Disk Performance of Copy-On-Write Snapshot Logical Volumes

by

Bhavana Shah

B.E., Indian Institute of Technology Roorkee, India , 1999

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

August, 2006

Abstract

Data Snapshot technology is a standard feature of modern storage systems. Most such systems use copy-on-write techniques to manage snapshot data in order to optimize storage space requirements for maintaining history data. Copy-on-write methods tend to write data out-of-place at a location which may be far away from the original location of the data on the disk. This phenomenon gradually leads to fragmentation of the on-disk snapshot data and degradation in the disk I/O performance. This work analyzes Logical Volume Manager's (LVM2) snapshot technology and studies the effect of copy-on-write on the on-disk placement of the snapshot data. Based on these findings, we propose new disk space allocation and data placement techniques for snapshot volumes in order to reduce physical distance between related blocks and improve disk access performance. A prototype is implemented and its performance is compared with the original LVM2 implementation in order to measure the effectiveness of the proposed schemes. The new schemes tend to perform better than the old LVM2 ranging from 18% to 40% at the cost of some performance penalty for first time writes in some cases.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisor Dr. Norman C. Hutchinson who not only helped shape the main ideas in this thesis but also gave me constant encouragement and support, making the whole experience very rewarding and enjoyable. Many times during the course of the thesis, he provided valuable insight into the problems and showed new research directions. Working under his mentorship has been a truly wonderful experience.

I am also grateful to Dr. Charles Krasic for being the second reader and helping with suggestions to improve this work. His efforts in setting up the emulab and maintaining the netbed cluster are very much appreciated as this has provided an excellent platform to the DSG students for conducting research and experimentation.

I would like to thank Cuong, Geoffrey, Brendan, Ken and other members of the DSG Lab for their tips, suggestions and feedback.

Last but not the least, I am indebted to the Department and the Faculty of Computer Science for providing continuous financial assistance, computing facilities and an inspiring work atmosphere for conducting research.

Chapter 1

Introduction

1.1 Problem Statement

This thesis aims to study the fragmentation effects of copy-on-write techniques on the snapshot logical volumes and to devise new disk allocation and data placement schemes for these volumes in order to improve disk I/O performance. In this section we briefly describe the environments in which snapshot logical volumes are used, the ways in which copy-on-write techniques effect their on-disk data placement and the characteristics of modern day disk systems which impact their performance.

In today's information age, the need to store large amounts of data, maintaining data history, and retrieving it efficiently has grown significantly. Modern storage systems invariably provide some means of capturing current data state and storing it for later use. Data snapshotting functionality is provided by the storage systems either at the file system level or at the disk block level. Section 4.1 provides a survey of such systems. Traditionally data backups have been used for purposes like data mining and recovery from accidental errors. Recently, OS-virtualized architectures like Xen [16] and Denali [9] have provided yet another reason to use data snapshots. In such environments, the physical storage system needs to be transparently and efficiently shared by multiple, independently running virtual machines, each requiring its own private storage volume to host its file systems. Preparing a fresh storage volume and copying the entire root filesystem on it, for each new virtual machine, can be a time-consuming and disk-devouring process. An alternate solution is to create instant storage volume snapshots, which are consistent point-in-time copies of a base root filesystem image, using copy-on-write techniques, and then using them to support the active file systems of each virtual machine.

Copy-on-write technique for snapshotting data is fast because it does not prepare a data copy at the time of the snapshot creation. Data blocks are copied from the original data store to the snapshot data store as they are written. Thus, only those data blocks which are changed after the creation of the snapshot occupy space in the snapshot data store. Although fast and disk space-efficient, copy-on-write technique writes data out of place and breaks physical contiguity of logical blocks. The resulting disk fragmentation raises concerns about the performance of snapshot volumes. Moreover, since the file systems operating above these volumes are mostly unaware of this block-level indirection, it further complicates the performance prediction of such systems under various different workloads.

The problem of disk fragmentation is important to attend because the placement of data on the disk can greatly impact its performance. The access time for hard disk drives is a combination of seek time, rotational latency and data transfer time. Seek time is the time required to position the disk head over the required track and rotational delay is the time taken by the required sector to rotate underneath the head. Data transfer rate is the number of bytes rotating under the head per second. Trends in disk technology show that while the data transfer rates are improving with increasing disk data densities, the mechanical delays (seek time, rotational delay) are not improving at the same pace. In the face of these limitations, system designers employ different techniques to partially offset these delays and improve disk performance. Some of these techniques include using a buffer cache to serve I/O requests from the cache instead of going to the disk for every request, increasing block size to reduce seek overhead, permuting the disk head requests in order to reduce seek distances etc. Another set of efforts in this field is to carefully place free blocks on the disk in order to optimize disk performance for different workloads. This task is challenging because modern day disks rarely expose their true geometries to the BIOS for reasons of complexity and transparency. The disk blocks are accessed using linear block addressing (LBA). In this scheme, sectors are numbered sequentially starting from zero and the drive internally translates these sequential numbers into physical sector locations. There is a general understanding that disks map sequential logical block numbers(LBN) to adjoining sectors and therefore blocks closer to each other in the logical addressing space are very likely to be physically close on the disk. Therefore, instead of randomly allocating free blocks, system designers aim to place related data closer in the logical address domain so that they end up physically closer thereby reducing seek delays.

