the blockstore. A volume must be live during the reclamation process and is removed completely from the system after a subsequent `Remove()` operation.

- `Scan()` provides a user with internal statistics on the volume. These statistics guide the user in volume maintenance. Typical statistics are the actual number of blocks occupied by the volume in the blockstore, the ancestor of the volume in its version tree, volume labels, creation times, etc.

The above set of operations has been modeled after LVM2's primitives and are by no means exhaustive but provide a concrete base for implementing a wide range of administrative policies. Since, RSnap tracks individual volumes through a 64-bit id, it can be quite strenuous for a user to remember the identifiers for a large set of volumes. We overcome this by maintaining a configuration file that maps volume names to its identifiers within the blockstore. This scheme is implemented entirely in user space and is currently being used in LVM2 as well.

Implementing RSnap at the block layer was a natural choice. Due to this design decision, RSnap can be inserted arbitrarily in a system's layered block I/O hierarchy. This stackable driver concept can be observed in the design of other block I/O abstractions, such as software RAID systems and various volume managers[17][4]. RSnap can transparently accept I/O requests from a file-system as well as a similar block layer virtualization system. Similarly, RSnap can forward I/O requests to an actual device driver or another block I/O system.

At the time of their creation, volumes that are created through the operation `CreateSnapshot-RW()` are sparse. Although the file-system residing on top of the volume believes that its size is equal to the size of its base image, in reality, there might be no physical blocks allocated to the volume. Later, as write operations are made to the volume, new blocks are dynamically allocated from the blockstore. Obviously, for such a system to work the size of the blockstore should be much larger that the size of any volume hosted in it. Ideally, the blockstore should be extensible so that when all blocks in the blockstore are exhausted, new disks can be transparently added to create more space. As of now, RSnap does not support the extensibility of the blockstore but its simple design can accommodate that feature with minimal programmer effort.

3.2.2 Volume Metadata

While designing any volume management system, it is imperative to think about a good indexing scheme. The choice of an indexing structure influences various design aspects such as address translation, block allocation, read/write performance, block reclamation and even snapshotting mechanisms. In order to settle upon a suitable indexing structure, we studied a variety of storage management schemes. In general, the data structures used by these systems are either linear or tree based. We discovered that linear structures are not ideal for systems exhibiting recursive properties. The problem is that whenever an attempt is made to deploy them in a recursive scenario they tend to degenerate into trees.

RSnap uses the radix tree data structure. Figure 3.2 shows an example radix tree for a volume and its snapshot (Note that this example was first presented in the Parallax[28] work). The radix tree has been created for a hypothetical 6-bit address space. Solid arrows from one radix tree node to another represent writable links, whereas, read-only links are represented by dotted arrows. Whenever a user creates a volume, its corresponding radix tree is statically created in the blockstore. Each node of the radix tree contains writable pointers only. The radix tree acts as the metadata for the volume and is used to translate logical block numbers in to physical blocks of the volume. On the other hand snapshots are created by merely replicating the root of its parent's radix tree and marking all its pointers as read-only.

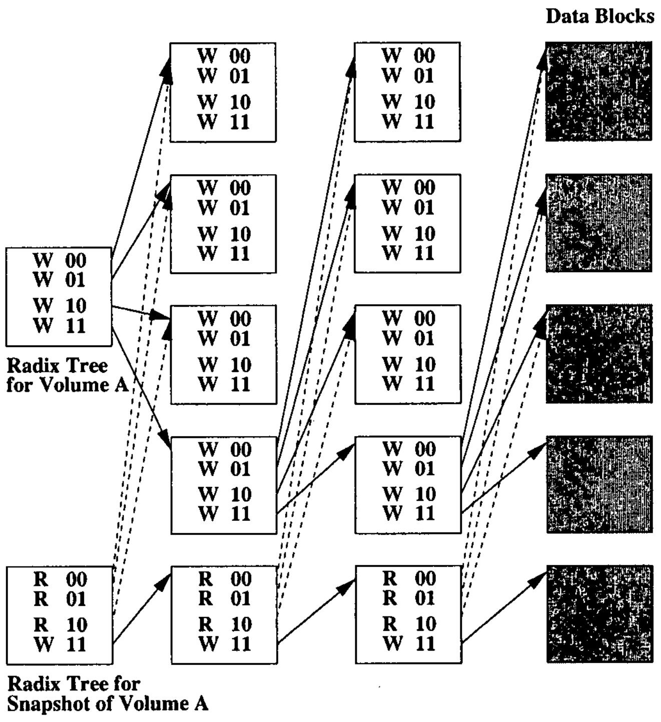


Figure 3.2: Radix Tree for a Volume and its Snapshot.

Further, a snapshot's metadata is dynamically allocated in a "Dedicate-on-write" fashion. As shown in Figure 3.2, a new branch was allocated to the snapshot after a write was made to its logical address 0x111111. The one concern with a radix tree based scheme is that address translation requires a traversal through multiple levels of indirection. However, we have observed that this problem can be easily mitigated with ample buffering.

RSnap partitions both the data and metadata of a volume into fixed sized blocks. Each data or metadata block is equal in size to the blockstore's universal block size. The blockstore's universal block size can be any of 1K, 2K or 4K. The motivation behind restricting the block size to the above mentioned values is in utilizing the buffer cache for caching metadata blocks during the logical to physical address translation process. Thus, the metadata overhead is in not just extra storage but it also consumes main memory during system execution. Our measurements show that the overhead of maintaining RSnap's metadata is reasonable for even very strenuous workloads(refer to section 5.4).

The size of the metadata for a volume depends upon its total size, the blockstore's block size and the volume's type. The metadata is persistent and shares space with the data blocks of the volume within the blockstore. For a newly created 100GB volume, the metadata is around 100MB which is about 0.1% of the total size of the volume. The size of the metadata for a volume created out of a read-only snapshot depends upon the number of copy-on-write operations performed upon it.

3.2.3 Consistency Maintenance

RSnap's metadata is organized into simple but crash vulnerable data structures. Given the unpredictability of system failures, a crash might occur during volume creation, volume snapshot, volume reclamation or while a copy-on-write operation is in progress. These are times when RSnap is actively updating metadata. In order to maintain good performance RSnap does not perform synchronous updates of metadata; metadata is first updated in the page cache and corresponding buffers are left marked dirty. It is expected that the operating system will later flush the dirty buffers to disk. Thus, if a failure happens before the metadata is updated to disk, RSnap's data structures might become corrupted. Not only does this have the potential to make a volume's file-system inconsistent but it might also make a blockstore's block appear as falsely allocated.

Naturally, recovery after a crash in RSnap will involve a two phase process. In the first phase, RSnap's metadata structures will need to be restored to a consistent state, and in the second phase any file-system specific recovery might have to be done. We considered multiple ways of ensuring RSnap's metadata consistency. Most noticeable were the BSD Soft Updates[5], and journaling[6].

Soft Updates attack the metadata update problem by guaranteeing that blocks are written to disk in their required order without using synchronous disk I/Os. In general, a Soft Updates system must maintain *dependency* information, or detailed information about the relationship between cached pieces of data. For example, when a file is created, the system must ensure that the new inode is written to disk before the directory that references it is. Such tracking of multiple dependencies at the block level can frequently lead to *cyclic dependencies*. Soft Updates resolve cyclic dependencies by doing a roll back on the data buffer to a previously consistent state before writing it to disk and then rolling it forward to its current value after the write has completed. Thus, with Soft Updates, applications always see the most recent copies of metadata blocks and the disk always sees copies that are consistent with its other contents.

Journaling file-systems solve the metadata-update problem by maintaining an auxiliary log that records all metadata before it is actually written to its location on disk. Usually, journaling consists of two phases: the commit phase and the checkpoint phase. In the commit phase all dirty buffers are flushed to the log and in the checkpoint phase they are written to their actual location on disk. In case of a crash, committed transactions from the log can be replayed to restore system consistency. Journaling is available in both synchronous and asynchronous flavours.

Seltzer et al.[24] have performed a very comprehensive comparison between journaling and soft updates. Their first observation is that both mechanisms perform acceptably only when higher layers do not require synchronous metadata updates. They demonstrate that the overall performance of both techniques is comparable in most cases. In some cases, Soft Updates exhibit certain side effects that result in a higher throughput whereas in others these very side effects can lower system performance. Citing the example of Log-structured file systems[21] they interestingly conclude that in certain cases a combination of the two techniques is required.

The Linux community continues to remain divided over the issue of Soft Updates vs. Journaling. The one feature that seems to make journaling more popular is its ability to avoid an `fsck()` operation. Although, Soft Updates allow a file-system to be mounted immediately after a crash, it requires that `fsck()` be run in the background for restoring file-system consistency. A secondary reason hindering the adoption of Soft Updates is the complexity associated with its implementation.