In the case of copy-on-write, the need is to co-locate related blocks of the snapshot volume some of which may be lying on the original volume data store. This will help reduce the number of long disk seeks to and from the original and the snapshot volume while running a workload on the snapshot volume. In this thesis, we study Linux's Logical Volume Manager (LVM2) and its copy-on-write snapshot technology. Based on our findings about the performance of the LVM2 snapshot volumes, we propose new disk space allocation and block placement techniques and build a prototype which serves as a proof of concept. In the subsequent sections, we provide a primer on the design and software architecture of LVM2 followed by a detailed description of its snapshot technology. The remainder of this thesis is structured as follows:

Design and Implementation: In Chapter 2, we first present the results of some of our preliminary experiments conducted over LVM2 snapshot volumes in order to verify our concerns regarding degradation in the disk I/O performance due to copy-on-write data displacement. Subsequently, we describe the design decisions we made, the desgin itself and the implementation details of the LVM2 prototype.

Evaluation: In Chapter 3, we present the results of our experiments which compare the disk I/O performance of the old LVM2 snapshot volumes and the new LVM2 snapshot volumes and analyze if the new scheme is effective in reducing disk seeks and improving performance.

Related Work: Chapter 4 provides a survey of various storage systems which have implemented copy-on-write snapshot technology, their strategies to organize snapshot data on the disk and any available performance statistics for these systems. Further, we discuss the work on 'Virtual Contiguity', which deals with similiar problems as ours at the file system level.

Conclusion: Chapter 5 presents the conclusions we derive from this work and provides pointers for future work.

1.2 Background on LVM2

LVM2 [13] is the latest Logical Volume Manager for Linux. It provides a higher-level view of the storage system than the traditional view of disks and partitions. By hiding away the details of physical disk management, LVM2 offers an easy-to-use and flexible interface to manage disk space.

The physical disk interface imposes lots of restrictions on the way disks can be managed and configured. Disk partitions, once created are difficult to resize, and that too, can not be done online, that is without unmounting the file system based on them. Also, the blocks constituting a physical partition need to be contiguous on the disk. Thus, the size of a file system mounted on a physical partition is bounded by the partition size and finally by the physical disk size. Another problem with this static scheme is that it does not allow for shrinking and growing of partitions in the face of changing needs of the users with time. These limitations are especially problematic in multi-user environments with large numbers of disks to manage.

A logical volume manager hides all the above mentioned limitations from the user and allows the system administrator to flexibly allocate disk space to users and applications. It sits between the file system and disk partitions and provides a seamless interface to higher layers by binding the disparate disks and partitions underneath. A logical volume can be grown or shrunk in size, even without dismounting the existing file system in many cases. An LVM2 logical volume doesn't have to be made of continuous blocks, it even doesn't have to be on the same disk. LVM2 takes care of binding all the pieces together and mapping the blocks of a logical volume to the correct disk blocks. With LVM2, adding or removing of disks can be done transparently. The logical volumes can be simply resized to adjust to the new underlying disk configuration in such a case.

In the following subsections, we shall discuss the common terms used with LVM2, its design, software architecture and important datastructures, which are helpful in understanding its internal workings.

1.2.1 LVM2 Terminology

LVM2's storage model can be understood as a hierarchical structure as depicted in Figure 1.1.

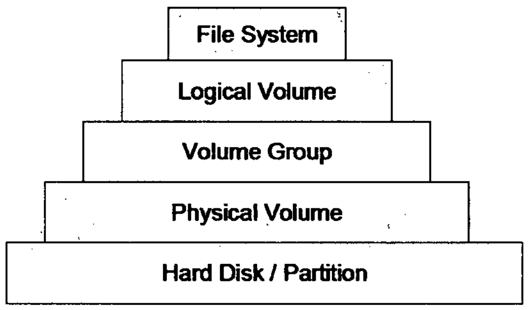


Figure 1.1: LVM2's Storage Hierarchy

An LVM2 **Physical Volume (PV)** is typically a hard disk, or a hard disk partition which has been prepared to be used by LVM2. By prepared, we mean LVM2-specific configuration and identification information has been written on it. This information includes the LVM2 label, one or two copies of LVM2 metadata and a physical volume identifier which is unique to the system. The contents of the LVM2 metadata are discussed later.

Once we have created the physical volumes, we can create a volume group from them. An LVM2 **Volume Group (VG)** is made up of one or more PVs. At the time of the creation of a volume group, the user can specify the basic unit of allocation, which will be used later to allocate space from the contained physical volumes. This basic unit of allocation is termed as **Physical Extent Size**.

After the creation of a volume group, logical volumes can be created within it. LVM2 **Logical Volumes (LV)** are allocated space from the PVs contained in the VG. The logical volume size should be a multiple of the logical extent size, which is the basic unit of allocation within a VG and is equal to VG's physical extent size. The space allocation is done according to the allocation policy specified by the user at the time of creation of the logical volume. The various allocation policies supported by LVM2 are discussed in Section 1.2.2.

Finally, one can create a filesystem on top of the logical volume.

The containment rules for PV, VG and LVs are depicted in Figure 1.2.

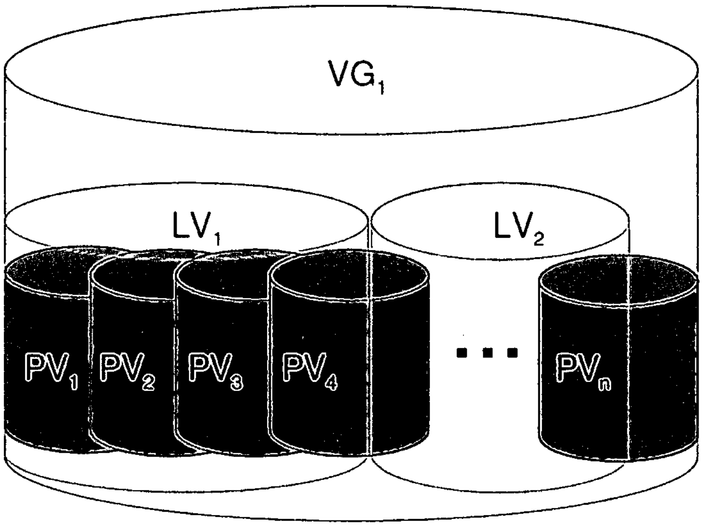


Figure 1.2: Relation between LVM2 VGs, PVs and LVs (source [14])

A PV can only include a single hard disk or a single hard disk partition. A VG can be made of multiple PVs but a PV can only belong to one VG. Similarly, multiple LVs can be carved out of a VG but an LV can only belong to one VG. At the same time, an LV can be allocated space from one or more PVs contained in the LV's volume group. Also, there can be only one filesystem based on top of a logical volume.

LVM2 Metadata Format and contents: LVM2's metadata stores information about the PVs, VGs and LVs, which needs to be persistent across machine reboots and is required by LVM2 to detect its volumes on the disk at boot. The LVM2 metadata format is an ASCII text format. Each VG has metadata defined for it describing the physical volumes and the logical volumes contained in it. Normally, there is one copy of metadata placed at the beginning of each PV contained inside the VG. The metadata is updated everytime any change is done to the volume group like creation, deletion or resizing of a logical volume, addition or removal of a physical volume, etc.

The on disk LVM2 metadata has the following parts:

a) Label : Occupies one sector near the start of each PV . It contains the LVM2 label, a unique identification number of the PV and a pointer to the metadata areas and data areas on the physical volume.

b) Metadata Areas : Each metadata area in turn has a header section and a circular buffer containing the metadata. The header contains the checksum of the metadata, the start position and size of the metadata area. The circular buffer, in turn, specifies to which volume group the physical volume belongs, and for the volume group, describes all the physical volumes and logical volumes contained within it. A logical volume is described as an ordered list of logical segments each of which maps to a corresponding physical segment.

1.2.2 LVM2 Design and Software Architecture

LVM2 is designed as a user-space command line interface which in turn communicates with a kernel-driver to manage logical volume mappings. The software architecture of LVM2 is detailed in Figure 1.3

As shown in the figure, the LVM2 software architecture is composed of three main components, namely: the command line interface, the device mapper library, and the device-mapper kernel driver. These are described below:

The LVM2 command line interface (CLI): LVM2 provides a unified command line interface to manage physical volumes, volume groups and logical volumes. The CLI has commands for creation, deletion, resizing, attribute modification, scanning, and displaying of PVs, VGs and LVs. This layer of software resides in the user-space and is responsible for reading and updating the LVM2 metadata. In the following paragraph,

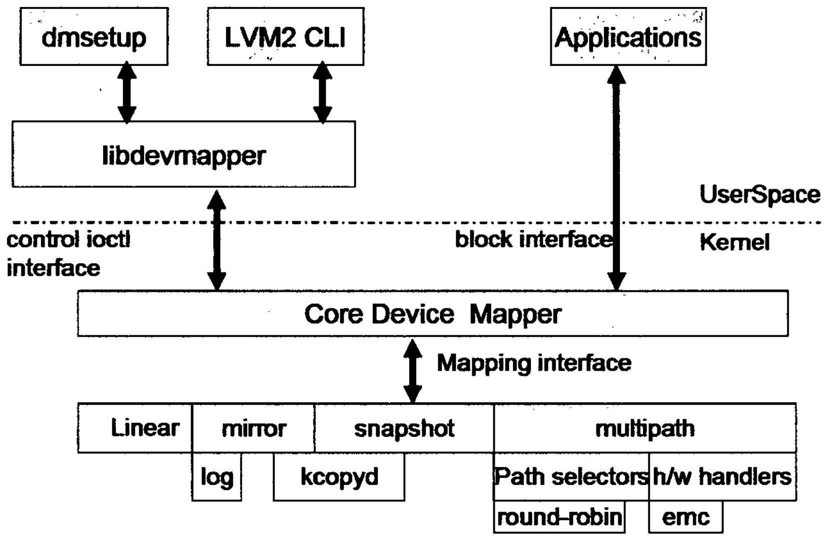


Figure 1.3: LVM2 Software Architecture

we discuss the process of creation of a new logical volume, with emphasis on the space allocation policies of LVM2.

Creation of a New Logical Volume: In order to create a new logical volume, the user has to specify the name, size, and volume group of the new logical volume. Optionally, he can also specify the list of physical volumes on which to allocate the logical volume, and/or an allocation policy. LVM2 first prepares a list of free physical segments on the PVs specified by the user or on all of the PVs of the VG, if no preference is given. This list of free physical segments is maintained in increasing order of their size. After that, LVM2 allocates space to the logical volume from these physical segments, depending on the allocation policy specified by the user. LVM2 supports following two allocation policies :

Contiguous: According to this allocation policy, the logical volume should be laid down on the disk in one single continuous chunk. The creation of the LV may fail if a physical segment of the required size is not found in the list of free segments.

Anywhere: This is the default allocation policy for logical volumes. As the name suggests, in this case LVM2 starts allocating segments to the logical volume from the head of the free segment list, till the required amount of space has been allocated. If the last allocated segment size is bigger than the space left to be allocated, the segment is split.