The Linux Journaling implementation is quite generic and exposes a set of APIs that can be used within any file-system. The code is optimized, reliable and is currently an integral part of the Ext3 file system. For these reasons, we chose to use the Linux journaling layer to provide consistency for RSnap's metadata structures. By default, RSnap reserves about 1% of the total space available in the blockstore to maintain the journal. For larger blockstores, a user can configure the journal size to be smaller. The journal tracks metadata updates for all of RSnap's operations such as volume creation, reclamation, copy-on-write etc. If a crash happens during system operation, the journal is transparently replayed upon the next creation or instantiation of a volume.

Journalling is particularly interesting when a copy-on-write operation is in progress. This is because the new version of the data block must be written before we can commit its associated metadata. In this context, RSnap allows a user to configure it in either writeback or ordered mode. In ordered mode the data block will be written to disk before jbd can commit metadata to the journal.

3.2.4 Block Reclamation

Motivation

A user can create free space in the blockstore by reclaiming blocks allocated to a volume. In RSnap it is possible to delete any volume at will. The motivation behind deleting leaf volumes is quite obvious but the motivation behind deleting non-leaf volumes is a little subtle. A non-leaf volume can be profitably deleted when its current snapshots have remapped a large number of its blocks. This means that most of the read only pointers in the children have become writable and do not refer to the parent's data blocks anymore. A second reason to delete a non-leaf volume arises when it has remapped very few data blocks of its parent. This means that it did not perform a large number of copy-on-writes before it was snapshotted and its presence is merely ornamental in the family. Note that once a non-leaf volume has been deleted it cannot be used to create new writable snapshots.

When a leaf-volume is to be deleted then all one has to do is traverse its writable pointers to reclaim the space. However, deleting non-leaf volumes requires a more involved mechanism. The problem is that at any given time it is hard to establish which blocks are being used by any of its descendants. This is because when snapshots are taken, only the root of the radix tree is replicated and since internal nodes and leaves do not have back pointers to their parents, it is difficult to determine the exact number of metadata blocks that point to them. Intuitively, this appears to be a garbage collection problem.

Garbage Collection Approach

Most garbage collection problems can be solved by using either reference counting or a mark and sweep approach[30]. Garbage collectors based on reference counting have the advantage that they do not require multiple traversals over the data. Unreachable blocks can be immediately reclaimed without an explicit collection phase. Based on this observation, it was quite tempting to adopt a per-block reference counting approach for RSnap. However, we realized that maintaining persistent reference counts in a system can be quite expensive in terms of both space and increased disk I/O. Further, reference counting structures have to be fault tolerant and require reconstruction as part of system recovery[1]. On the other hand, mark and sweep algorithms keep very little metadata for tracking data liveness and do not entail transactional support. Therefore, RSnap uses a mark and sweep garbage collector, which performs a periodic blockstore wide scan to identify blocks that can be reclaimed.

The algorithm behind our garbage collector is very simple. When a non-leaf volume is reclaimed, all its writable blocks are marked as tentatively deleted but reachable. The garbage collector operates in four phases:

1. In the first phase it marks all tentatively deleted blocks as unreachable.
2. In the second phase it reads all metadata blocks which have read-only pointers in them. This includes the root blocks of radix trees belonging to snapshots. It then interprets the read-only pointers in these blocks to identify tentatively deleted blocks which are reachable. Note that in this phase tentatively deleted blocks are not read and hence if a tentatively deleted block is reachable from another tentatively deleted block only, then it will not be marked as reachable.
3. In this phase reachable tentatively deleted metadata blocks are read and then their pointers are traversed to identify those tentatively deleted blocks which may be reachable only from them. This operation may have to be repeated several times corresponding to the height of the tallest radix tree present in the blockstore.
4. In this phase tentatively deleted blocks which are not reachable are reclaimed.

To ensure the garbage collector's progress in the event of a system crash, we journal every garbage collection operation along with a progress indicator in a single transaction. Progress is defined by the garbage collector's current phase number and the last block number it diagnosed. Intuitively, it is hard to continue with the garbage collector while normal I/O operations are in progress. This is mainly because normal metadata updates might violate the garbage collector's invariants of block reachability. In the following paragraphs we discuss such metadata updates and show that by using appropriate journalling, the above algorithm can be run in parallel to any normal I/O operation on the blockstore.

The first operation of interest is the copy-on-write on the radix tree. During a copy-on-write operation, read-only pointers in metadata blocks are replaced with writable pointers. If the garbage collector is in any of

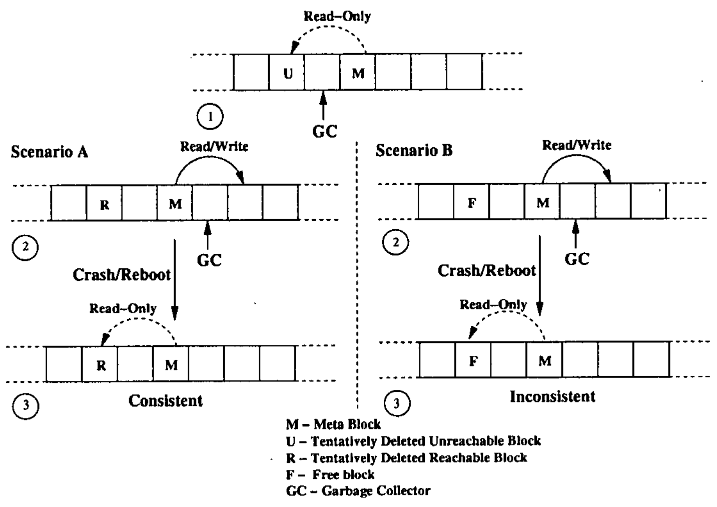


Figure 3.3: Consistency Analysis during simultaneous execution of Copy-on-Write and Garbage Collection.

phases 1, 3 or 4 then its invariant is unaffected. If the garbage collector is in phase 2 then there are two possible scenarios (refer to Figure 3.3):

In scenario A, the garbage collector read the read-only pointer in the metadata block before it was replaced with a writable pointer. Therefore, the tentatively deleted but unreachable block was marked as reachable. Now, if a crash happens and the update to the metadata block could not be committed to disk then the read-only pointer will persist upon reboot. This is not a problem because the block pointed to was already marked as reachable and therefore was not reclaimed by the garbage collector.

On the other hand, in Scenario B the garbage collector read the metadata block after the copy-on-write operation was performed and could therefore mark the unreachable block as free. Now, if the read-only pointer persists across a crash and reboot then it might end up referencing a freed block.

We discovered that a copy-on-write operation on a metadata block can safely continue even if it has not been diagnosed in phase 2. This is because we journal both the copy-on-write and garbage collection operations. If a copy-on-write happens before the garbage collector's scan then the changes to the metadata caused by the copy-on-write operation will be packed in a transaction preceeding the one containing decisions made by the garbage collector. Therefore, if the copy-on-write operation does not make it to the disk then the garbage collection updates will be cancelled automatically. Orthogonally, if the copy-on-write was done after the garbage collector's scan then a read-only block may be incorrectly marked as reachable and will not be reclaimed until the next garbage collection cycle.

The next operation of interest is volume delete. When a leaf volume is deleted all its writable blocks are marked as reclaimed. This does not affect the garbage collector's operations in phase 1, 3 or 4. However, if the garbage collector was in phase 2 then, once again, there are two possibile scenarios(refer to Figure 3.4):

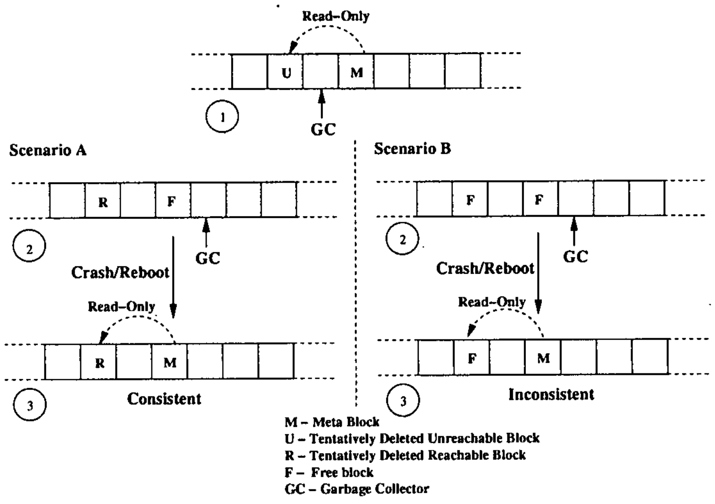


Figure 3.4: Consistency Analysis during simultaneous execution of leaf Volume Delete and Garbage Collection.