The device-mapper kernel driver: Once a logical volume is created using the LVM2 command line interface, it can be used to create file systems on top of it and direct read/write I/O to it. We need a kernel mapping driver which maintains the mapping from the logical volume to the physical disk sectors and routes these disk requests to the required physical blocks. The device-mapper kernel driver is that piece of software which takes care of this. It provides a generic framework for volume management. It has no knowledge of the volume groups and metadata formats used by the user space applications like LVM2. It only has the concept of a logical block device, for which it maintains a mapping table that specifies how to map each range of logical sectors of the device onto a target device, using one of the supported mapping types. The device mapper supports various kinds of sector mappings from a source block device to a target block device. Table 1.1 lists some of the mapping types implemented in the current version of the device mapper. Each of these mapping types are loaded as separate modules and registered with the core device mapper module.

Each table mapping for a block device has the form :
<start-sector> <length> mapping-type <mapping parameters>
where the mapping parameters are dependent on the type of mapping. The start-sector and length fields are in the logical domain.

LVM2 creates and registers a logical block device with the kernel-resident device mapper for each of its logical volumes. The notion of Volume Groups and Physical Volumes is only maintained at the LVM2 level and is not exposed to the device mapper which treats each logical volume as a uniquely identifiable and independent block device. At machine boot, LVM2 scans its metadata to get information about the volume groups and the logical volumes they contain. Then it registers each logical volume and its mapping table with the device mapper through the control ioctl interface exported by the device mapper. The device mapper, in turn, assigns these logical block devices a

Mapping	Arguments	Action
Linear	<start> <len> linear <dev> <start>	Maps onto a continuous range of other block device
Error	<star> <len> error	All I/O to this mapping is dicarded and error returned
Zero	<start> <len> zero	Read returns blocks of zero Writes are silently discarded
Striped	<start> <len> striped <#stripes> <chunk size> [<device> <start>]	Stripes data across devices
Crypt	<start> <len> crypt <cipher> <key> <IV offset> <device> <start>	Encrypts the data passing through
Snapshot -origin	<start> <len> snapshot-origin <origin device>	Reads go directly to origin device For Writes, first make a copy of the sector on all those snapshots of the origin device, which are sharing the sector with it and then proceed with write on origin.
Snapshot	<start> <len> snapshot <origin device> <snapshot device> <persistency> <chunk size>	Reads go to origin device if sector is still shared with origin, otherwise goes to snapshot device. If the sector to be written is shared with origin, first copy the corresponding chunk of sectors from origin to snapshot device , then proceed with write on snapshot.

Table 1.1: Types of Device Mapper Mappings

major and a minor number and registers them with the kernel. Subsequently, any block I/O request coming for these devices are routed to the device mapper module, which redirects the I/O to the appropriate target block device, after consulting the device's mapping table.

The device mapper library: The device mapper library provides a programming interface to the applications using the device-mapper. It hides the difference between various versions of the device-mapper driver. The purpose of the library is to marshall the arguments for the ioctl commands given by the applications, according to the underlying kernel driver version and unmarshall the results returned from the driver into a format appropriate for the applications.

1.2.3 The Mapping Table

The mapping table maintained by the device mapper for each block device is arranged as a btree. Each of the segments in a logical volume is mapped as a specific target inside the device mapper. The device mapper allows the mapping type for each segment to be different. For instance, some segments of a block device may have a linear mapping to a physical volume area while others may have an error mapping. The keys of the btree are the boundary logical block numbers of the block device segments and the leaf nodes are the mapping-specific target datastructures which contain the necessary information and functions to map the incoming block requests to the target block device.

At the time of the block device creation, the device mapper prepares the leaf nodes of the btree, one target node for each logical segment inside an LV. Once all the targets are added , the device mapper prepares a set of btree indexes based on the logical block numbers these segments represent, thus completing the mapping table.

An example mapping is shown in Figure 1.4. It shows a logical volume at the top along with its nine logical segments. Eight of these segments are linearly mapped to segments on the physical disk while the ninth one is mapped to an error segment. The figure shows the btree data structure which maps the logical segments to their targets. Each node within the btree, except the leaf nodes, holds multiple keys each pointing to one child node. The keys denote the high boundaries of the logical segments represented by the child nodes. By following down the index nodes in the btree starting from the root node using the high boundary of the logical segment one wants to map, one can get to the segment's physical target.

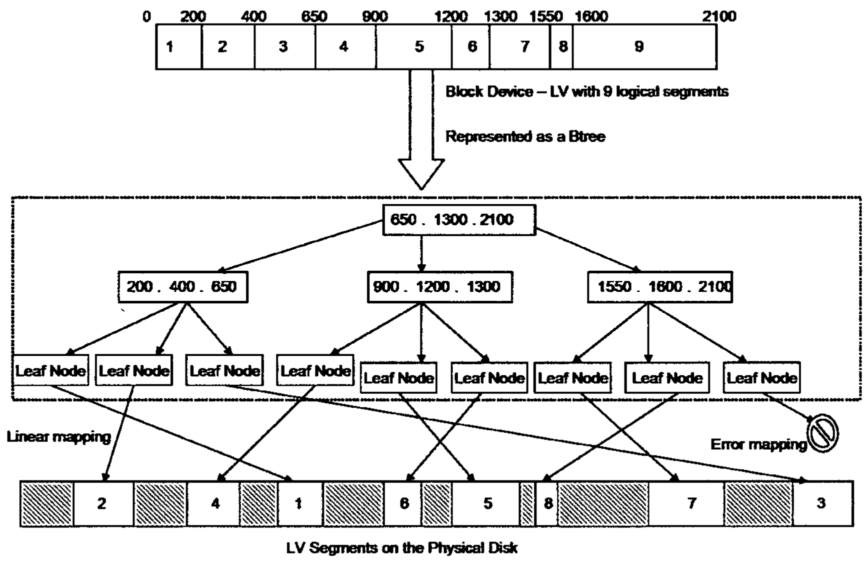


Figure 1.4: An Example of a Btree Mapping of a Logical Volume

1.3 LVM2's Snapshot Technology : Design and Implementation

Unlike most storage snapshotting systems, which support readonly snapshots only, LVM2 supports writable snapshots of the logical volumes. The writability feature makes the usage of snapshots more flexible, as it allows to capture the state of the storage device at some point in time, remount it later and run live applications on it. This scenario is especially relevant in virtual machine environments where there is a need to configure filesystems for each new guest operating system before running it, and doing so by taking a snapshot of a standard base image is fast and convenient.