In scenario A, the garbage collector read the read the read-only pointer before the metadata block containing it was marked as free. Hence, upon a subsequent crash and reboot even if the freeing of the metadata block could not be committed to disk then we do not run into an inconsistent state.

In scenario B, the metadata block was marked as free before the garbage collector could diagnose it in phase 2 and therefore it is possible to reclaim the tentatively deleted unreachable block even before the freeing operation is committed to disk. Hence, upon reboot it is possible that the metadata block might be referencing a free block.

Once again the journalling layer comes to our rescue. Metadata updates describing reclamation of writable blocks will be in the same or successive transaction as the garbage collection updates made by either reading or ignoring these blocks in phase 2. Thus all updates will hit the disk in proper order. In case none of them gets committed then the garbage collector can restart from its last checkpointed progress indicator.

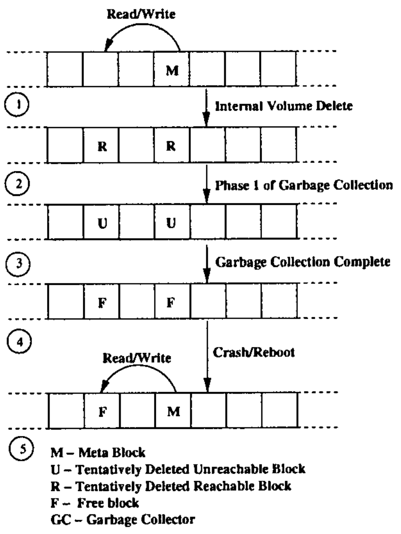

Read/Write
M
1
Internal Volume Delete
R
R
2
Phase 1 of Garbage Collection
U
U
3
Garbage Collection Complete
F
F
4
Read/Write
Crash/Reboot
F
M
5
M – Meta Block
U – Tentatively Deleted Unreachable Block
R – Tentatively Deleted Reachable Block
F – Free block
GC – Garbage Collector

Figure 3.5: Consistency Analysis during simultaneous execution of internal Volume Delete and Garbage Collection.

Journalling protects us even if an internal volume is deleted during any of the phases of garbage collection. When an internal volume is deleted then all its writable blocks are marked as tentatively deleted but reachable. This can cause problems if the garbage collector is in phase 1 and marks any of these blocks as unreachable. Figure 3.5 depicts a scenario in which an inconsistent system state might prevail. As shown in the figure, after phase 1 of garbage collection the two blocks belonging to the internal volume are marked as unreachable. If there are no other references to these blocks then the garbage collector might mark them as free upon completion. Now, if a crash occurs and the free state of one of these blocks could not be committed to disk then upon reboot it might reference a freed block.

We can avoid this conflict by journalling the updates caused by the delete operation and the garbage collector. In this way, if the delete happened before the invocation of the garbage collection then transactions corresponding to the delete operation will preceed those corresponding to the garbage collection cycle.

On the other hand, if the garbage collector had already completed phase 1 when the delete happened then the tentatively deleted metadata block will be used in phase 3 to mark other tentatively deleted blocks it can reach.

All other operations such as static volume creation, snapshot creation and reinstating volumes do not affect the garbage collector's invariants. The observation to be made here is that garbage collection updates will either be in the previous, same or next journalling transaction which includes regular volume metadata operations. This preserves the garbage collection invariant at all times.

Suitability and Alternatives

Before settling upon an algorithm for garbage collection we explored a few other alternatives. Although most of them were rejected due to their associated costs we list them here for the sake of completeness.

One of the first ideas was to compare a parent's radix tree to all its children and then reclaim blocks which have been remapped by all children. Additionally, if a child volume references a block then we could grant it exclusive ownership of that block. Permanent ownership can be granted if no other child is referencing that block and temporary ownership will be granted if other children reference that block as well. The problem with this approach is that an address based tree comparison can be quite expensive in terms of disk seeks. Secondly, if we were to assign temporary ownership of a block to a child then as soon as it performs a copy-on-write operation on that block, we will have to transfer the ownership to one of its sibling or one of its children. This can significantly slow down copy-on-write operations.

Our second idea was to maintain a reference count per block in the blockstore. Thus, various operations such as volume creation,deletion and copy-on-write would result in the increment and decrement of reference counts associated with the blocks they manipulate. We soon realized that given the structure of the radix tree and the kind of operations we intend to perform on it, maintaining per-block reference counts is not a wise idea. As an example consider a radix tree with a 4K node size. This node can contain 512 block addresses. Thus, whenever a new radix tree node is allocated or deallocated one must update a minimum of 512 reference counts.

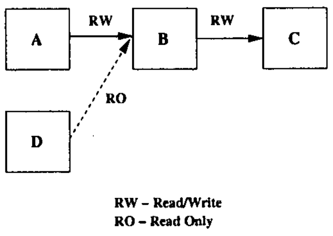


Figure 3.6: Node sharing between two radix trees.

The problem becomes worse when we begin to share radix tree nodes between two volumes. In Figure 3.6 we instantiated a new radix tree root D by replicating an existing root A. In this case the reference counts for both nodes B and C would have to be incremented. This means that whenever we copy a radix tree node we must perform a depth first traversal to increment the reference counts for all nodes that form a part of its subtree. Clearly, this can be extremely expensive.

Once reference counting was out of consideration we began to explore potential mark and sweep algorithms. One of our initial approach was to first mark all tentatively deleted blocks as unreachable and then scan the metadata of all volumes searching for read-only pointers which can be used to identify reachable tentatively deleted blocks. Later, all tentatively deleted blocks which are not reachable can be reclaimed in a single scan. The problem with this approach is that it involves depth first scanning of all radix trees. We realized that this could lead to a very randomized read workload on the disk and therefore decided to go with our multistage approach instead.

Although the details of our garbage collector may appear to be involved, it has several interesting properties. One unique feature is that we do not read all metadata blocks. Since, we tag the blocks containing read-only pointers, we do not have to spend time diagnosing uninteresting metadata blocks. Secondly as compared to a reference count based scheme we require very little metadata. Thirdly, our phases have been designed to progress by linearly scanning the disk. It is possible that some phases might perform repetitive scans but overall the disk is never subjected to a burst of random activity. Finally, by designing the garbage collector as a background process modularized into multiple phases we can exert a much fine grained control over its execution. For example when the system is under heavy load we can lower its priority in favor of regular I/O operations.

3.3 Chapter Summary

In this chapter we presented RSnap's high level design objectives and have dicussed various decisions we took for their realization. In particular, we have provided details on our core metadata structure and how we guarantee its consistency while ensuring high performance. Additionally, we have presented the algorithm behind our garbage collector and have compared it to other alternative approaches.

Chapter 4

RSnap Implementation

This chapter presents interesting aspects of RSnap's implementation. We begin with a brief introduction to the Linux device-mapper and the journaling layer and describe how we utilize them to implement RSnap. Next, we illustrate how RSnap formats a device before use and then describe the implementation of our block allocator. We also provide implementation details behind operations such as volume creation, reclamation, snapshot creation and address translation and discuss how journaling affects block allocation/deallocation strategies during these operations.

4.1 Kernel Infrastructure

4.1.1 Linux device-mapper

The Linux device-mapper is a simple device driver that redirects I/O requests from virtual devices to actual physical devices. Clients of the device-mapper submit I/O requests in the form of a structure known as the *bio* vector, which contains an array of dirty pages that are to be written contiguously on a backend device. The *bio* vector also contains the first sector corresponding to the data and a pointer to the target device. When the device-mapper is in use both the sector number and the device are virtual. The basic task of the device-mapper is to replace these virtual entities with their physical counterparts and then forward the *bio* vector to a real driver's I/O queue. This virtual to physical transformation can be done in multiple ways and corresponding to each, there exists a separate target of the device-mapper. Example targets are linear, RAID, snapshot, encrypt, etc.

RSnap is implemented as a new target for the device-mapper. RSnap transforms a <virtual-device, logical sector number> pair into a <blockstore-device, physical sector number> pair. For performing the transformation it consults the radix tree of the virtual-device, which is persistently maintained within the blockstore.

4.1.2 Linux journaling layer

The Linux journaling layer (*jbd*) provides a set of APIs which can be used to initialize the journal and bundle various system operations into transactions. It is similar to the journaling implementations observed in regular databases but with one important distinction. Unlike databases, *jbd* does not allow users to create a new transaction for every operation. Instead, it automatically creates a new transaction when an existing one has become large enough. From a user's perspective an operation may comprise of multiple updates which must occur atomically and the *jbd* guarantees that all of them will be bundled in a single transaction that might include other operations as well.

Before starting an operation, a user calls `journal_start()` to specify the maximum number of buffers (also known as credits) that may have to be journalled in the running transaction. Before a buffer is modified, it must be first registered with the *jbd* through one of `journal_get_write_access()`, `journal_get_create_access()` or `journal_get_undo_access()` calls. A call to `journal_get_undo_access()` ensures that previous modifications to the buffer have been recorded in the journal. This is required during bitmap update operations to avoid irrecoverable deletes after a crash. After modification the user calls `journal_dirty_metadata()` or `journal_dirty_data()` to mark the buffer as dirty. After all modifications to the buffers are complete, a call to `journal_stop()` is made to mark the completion of the operation. The *jbd* guarantees that all modifications between a `journal_start()` and `journal_stop()` are included in the same transaction and will be written to their respective location on disk only after they have been committed to the journal.

4.2 Blockstore format

RSnap comes with a user space tool, *mkRSnap* which is used to format a physical device into a blockstore. *mkRSnap* accepts three parameters: a) blockstore's block size b) the path of the device to be formatted and c) the percentage space that is to be reserved for journaling. Based on these attributes *mkRSnap* creates the blockstore's superblock, zeroes out space for the journal and prepares a nibble map for all blocks on the device. The blockstore's superblock contains parameters such as `journal_start`, `journal_length` and `nibble_map_start` and `nibble_map_end`. These parameters guide the journaling layer and RSnap's block allocator respectively. Figure 4.1 shows a formatted blockstore as an example. Notice that the jour-

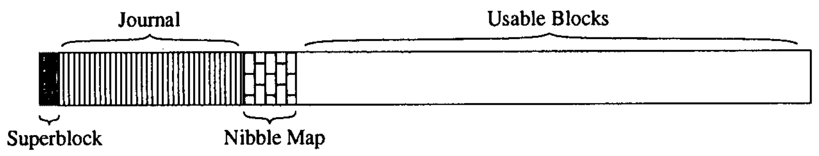


Figure 4.1: A formatted blockstore.

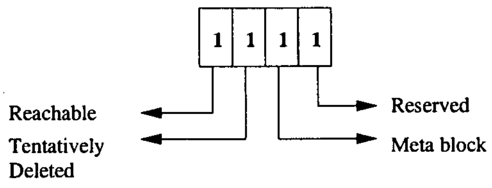


Figure 4.2: Individual bits of the nibble map.

nal and the nibble map are adjacent to each other. This reduces the number of seeks during post-crash journal replay.

As its name suggests, the nibble map contains a nibble(4-bits) for each block in the blockstore. This includes the blockstore's superblock, blocks comprising the journal and blocks which make up the nibble map. Figure 4.2 shows what each bit in the nibble stands for. Initially, *mkRSnap* marks nibbles corresponding to the superblock, journaling blocks and the nibble map as `Reserved`. Therefore, the block allocator ignores them during its regular operations. A zeroed nibble means that its corresponding block is free for use.

Usable blocks in the blockstore store all the data and metadata corresponding to volumes.

4.3 Block Allocator

RSnap's block allocator is quite simple. It consists of two basic functions `balloc()` and `bfree()`. As its name suggests `balloc()` is used to request a free block from the blockstore. The block allocator scans through the nibblemap in a circular fashion to locate a nibble that indicates a free block. It then uses a linear function to compute the sector address of the block corresponding to the nibble, which is then returned to the caller. Before returning the block to the user, it marks it as `Reserved` along with any other flags which the user may have specified. Similarly `bfree()` accepts a block number and a set of flags from the caller. If no flags have been specified then the nibble corresponding to the block number is zeroed out otherwise the flags are used to set a more accurate status of the block.

All changes to the nibble map are journalled using the *jbd* APIs. `balloc()` and `bfree()` assume that the caller must have already made a call to `journal_start()` with the requisite number of credits and therefore they call into the *jbd* only to express the intent to dirty appropriate buffers.

4.4 RSnap Operations

4.4.1 Volume Creation

RSnap users can create three types of volumes. The first type can be created through the `Create()` interface, which results in a static allocation of the volume's entire radix tree. This volume is new and does not share disk blocks with any other volume in the system. The second type of volume is a read-only snapshot of some base volume and is created through the `CreateSnapshot-RO()` interface. This volume cannot be modified but can be mounted as read-only. The third type of volume is created through the `CreateSnapshot-RW()` interface and is a mutable snapshot of some read-only snapshot volume. Note that the creation of the second and third types does not involve static allocation of complete radix tree structures. Instead merely replicating the root is enough for the volume creation operation.

In addition to initializing a volume's radix tree, the volume creation operation also allocates a superblock for the volume. This superblock contains parameters such as the volume's size, the height of its radix tree, the sector address of the block containing the root of its radix tree, the number of children sharing blocks with the volume and the volume's parent if any. All volume superblocks in the blockstore are tracked by a universal radix tree which is indexed on the 64-bit volume id. This tree is sparse initially and storage is allocated to it out of the usable space within the blockstore. During the `Reinstate()` operation, a volume's superblock is read and kept persistently in memory.

Static volume creation

During static volume creation, the blockstore's block size and the size of the volume are used to compute the height of a suitable radix tree to represent the volume. Each entry inside a radix tree node is a 64-bit block address in which one bit is reserved to mark readability or writability of blocks.

All addresses inside the radix tree nodes of a statically created volume are marked writable.

The tree creation algorithm works by allocating a range of block addresses for the volume at a time. In each iteration of the algorithm one complete leaf is allocated. This means that if the blockstore's block size is 4K then 512 data blocks are allocated to the volume in one iteration. This is done so as to minimize the number of transactions on the nibble map. We found that allocating one block address between a call to `journal_start()` and `journal_end()` is not very efficient, because the nibble map has to be committed to the disk more often. Therefore, we avoid that by allocating several blocks in between a call to `journal_start()` and `journal_end()`.

Read-Only Snapshot Creation

Before attempting to create a writable snapshot of some parent volume, a user must create its read-only snapshot. The advantage in doing so is that multiple writable snapshots can be created out of it in the future. The `CreateSnapshot-RO()` operation facilitates the creation of a read-only snapshot. The operation starts by detecting if the parent volume is active or not. If the parent volume is active then it is frozen and all its dirty buffers are flushed to disk using the Linux `freeze_bdev()` interface. Next, a superblock and a radix tree root are allocated for the new read-only snapshot volume. The root is filled with the contents of the current root of the radix tree belonging to the parent volume and then as a secondary step, all pointers in the current root are marked as read-only. Additionally, the parent field in the parent volume's superblock is set to the 64-bit id of the read-only snapshot. This ensures that the location of the root of the parent volume is not changed on disk. This entire operation (excluding the buffer flush) is journalled, so as to make it atomic. Beyond this point, the device corresponding to the parent volume is thawed using the Linux `thaw_bdev()` interface. Future writes on the parent volume trigger copy-on-write operations.

Writable Snapshot Creation

Writable snapshots are created out of read-only snapshots. The system first creates a new superblock and an empty radix tree root for the writable snapshot. The contents of the radix tree root of the read-only snapshot are read and converted into read-only pointers before copying them into the writable snapshot's root. Finally, the number of children of the read-only snapshot is incremented by 1 and the parent field of the writable snapshot's superblock is set to the 64-bit id of the read-only snapshot. Once, again the entire operation is journalled so that a crash does not result in erroneous counts or pointers.

4.4.2 Address Translation

Since RSnap provides virtual volumes, its clients operate on virtual block numbers. These virtual block numbers need to be translated into physical sector numbers within the blockstore device. The implementation of the translation mechanism is quite similar to the ext2/ext3 address translation mechanism. The difference is that instead of an inode we have a radix tree. In some sense the radix tree is very similar to an inode. The inode can be considered as a radix tree whose height increases with larger block addresses. It should be noted that the translation mechanism in both schemes is similar to virtual memory addressing techniques.

The bits in a virtual address are partitioned into multiple sets containing equal number of bits. The number of bits in each set corresponds to the number of sector addresses that can be accomodated in a block. For example, if the blockstore's block size is 4K, then 512, 64-bit addresses can be packed into one block. Thus the number of bits required to offset into a single block is 9 and therefore each set will contain 9 address bits. Thus the relationship between a block size and the number of bits used to offset within the block can be defined as follows:

$$NRBITS(block\ size) = log_2((block\ size)/8) \tag{4.1}$$

Also note that the value of `NRBITS()` along with the height of the tree determines the maximum virtual block address for a volume. In the above example, if the height of the tree is 3 then any virtual address for the volume can be atmost 27 bits long. In other words the largest virtual address will be $2^{27} - 1$.

Algorithm 4.4.1 depicts RSnap's basic method to compute a physical block address from a virtual block address. This algorithm is particularly used when a read operation is performed on an RSnap volume.