LVM2 uses copy-on-write technique to maintain snapshot data. The basic idea is that the snapshot volume initially points to its origin volume blocks. After the snapshot volume creation, any block which needs to be written, either on the origin volume or on the snapshot volume, is first copied from the origin volume to the snapshot volume and then written to. Thus, the snapshot volume holds only the changed blocks while blocks which are unchanged since the time of their creation, are still shared with the origin volume.

In the following sections, we shall discuss how LVM2 creates a snapshot logical volume and does the bookkeeping for tracking changed blocks.

1.3.1 Creation of an LVM2 snapshot logical volume

LVM2 creates a snapshot logical volume in two steps. The first step is similiar to the creation of a plain logical volume, as discussed in Section 1.2.2. The important thing to note here is that LVM2 does the space allocation to the snapshot logical volume without any consideration of the position of its origin volume on the disk. Therefore, the snapshot volume segments may lie anywhere on the disk with respect to its origin volume segments. Figure 1.5 shows an example scenario of the placement of a snapshot and its origin volume segments on the physical volume, assuming that both of the volumes are allocated on the same physical volume.

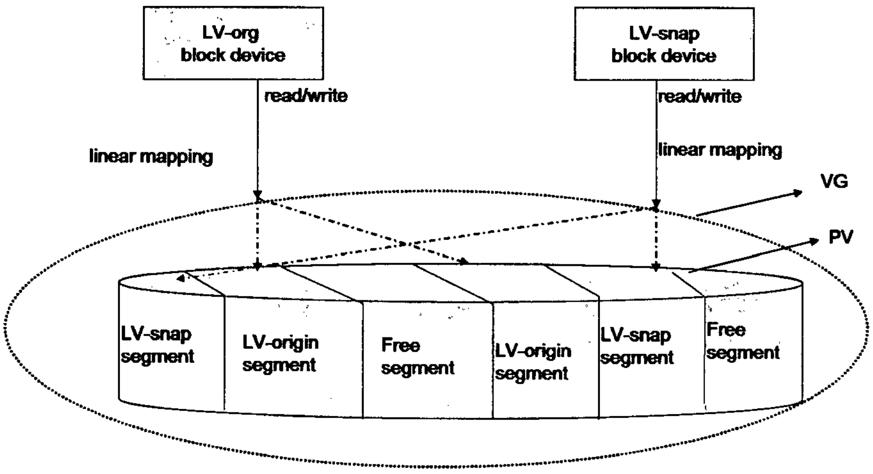


Figure 1.5: LVM2 Snapshot Volume Creation - Step One

In second step, LVM2 inserts another virtualization layer between the logical volume layer and the physical volume layer. This layer contains the logic for keeping track of which blocks have changed since the creation of the snapshot volume and where they are located on the snapshot volume. Figure 1.6 shows the final mapping of the snapshot and origin logical volume segments to their physical position on the disk.

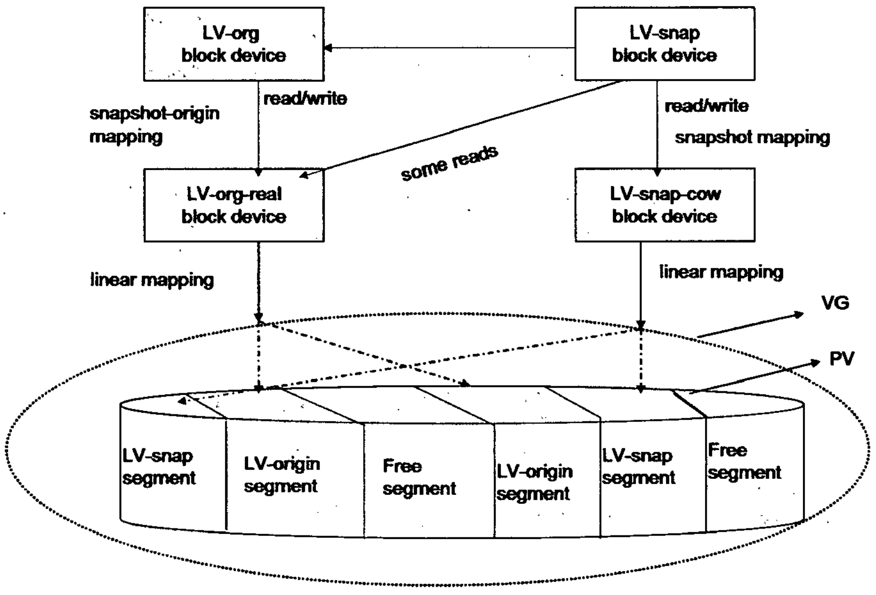


Figure 1.6: LVM2 Snapshot Volume Creation - Step Two

As shown in the figure above, the origin LV is mapped to a 'origin-real' device using the 'snapshot-origin' mapping type. Similarily, the snapshot LV is mapped to both the 'origin-real' and 'snapshot-cow' devices using the 'snapshot' mapping type. In turn, the 'origin-real' device is mapped to the origin LV segments, which were allocated to the origin LV at the time of its creation, using a linear mapping. Likewise, the 'snap-cow' device is mapped to the snapshot LV segments, allocated to the snapshot LV in step one, using a linear mapping.