Algorithm 4.4.1: LOOKUP(*root*, *virtual address*, *height*, *block size*)

```
while ($height > 0$)
    do
        $height \leftarrow (height - 1)$
        $rootbh \leftarrow READ\_BLOCK(root)$
        $offset \leftarrow COMPUTE\_OFFSET(virtual\ address, height, block\ size)$
        $root \leftarrow rootbh[offset]$
return ($root$)
```

Starting from the root, each radix tree node is read through the function `READ_BLOCK()` and then the function `COMPUTE_OFFSET()` is used to compute the `offset` within that node. This operation is repeated until we reach a leaf of the radix tree and then the last *offset* is used to compute the physical address of the block. The `COMPUTE_OFFSET()` function is defined as follows:

$$COMPUTE\_OFFSET(virtual\ address, height, block\ size) = \\ ((virtual\ address) >> (height * NRBITS(block\ size)))\ \& \\ ((1 << NRBITS(block\ size)) - 1) \quad (4.2)$$

Depending upon the parameter `height`, the `COMPUTE_OFFSET()` function performs a right shift on the virtual address and then applies a bit mask to isolate the least significant `NRBITS()`. These least significant bits are then used as an offset into the radix tree node at level `height`.

Note that `READ_BLOCK()` uses the Linux `__bread()` interface to synchronously read blocks from the disk into the page cache. Therefore, future reads upon the same radix tree node can be served directly from the blockstore's page cache. Further, in the actual implementation, the buffer holding the node is kept locked while an address is being extracted from it. This fine grained locking allows multiple address translations to occur on different parts of the radix tree in parallel.

The translation process gets complicated during a write operation on a snapshot. This is because the write may result in the invocation of a copy-on-write operation which entails the use of journalling for maintaining consistency of the radix tree belonging to the volume. Algorithm 4.4.2

depicts RSnap's behaviour when a write operation is in progress.

Algorithm 4.4.2: LOOKUP_COW(*root, virtual address, height, block size*)

```
$chain \leftarrow \phi$
$currentbh \leftarrow READ\_BLOCK(root)$
while $(height > 0)$
  do
    $height \leftarrow (height - 1)$
    $offset \leftarrow COMPUTE\_OFFSET(virtual\ address, height, block\ size)$
    $nextroot \leftarrow currentbh[offset]$
    if $writable(nextroot)$
      then
        comment: A Copy – On – Write operation is NOT required.
        if $(height == 0)$ break
        $currentbh \leftarrow READ\_BLOCK(nextroot)$
      else
        comment: A Copy – On – Write operation IS required.
        if $(chain == \phi)$
          then
            $rsnap\_journal\_start(height)$
            $chain \leftarrow ALLOC\_CHAIN(height)$
            $ichain \leftarrow 0$
            $chain[ichain] \leftarrow currentbh$
        if $(height > 0)$
          then
            comment: Allocate writable metadata block.
            $newroot \leftarrow balloc(META)$
            $newrootbh \leftarrow GET\_BLOCK(newroot)$
            $readonlybh \leftarrow READ\_BLOCK(nextroot)$
            $COPY\_BLOCK(newrootbh, readonlybh, READ\_ONLY)$

            $currentbh[offset] \leftarrow newroot$

            $currentbh \leftarrow newrootbh$
            $ichain{+}{+}$
            $chain[ichain] \leftarrow currentbh$
          else
            comment: Allocate writable data block.
            $currentbh[offset] = balloc()$
$rsnap\_journal\_dirty\_metadata(chain, ichain)$
$rsnap\_journal\_stop()$
return $(currentbh[offset])$
```

During a write, the translation function verifies if the next radix tree node to be read is `writable`. If the node is read-only then the copy-on-write operation is invoked. The copy-on-write implementation is similar to splicing a new indirect branch to an inode. The algorithm begins with determining the height of the branch that is to be spliced into the current radix node. Based on that height, it invokes `rsnap_journal_start()` to reserve the required number of blocks in the journal. Later new metadata blocks are acquired through `balloc(META)`. The `META` flag is specified so as to assist the garbage collector.

Notice that whenever a new writable metadata block is acquired the contents of its corresponding read-only metadata block are copied into it. The `READ_ONLY` flag to the `COPY_BLOCK()` operation ensures that all copied pointers are marked as read-only. Also, the read-only pointer in the current buffer head(`currentbh`) is replaced with a pointer to the newly acquired metadata block. The algorithm concludes by acquiring a data block.

All modified buffers are tracked by using the `chain` data structure. Later, the `chain` is traversed in the function `rsnap_journal_dirty_metadata()` so as to convey the intention to the `jbd` layer.

Further, in the actual implementation, all nodes which are being modified are kept locked until the copy-on-write operation completes. This is to ensure that a parallel copy-on-write is not triggered on the same block address. However, I/O requests on other parts of the radix tree can proceed without any interference. A call to `rsnap_journal_stop()` is finally made to complete the copy-on-write process.

An issue we encountered during the implementation of RSnap's ordered mode was that jbd expects buffer heads to impose data-before-metadata ordering and unfortunately the linux block layer does not allow direct access to buffer heads belonging to data blocks. We solved this by introducing an atomic counter with every transaction. Before submitting an I/O request to the physical disk driver we increment the atomic counter and then decrement it in the asynchronous I/O notification received as part of request completion. We further modified jbd to wait until our atomic counter becomes 0 before committing any metadata blocks to the journal.

Algorithm 4.4.2 works correctly only if the blockstore block size is equal to the file-system block size. If the file-system block size is less than the blockstore's block size, then the copy-on-write process entails reading the read-only *data block* as well. This is because only a small part of the blockstore's data block is being modified. Hence, the new data block on the blockstore will partly contain data from the read-only block and partly from the I/O request. In such a scenario, we first read the read-only data block using the `__bread()` interface, so that future copy-on-write operations on the same data block can benefit from its presence in the page cache. Next, we allocate a new data block in which we first copy the read-only data block and then update it using the contents of the write request. Since, the write request should not wait any longer, we synchronously write out the new block to disk and upon its completion we call appropriate request completion functions by hand.

4.4.3 Volume Reclamation

As described in the design section, depending upon the type of the volume, reclamation can either be synchronous or will be done through a background garbage collector. In general when a volume is deleted, reclamation is done by traversing its radix tree in a depth-first manner. Further, we deallocate all data blocks in a leaf at once. Again, this is intended to reduce the number of nibble map commits to the journal.

If a leaf volume is being deleted then all nibbles corresponding to its writable blocks are zeroed out immediately, whereas, all blocks that appear as read-only in the radix tree are ignored. On the other hand if a read-only snapshot is being deleted then nibbles corresponding to its writable blocks are marked as Tentatively deleted but Reachable. Additionally, nibbles corresponding to its writable radix tree nodes are marked as Meta blocks.

1. The first phase of the garbage collector reads the blockstore's nibble map to identify Tentatively deleted but Reachable blocks and resets their Rechability bit.

2. The second phase of the garbage collector reads the nibble map to identify Meta blocks and then all such blocks are read and the nibbles corresponding to any read-only addresses they contain are set as Reachable. A small optimization we have done is to record at least one full page of Meta block addresses before proceeding to diagnose the addresses inside them.

3. In the third phase the garbage collector reads the nibble map again to identify Tentatively deleted, Meta blocks which are reachable. Nibbles corresponding to all block addresses inside such blocks are marked as Reachable. As previously mentioned, this phase may have to be repeated several times until no more blocks can be marked reachable. In order to ignore blocks in successive iterations of the third phase we reset the Reserved bit in their nibbles. The bit is set again while marking such blocks as reachable in phase 2.

4. The fourth phase simply reads the nibble map and zeroes out all nibbles which have their Tentatively deleted bit set and their Reachability bit cleared.

The progress of the garbage collector is recorded into the blockstore's superblock and is journalled along with each of its updates to the nibble map. This ensures that after a crash, the garbage collector does not have to repeat phases which are finished.

4.5 Chapter Summary

In this chapter we studied RSnap's implementation details. We studied how RSnap formats a disk before use and its simple round-robin styled block allocation policy. Further, we gained insight into the different types of volumes that can be created as part of RSnap's operations and the algorithms used during address translation. Finally, we studied the implementation details of RSnap's garbage collector.

Chapter 5

Performance Evaluation

This chapter presents details on RSnap's performance. Intuitively, the indirection hierarchy imposed by a radix tree should cause significant degradation in performance. Thus, our first set of experiments measure the performance degradation in using a RSnap virtual volume as compared to a raw disk. Since the blockstore block size can cause variance in metadata organization and caching overheads, our second set of experiments analyze RSnap's copy-on-write behavior with various combinations of blockstore block size and file-system block size. One of RSnap's primary purpose is to virtualize a physical disk into multiple volumes and therefore our third set of experiments study RSnap's performance when multiple snapshot volumes are being used in parallel. Since RSnap uses the page cache as a temporary storage for metadata, our fourth set of experiments measure the amount of page cache consumed during strenuous workloads. Lastly, we present some statistics on RSnap's garbage collector.