Figure 1.7 shows how the data on the snapshot volume is arranged on the 'snap-cow' device. The device is shown to be divided into equal pieces called 'chunks'. A 'chunk' is a contiguous collection of blocks, whose purpose is explained a little later in this section. The first chunk of the 'snap-cow' device contains the label for the LVM2 cow device type. The rest of the device contains the data blocks which have been copied from the origin volume and the metadata, called an 'exception-table', which has entries mapping the copied data location on the 'origin-real' device to the block location on the 'snap-cow' device. Data chunks are allocated on the 'snap-cow' device starting from the third chunk in a sequential order. Metadata chunks are allocated on the 'snap-cow' device starting from the second chunk, spaced out by the number of data chunks whose mapping can be held in one chunk of metadata. The 'exception-table' mapping helps to route read/write I/O requests coming for the snapshot volume to the right place, as described in the following paragraphs.

In order to reduce the size of the exception table, LVM2 copies the data and maintains its mapping chunk-wise, which is by default equal to 8K in size, instead of block-wise. An asynchronous copying daemon, called 'kcopyd daemon', handles the copying of chunks from the origin to the snapshot volume and informing the main request processing thread of the copy completion. The snapshot mapping module reads the entire exception table from the 'snap-cow' disk into main memory, at machine boot, for faster processing of incoming I/O requests. This table is called 'completed exception table'. Another table called 'pending exception table' maintains all those exceptions, which have been allocated a new chunk on the snapshot device but for which the chunk copy has not been completed yet.

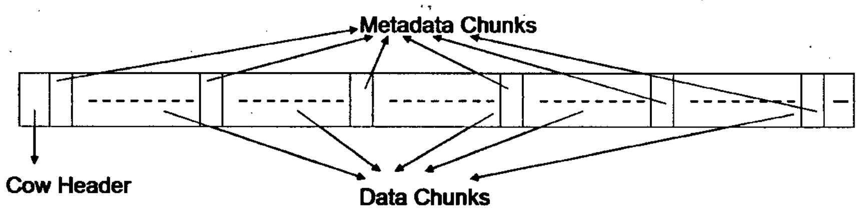


Figure 1.7: Snapshot Cow Device

The **'snapshot-origin'** mapping handles the I/O for a logical volume which is an origin LV for one or more snapshot logical volumes in the following way:

Read: A read request on an origin volume is simply routed to the 'origin-real' block device, without any change.

Write: For a write request, the exception tables of each of the snapshot volumes of this origin volume are consulted to check if the particular block to be written has been copied to the snapshot volume or not. If not, then a free chunk is allocated on the snapshot volume, and a request is issued to the ‘kcopyd’ daemon to copy the chunk containing the block from the ‘origin-real’ device to the ’snap-cow’ device. Once the chunk has been copied to the ‘snap-cow’ device, an entry is made in the exception table mapping the old logical chunk number in the ‘origin-real’ device to the new logical chunk number on the ‘snapshot-cow’ device. Finally, the write request is routed to the ’origin-real’ device.

The **‘snapshot’** mapping handles the I/O for a snapshot logical volume in the following way:

Read: For a read request on a snapshot volume, the exception table of the volume is consulted to check if the block to be read has been copied to the ‘snap-cow’ device or not. If yes, then the read request is routed to the ‘snap-cow’ device. Otherwise, it is routed to the ‘origin-real’ device.

Write: For a write request, the exception table of the snapshot volume is consulted to check if the particular block to be written has been copied to the snapshot volume or not. If not, then a free chunk is allocated on the snapshot volume, and a request is issued to the ‘kcopyd’ daemon to copy the chunk containing the block from the ‘origin-real’ device to the ’snap-cow’ device. Once the chunk has been copied to the ‘snap-cow’ device, an entry is made in the exception table mapping the old logical chunk number in the ‘origin-real’ device to the new logical chunk number on the ‘snapshot-cow’ device. Finally, this entry is used to get the block number on ‘snap-cow’ which is to written, and the request is updated with this block number and routed to the ‘snap-cow’ device.

Once the read or write request reaches the ‘snap-cow’ or ‘origin-real’ device, it is mapped to the final physical block on the disk using linear mapping.

1.4 LVM2's Snapshot Technology : Disk Block Placement Analysis

In the last section, we discussed the way LVM2 creates logical volume snapshots and maintains their copy-on-write block mappings. In this section, we shall analyse the ways in which virtualization at the logical volume level may affect disk I/O performance.

File systems make their own allocation decisions for file, directory and metadata blocks, at their level, depending on their goals and policies. Some file systems optimize data placement for improving disk performance in general, like the Fast File System [7] which tries to co-locate file metadata and data on the disk for fast reads and writes, whereas some have their allocation policies designed for optimizing write performance, as in the Log Structured File System [17], where the writes go to a sequential log on the disk, thereby speeding up the write performance.

By having logical volumes underneath the file systems instead of actual physical volumes, some of the assumptions made by the file system may no longer hold true. In case of logical volumes, the file system has no knowledge that there is another layer of software underneath, which is re-routing its allocation requests. Logical volumes, unlike physical disks or partitions, are not guaranteed to be continuously laid down on the disk, and may even be allocated on multiple disks or partitions. Because of this possible segmentation in logical volumes, the allocation decisions made at the file system layer may not pay-off or may even turn out to be sub-optimal.