For each of our experiments, we use a 3.2GHz Pentium 4 machine with 1GB of RAM and a 80GB SATA disk configured to give about 70MB/s of raw I/O performance. For evaluation purposes, we use a modified version of Bonnie++ to generate a stress test workload and the Postmark[13] benchmark to simulate a more realistic usage scenario.

The Bonnie++ benchmark performs a series of tests which saturate the file-system's buffer cache by creating a contiguous file(s) which is double the size of the RAM. It then performs the following operations in sequence: character writes, block writes, block rewrites, character reads and block reads. For each of the tests involving a write a new file is created. In the process it attempts to measure the raw throughput of the underlying disk. We modified the benchmark to do block writes before character writes. This is because character writes tend to be more CPU intensive than block writes and we are interested in measuring the copy-on-write performance for pure I/O workloads. If the character write test is executed first then most of the copy-on-write is already done by the time block write tests are invoked. Thus in an attempt to measure the true impact of copy-on-write on performance we interchanged their sequence of execution.

On the other hand the Postmark benchmark is designed to create a large pool of continuously changing files and to measure transaction rates for a workload approximating an Internet electronic mail server. The file-system performance is measured by generating a random set of text files which vary in size and then performing a configurable number of transactions which involve operations such as file creation, deletion, read or append. In our test cases, Postmark was configured to work with a set of 20,000 files and 50,000 transactions. This results in approximately 200MB of data being written to the underlying disk.

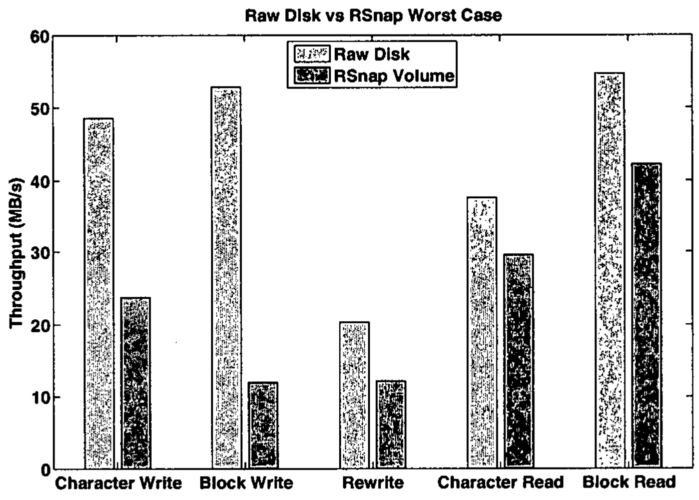
We further observed that RSnap's ordered mode and writeback mode give similar performance and therefore document results for ordered mode only. The similarity is because at the block layer data writes are not buffered and therefore most of the time metadata can be committed to disk without significant waiting.

5.1 Raw Disk versus RSnap volumes

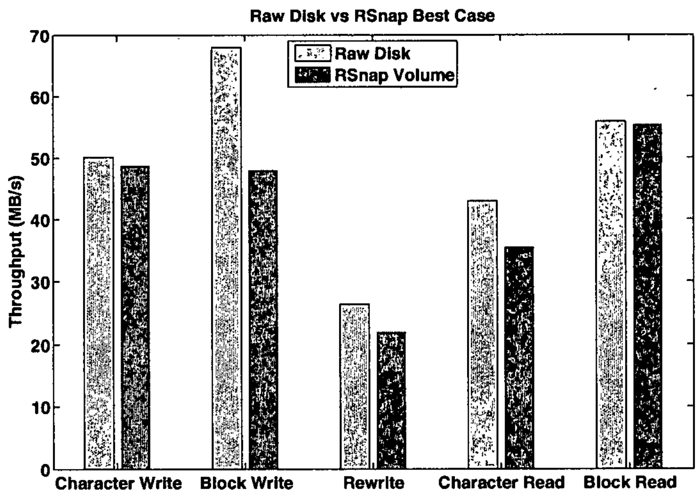
Figure 5.1 depicts the comparison between a raw disk and a RSnap radix tree volume. For comparitive purposes we configure RSnap in its best case and worst case scenarios. We observed that RSnap performs best when the blockstore block size is 4K and worst when it is 1K. For each of these tests, the file-system size was restricted to 3GB.

In Figure 5.1(a) both the raw disk and the RSnap volume are formatted with a 1K block sized ext2 file-system. Additionally, to force RSnap into its worst case behavior its blockstore is formatted with a 1K block size. On the other hand for the measurements in Figure 5.1(b), both the raw disk and the RSnap volume are formatted with a 4K block sized ext2 file-system. In this case, RSnap's blockstore is formatted with a 4K block size. We ran Bonnie++ on these different configurations to compare their performance. Each value in the graph is averaged over 3 consecutive runs of Bonnie++.

From Figure 5.1(a) we can observe that for 1K block sizes the performance of block writes on an RSnap volume can go down by 77%. However, for a 4K block size (Figure 5.1(b), it goes down by only 36%. The difference can be attributed to the fact that a radix tree node with a 1K block size can only store 128 64-bit addresses in it, whereas a 4K radix tree node can store 512 64-bit addresses in it. Due to this, the height of the radix tree for a 4K block size is always less than the one with a 1K block size. Thus, with a 1K block size, the metadata upkeep can be expensive in terms of both RAM consumption and disk seeks involving metadata reads. Note

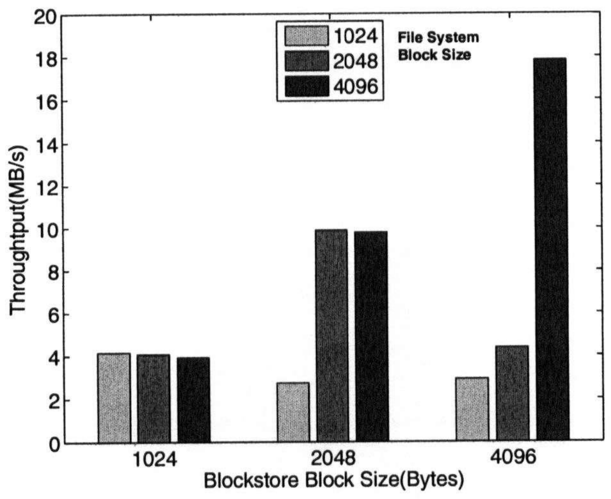


(a) Raw Disk performance as compared to RSnap worst case. For both cases, file-system block size was set to 1K. Blockstore block size was also set to 1K.

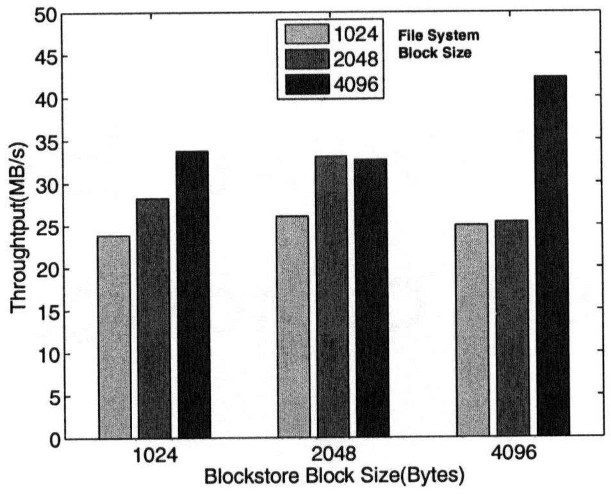


(b) Raw Disk performance as compared to RSnap best case. For both cases, file-system block size was set to 4K. Blockstore block size was also set to 4K.

Figure 5.1: Raw Disk versus RSnap volumes



(a) Block Write Performance



(b) Block Read Performance

Figure 5.2: Copy-on-Write Performance for One Level Snapshots

that in Figure 5.1(a), character write performance is much better than block write performance. This is because we modified Bonnie++ to perform block writes first. Due to this, the block write test is performed with a cold cache and later tests benefit from RSnap's metadata cache. Overall, the overhead of the radix tree metadata is tolerable when the blockstore block size is 4K.

5.2 Copy-on-Write Performance

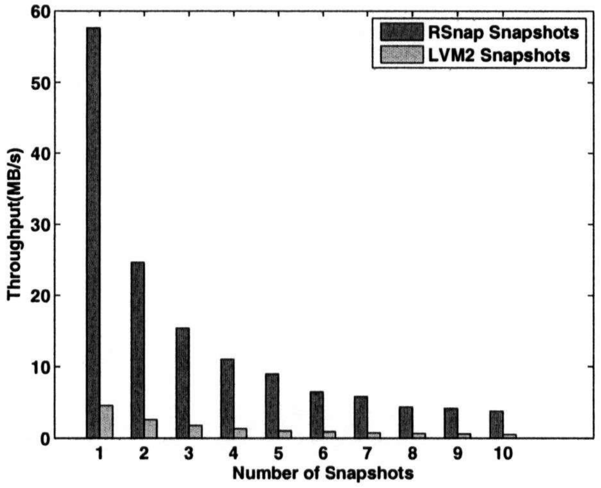
Figure 5.2 depicts the modified Bonnie++'s performance on RSnap's one level snapshots. Snapshots were tested with different combinations of blockstore and file-system block size. To create a more realistic scenario, we modified *mkRSnap* to randomly reserve 20% blocks in the blockstore. This ages the blockstore artificially and leads to non-contiguous block allocations during copy-on-write operations.

Since our version of Bonnie++ performs block writes first, the Copy-on-Write overhead is distinctly observable in Figure 5.2(a). The performance improves significantly during Bonnie++'s block read phase because there are no copy-on-writes in this phase and the relevant radix tree metadata is cached.

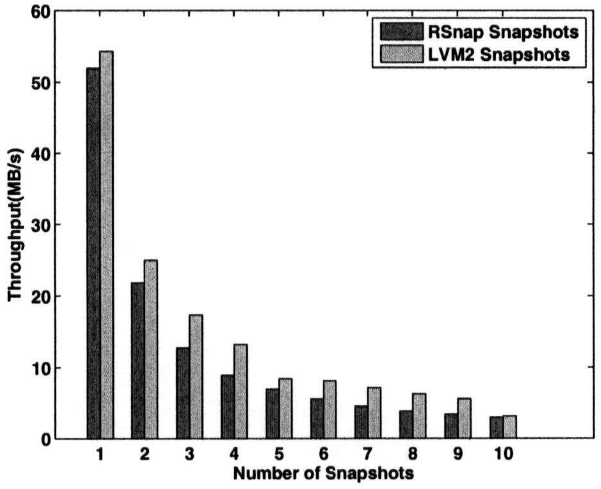
Note that performance is lower when the file-system's block size is less than the blockstore's block size. This is because in order to perform a copy-on-write the blockstore's read-only data block has to be read and then written at a new location. On the other hand, performance is best when the file-system's block size is equivalent to the blockstore's block size. Obviously, this is because the extra read is avoided. When the file-system's block size is larger than the blockstore's block size, then the device-mapper chops each I/O request into the blockstore's block size. Thus in this case performance is similar to the case when the file-system block is equal in size to the blockstore's block size.

5.3 Multiple Snapshots

In this test we create multiple snapshots of a single volume and then run our workloads on all of them simultaneously. RSnap's blockstore is formatted with a 4K block size and the base volume is formatted with a 4K block sized ext2 file-system. The radix tree for the base volume is 3 levels deep and represents around 3GB of storage. For comparitive purposes, we create a similar setup with LVM2 snapshots. Figure 5.3 depicts the average read and



(a) Average block write throughput per Volume



(b) Average block read throughput per Volume

Figure 5.3: Bonnie++ performance over multiple snapshots in parallel.

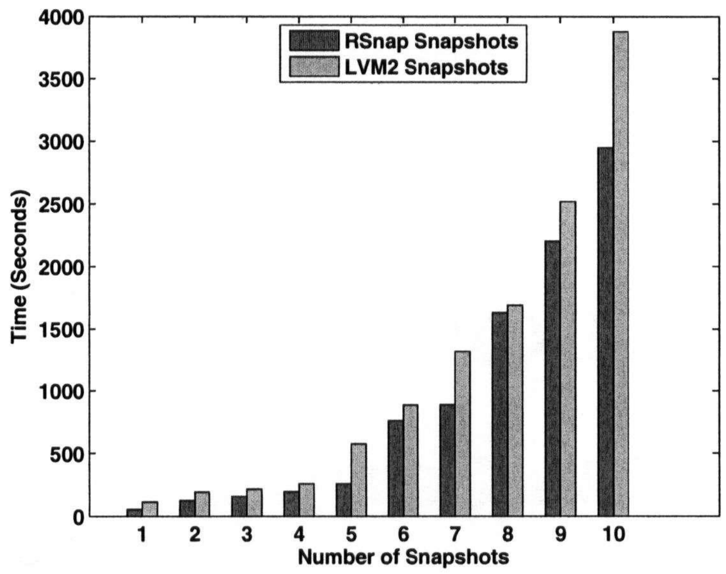


Figure 5.4: Average time taken per volume for a Postmark run

write throughput per volume with Bonnie++ as the workload and Figure 5.4 shows the average time taken per volume for a PostMark run.

From Figure 5.3(a) we can observe that the write performance for RSnap is up to 6-12 times better than LVM2. This is because LVM2 writes its copy-on-write mappings synchronously, whereas we perform in-memory metadata updates and rely on journalling for ensuring consistency. In the block read test LVM2 performs better than RSnap. This is because LVM2 data is organized contiguously on the disk, whereas RSnap incurs the overhead of the radix tree metadata. Also, when copy-on-write is being done on multiple snapshots in parallel, the organization of data for RSnap's volume tends to be more fragmented. As can be observed from Figure 5.3(b), the overall performance degradation varies between 5% to 40% with an average degradation of about 20%.

On the other hand, from the Postmark results in Figure 5.4, we can observe that RSnap actually performs better for more realistic workloads. The performance improvements observed are in the range of 3% to 55% with an average improvement of about 20%. This is mostly because RSnap does not perform metadata updates synchronously.

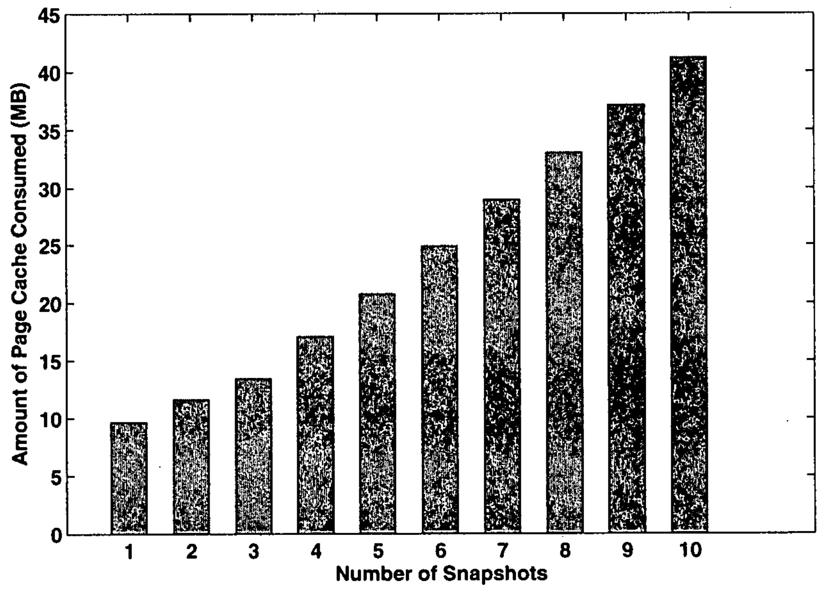


Figure 5.5: Page cache consumption of radix tree metadata.

5.4 Metadata Overhead

In this experiment we measure the page cache consumed by the radix tree metadata. As in the experiments in section 5.3, we first create a single volume and then create its multiple snapshots. Later, we run simultaneous instances of Bonnie++ on these snapshots and measure the cost of keeping their radix tree nodes in the page cache. Recall that we are working with a 1GB RAM and hence bonnie++ will try to saturate the cache with 2GB of data for each of the snapshot instances. In order to track the total amount of cache consumed by the metadata for all snapshots, we modified the `add_to_page_cache()` function to maintain the maximum number of pages associated with our blockstore device. Whenever, a new maxima is reached, the function logs it using a `printk()` call on a file maintained on an NFS server.

Figure 5.5 shows the results of our experiment. It can be observed that even with 10 instances of Bonnie++ the page cache consumption does not exceed 45MB, which is about 4.3% of the RAM size. Moreover the growth in the page cache consumption scales linearly with the number of snapshots.

Table 5.1: Garbage Collection Statistics after Postmark runs.

	Phase 2	Phase 3		
No. of Snaps.	No. of Copy-on-Write Blocks Read	No. of Tentatively Deleted Blocks Read	Time (sec)	Space Reclaimed(MB)
1	118	1422	2.2	164.77
2	203	1446	2.2	157.47
3	342	1446	1.14	132.50
4	449	1468	2.37	63.40
5	554	1447	4.53	143.57
6	679	1471	2.43	100.28
7	857	1469	5.37	75.20
8	955	1483	2.25	73.57
9	1078	1481	2.32	77.45
10	1128	1521	2.4	23.25

5.5 Evaluation of Garbage Collection

In order to quantify the performance of the garbage collector, we create multiple writable snapshots of a single volume and then run Postmark and Bonnie++ on them. After the tests are complete, we delete the parent volume so that all its blocks are marked as tentatively deleted and then run our garbage collector on the blockstore. Tables 5.1 and 5.2 show the results of our evaluations.