In addition, the LVM2's snapshotting logic adds another level of indirection in the I/O path, as seen in the above section. LVM2's snapshot logical volume does not have a direct linear mapping for its filesystem data blocks. It only keeps changed data blocks in its allocated space and, that too, in a sequential log fashion interspersed with regular snapshot metadata blocks. Rest of the unchanged blocks lie on the origin logical volume. Thus, we see that the file system, which is hosted on the snapshot logical volume, has some of its blocks lying in the origin volume space and the rest of them in the snapshot volume space, without its knowledge. Further, the origin volume and snapshot volume space on the disk itself may be segmented and may lie anywhere on the disk with no effort made by LVM2 to co-locate these segments. We saw in the last section that LVM2 allocates space to the snapshot volume just like any other ordinary logical volume with no consideration to the fact that this volume's data may be related to the data on its origin volume.

We see from the above discussion that LVM2's implementation of snapshot logical volumes does not make any special effort to place snapshot's changed data blocks near their original locations, in order to improve disk I/O performance for these volumes. While we expect some degradation in performance for first time writes to either of the origin or snapshot volumes due to the extra chunk copy from the origin to the snapshot volume, it is the read performance which we are concerned about because of the copy-on-write displacement of changed blocks from their original location, thereby disturbing the physical contiguity of related blocks. In the next Chapter, we present the results of our preliminary experiments on the LVM2 volumes, which verify some of our concerns related to I/O performance of snapshot logical volumes due to copy-on-write effects, and elaborate on the design decisions we took in proposing new schemes for space allocation to logical volumes and placement of changed blocks in snapshot volumes for LVM2.

This chapter presented the problem which this thesis deals with. Since our work is based on LVM2, we have provided a background on its terminology, software architecture and internal workings. Later, we describe the design and implementation of the snapshot logical volumes in LVM2.

Chapter 2

Design and Implementation

In this chapter we present the results of some of our early experiments with LVM2 snapshot volumes, which corroborated our concerns regarding performance degradation in LVM2 snapshot volumes due to poor block placement, and helped us in deriving design guidelines for creating new schemes for disk space allocation of LVM2 logical volumes and for smart placement of copy-on-written blocks on the snapshot volumes in order to maintain their proximity with their related blocks. Section 2.1 details these experiments and their findings. Section 2.2 discusses the conclusions which we draw from these results which informed our design of new allocation and block placement schemes for LVM2. Section 2.3 presents our design and Section 2.4 describes its implementation.

2.1 Experience With LVM2 Snapshot Volumes

We conducted a set of experiments to study the actual effects of copy-on-write block displacement on the disk I/O performance of LVM2 snapshot volumes. The purpose was to study the degree of fragmentation caused by copy-on-write on the snapshot volumes and to see if this fragmentation really affected the performance. Our premise was that the spatial discontiguity between different related blocks of a snapshot volume would lead to more disk seeks, amounting to an increase in the disk access times. In the following sections, we explain these experiments and their results.

2.1.1 Experiment Setup

We have conducted our experiments over the Xen [16] virtual machine monitor platform, which is an OS-virtualized environment running the Xen hypervisor at the hardware interface level and supporting the execution of multiple guest operating systems on top of it. Xen is open source software developed at the University of Cambridge. In OS-virtualized environments, LVM2 logical volume snapshots come to offer great ease in capturing the initial file system configuration from a base image and hosting live Guest OSes on top of them. Therefore, we contemplate that the performance of LVM2 snapshot volumes in these environments is important and interesting to investigate.

Instrumentation: We instrumented the ide disk driver in the linux kernel to record the following information about the I/O requests reaching the disk:

- Type of I/O request : read or write
- Time of I/O request arrival
- The disk block number which is being read or written
- Type of filesystem block, i.e., if it is an inode, file data, directory, indirect, journal, superblock, block bitmap, inode bitmap or group descriptor block which is being read or written.

This information is routed by the kernel syslogger to another machine on the local network, in order to prevent this logging activity from affecting our results. In the Xen environment, there is a single privileged GuestOS called Domain0 which has access to the hardware resources like the machine's physical disks and network devices. All the of other GuestOSes run in unprivileged mode in which they see virtual block devices (VBDs) and network interfaces (VIFs) which are created and configured within Domain0. The actual disk and network drivers thus reside in Domain0 and receive disk read/write requests from and send responses to unprivileged domains via Xen's inter-domain communication interface.

Platform Configuration: We run our experiments on a 2.8GHz Pentium 4 machine with 512MB RAM and 40GB Western Digital(WD400BB-23FJA0) ATA disk drive. The machine hosts Xen-2.0 ported over the base Fedora Core 4 operating system. The GuestOSes run modified unprivileged Fedora Core 4 operating systems. For our experiment, we run Domain0 on a 15GB origin logical volume and an unprivileged GuestOS on a 10GB snapshot logical volume. Figure 2.1 shows how these volumes are laid out on the disk.

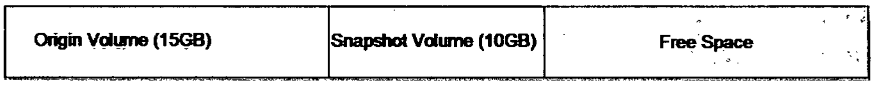


Figure 2.1: Origin and Snapshot volume on the Disk

Workload Description: In this experiment, we run a Linux kernel source tree compile and grep workload. We first install a clean kernel source tree on a 15GB logical volume. The size of the kernel source tree is around 209MB and after compilation, it becomes 1.22GB. Then we build this kernel source tree and record the time and disk block requests for this activity. After this, we run a grep on this compiled kernel source tree and record the time and disk block requests this time too. The grep is done on the entire kernel source tree with object files included. These build and grep times on the plain logical volume are the base times against which we shall compare the performance of build and grep on origin and snapshot logical volumes.