From both tables, we can observe that with the increase in the number of snapshots the number of copy-on-write metadata blocks to be read in Phase 2 increases. This further induces a rising trend in the number of tentatively deleted metadata blocks which are read in Phase 3. The total time taken by the garbage collector does not show any trend because it depends upon the number of metadata and nibble map blocks which are cached. The observation to be made is that with appropriate caching the time taken is in the order of seconds. Finally, the space reclaimed is quite variable in the case of Postmark and is almost the same in all test cases for Bonnie++. This is because Postmark has a more randomized nature whereas, Bonnie++ generates a very predictable workload. Further note that in the case of Postmark, the amount of space reclaimed tends to fall with an increase in

Table 5.2: Garbage Collection Statistics after Bonnie++ runs.

	Phase 2	Phase 3		
No. of Snap.	No. of Copy-on-Write Blocks Read	No. of Tentatively Deleted Blocks Read	Time (ms)	Space Reclaimed(GB)
1	1051	489	1.9	2.01
2	2100	494	1.76	2.00
3	3154	499	1.80	1.99
4	4203	511	1.98	1.97
5	5251	547	1.92	1.90
6	6303	511	1.89	1.97
7	7364	549	2.09	1.89
8	8400	519	2.06	1.95
9	9460	547	1.89	1.90
10	10525	549	2.36	1.89

the number of snapshots. This is because a randomized workload results in the overwriting of different read-only logical blocks in indvidual snapshots. Due to this a larger number of read-only blocks continue to be referenced in one volume or another and few blocks can be reclaimed by the garbage collector. On the other hand, in the case of Bonnie++, most of the logical blocks which are overwritten are the same in peer snapshots and hence almost a constant amount of space can be reclaimed after each run.

5.6 Evaluation Summary

A first observation to be made is that RSnap performs best when the block-store's block size is equivalent to the file-system block size. Further, the performance is better for larger block sizes. In our experiments, we observe peak performance when the block size is 4K.

The second observation to be made is that when multiple snapshots are running a stress test workload in parallel, the copy-on-write performance is 6-12 times better than LVM2 and post copy-on-write the average performance degradation is about 20%. On the other hand in realistic usage scenarios, RSnap may perform as much as 55% better than LVM2 snapshots.

A third observation to be made is that caching the radix tree metadata imposes very little overhead on the page cache. In our experiments, we ran 10 snapshots in parallel and their collective metadata overhead was merely 4.3% of the RAM. Further, assuming that the workload on various snapshots is similar, the cache consumption scales linearly with increase in the number of snapshots.

A final observation to be made is that garbage collection overhead is quite bearable. With adequate caching, a garbage collection cycle can be completed within seconds. However, the amount of data that can be reclaimed is dependent upon the type of workload the snapshots got exposed to. The garbage collector will reclaim more blocks if similar logical blocks get overwritten in peer snapshots.

Chapter 6

Future Work and Conclusion

6.1 Future Work

There are several avenues for future work in RSnap. A few interesting ones are listed in the following paragraphs.

6.1.1 Blockstore Extensibility

One of the shortcomings of RSnap is that the blockstore is not extensible. Once all space within the blockstore is exhausted and if even the garbage collector cannot free more space then any subsequent requests that entails free block allocation will fail. Being able to add new physical devices to augment the existing blockstore will be a worthwhile feature to explore.

6.1.2 Block Contiguity

As of now we follow a very simple round-robin style block allocation policy. This has the potential to override the contiguity assumptions made by a file-system residing on a volume. The block allocator may be modified to allocate logically contiguous blocks at physically contiguous locations. Alternatively, a utility to dynamically restructure the radix tree to bring logically contiguous blocks in physical proximity will be quite useful.

6.1.3 Limiting exportability

Copy-on-write volumes are a superb protection against malicious writes, however, they still can't prevent malicious reads. It would be useful to design a strategy in which a user can indicate the files they would prefer not exporting to a snapshot. Thus, any subsequent read/write attempts to that file in a child snapshot will be failed. File-grained semantic information may be communicated through techniques explored in the Semantically Smart Disks[25] project and the unused bits in each entry of a radix tree metadata node may serve as a store for per-block protection information.

6.2 Conclusion

We have designed and implemented RSnap, a volume manager that allows a user to create recursive writable snapshots of a logical volume. Whereas previous volume managers have considered snapshots as special read-only volumes dependent on their parent for existence, RSnap grants them considerable freedom with regard to mutability and lifetime.

For purposes of volume metadata upkeep, RSnap uses the light-weight radix tree data structure. In order to alleviate the latency in virtual to physical block address translation, it relies extensively on the kernel page cache for maintaining the intermediate nodes of a radix tree. Further, we have provided consistency guarantees on the radix tree data structure by capitalizing on the Linux journalling layer.

Our evaluations show that the overhead of page cache consumption is tolerable and scales with the number of volumes. Further we have shown that for strenuous workloads our radix tree based implementation outperforms LVM2's copy-on-write implementation by 6 to 12 times and on an average may perform only 20% slower on post copy-on-write operations. Further, for more realistic workloads RSnap's journalled metadata approach improves performance by up to 55% over LVM2 snapshots.

Lastly, we have addressed the complexity of reclaiming blocks from uninteresting read-only snapshots within a family of volumes and have presented a fast fault-tolerant garbage collector as a feasible solution.

Bibliography

[1] Laurent Amsaleg, Olivier Gruber, and Michael Franklin. Efficient incremental garbage collection for workstation-server database systems. In *Proceedings of the 21st Very Large Databases International Conference*, Zürich (Switzerland), 1995.

[2] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: a new approach to improving file systems. In *Proceedings of the 14th ACM symposium on Operating systems principles*, pages 15–28, New York, NY, USA, 1993. ACM Press.

[3] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, October 2003.

[4] Michail D. Flouris and Angelos Bilas. Clotho: Transparent data versioning at the block i/o level. In *12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies*, College Park,Maryland, 2004.

[5] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2):127–153, 2000.

[6] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, volume 21, pages 155–162, 1987.

[7] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, 1994.

[8] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, scalable disk imaging with frisbee. In *USENIX Annual Technical Conference*, June 2003.

[9] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, 17–21 1994.

[10] Geoffrey Hough and Sandeep Singh. 3par thin provisioning. In *3PAR White Paper*, June 2003.

[11] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. physical file system backup. In *Proceedings of the 3rd ACM Symposium on Operating systems design and implementation*, pages 239–249, Berkeley, CA, USA, 1999. USENIX Association.

[12] Charels B. Morrey III and Dirk Grunwald. Peabody: The time travelling disk. In *IEEE Mass Storage Systems and Technologies Conference*, April 2003.

[13] Jeffrey Katcher. Postmark: A new file system benchmark. *Network Appliance Technical Report Number TR3022.*

[14] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th ACM symposium on Operating systems principles*, pages 223–236, New York, NY, USA, 2003. ACM Press.

[15] Mirsolav Klivansky. A thorough introduction to flexclone volumes. In *Network Appliance Technical Report (TR3347)*, October 2004.

[16] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Cambridge, MA, USA, 1996.

[17] Sun Microsystems. Instant image white paper.

[18] Robert E. Passmore and April Adams. Dynamic volume expansion: Not all arrays are equal. *Gartner Report. Note Number G00127194*, 2005.

[19] Zachary Peterson and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[20] Sean Quinlan. A cached WORM file system. *Software — Practice and Experience*, 21(12):1289–1299, 1991.

[21] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[22] Yasushi Saito, Svend F., Alistair Veitch, Arif Merchant, and Susan Spence. Fab: building distributed enterprise disk arrays from commodity components. In *Architectural Support for Programming Languages and Operating Systems*, October 2004.

[23] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the 17th ACM symposium on Operating systems principles*, pages 110–123, Charleston, SC, United States, 1999.

[24] Margo Seltzer, Greg Ganger, M. Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX Annual Technical Conference*, pages 18–23, June 2000.

[25] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2003. USENIX Association.

[26] D. Teigland and H. Mauelshagen. Volume managers in linux. In *Proceedings of USENIX 2001 Technical Conference.*

[27] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex Snoeren, Geoff Voelker, and Stefan Savage. Scalability, fidelity and containment in the potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, United Kingdom, 2005.

[28] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: Managing storage for a million machines. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, USA, 2005.

[29] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th ACM Symposium on Operating systems design and implementation*, pages 77–90, San Francisco, CA, USA, 2004.

[30] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.