Next, we make a 10GB snapshot of this plain logical volume. The size of a snapshot volume can be less, equal to or more than its origin logical volume. Thus, now the plain logical volume becomes an origin logical volume for the newly created snapshot volume. Now, we clean the kernel source tree on the origin logical volume and build it again. This time all the first writes during kernel build on the origin logical volume shall generate a copy-on-write block copy from the origin to the snapshot volume. We record the time and disk block requests for this activity. Then we do a grep similiar to the previous one on the original logical volume's kernel source tree. Now, we instantiate a

GuestOS on the snapshot logical volume and also run a grep on its kernel source tree. For this source tree, the source files will lie in the origin volume space while those object files which have been overwritten will be copied to the snapshot volume area. Thus, the grep on the snapshot kernel source tree will exhibit performance penalties, if any, due to disk seeks between the source files and object files lying on different volumes.

2.1.2 Experiment Results

The results of our experiments are compiled in Table 2.1. From the table, we deduce that the kernel build time on the origin volume is 6.7% more than the kernel build time on the plain logical volume, due to the copy-on-write induced extra chunk copies from the origin to the snapshot volumes. More importantly, we find that the grep performance on the snapshot volume is 62% less than on the origin volume.

Volume	Kernel-Build Time (min:sec)	Kernel-Grep Time (min:sec)
Plain LV	39:39	4:40
Origin LV	42:20	4:45
Snapshot LV	–	7:42

Table 2.1: Kernel Build and Grep Timings on Plain, Origin and Snapshot Logical Volumes

We have plotted the disk block accesses for the kernel build workload on the plain LV and the origin LV. Figure 2.2 shows these graphs. The x-axis in these graphs represents the time in milliseconds during the kernel build and the y-axis represents the disk block number which was accessed at a particular point of time. In the graphs, the 'READ' data points represent the disk blocks which were read during the kernel build workload and the 'WRITE' data points represent the disk blocks which were written. During kernel source tree compilation, the source files contained in each directory of the tree are read and compiled. The objects files thus written on the origin volume cause the corresponding original data blocks to be copied from the origin to the snapshot volume.

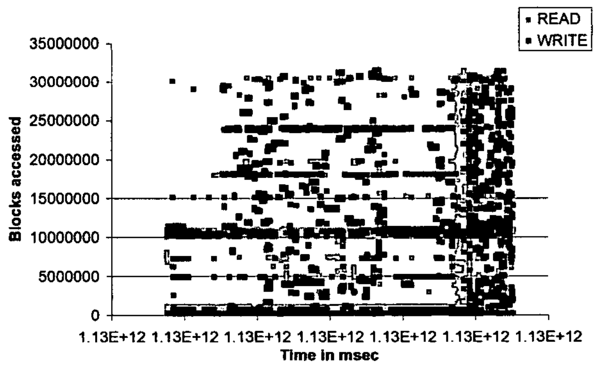
We see two dense horizontal bands of 'READ' blocks in each graph. The upper band is probably the region of disk space where the kernel source files are placed. The lower band on the origin volume contains the ext3 file system's metadata like journaling blocks, inode bitmap, block bitmap etc., which are heavily read and written during the kernel build workload. We notice that towards the end of the kernel build workload, there is a heavy vertical band of disk reads followed by disk writes. This disk read activity takes place at the object files' linking stage and the disk write activity is caused during the module building stage.

On comparing the two graphs, we note a linear 'WRITE' curve in the top portion of the lower graph. These are the copy-on-write disk writes during kernel build on the origin volume which copy original data from the origin volume to the snapshot volume.

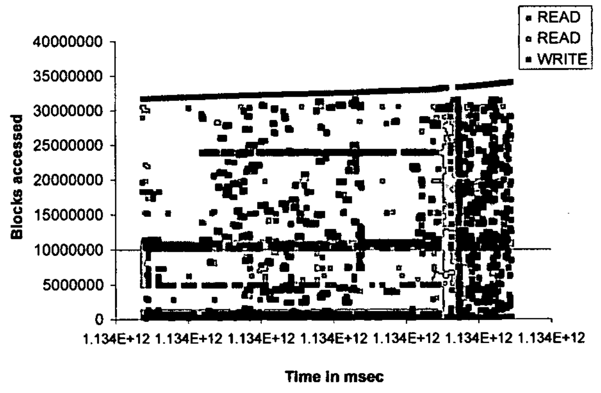
Similiarly, Figure 2.3 and Figure 2.4 show the disk block accesses for the kernel grep workload on the origin LV and the snapshot LV respectively. The disk block access graphs mostly have 'READ' data points as the kernel grep workload mostly consists of disk reads. Comparing the graphs in Figure 2.3(a) and Figure 2.4(a), we see that the graph pertaining to the snapshot volume has a prominent 'READ' band at the top of the graph. These data points represent the blocks which were copied from the origin volume to the snapshot volume during the kernel build workload.

These figures also contain graphs showing the disk access times vs. disk seek distances for this workload on the two volumes. The purpose of these graphs is to see if the 62% decrease in the snapshot read performance is due to increased disk seeks or not. The x-axis in these graphs is the seek distance in units of disk blocks and the y-axis represents the disk access time in milliseconds. Note that in Figure 2.4(b), the seek distances have increased for a large amount of disk accesses as compared to Figure 2.3(b). This increase is due to the longs seeks which the disk has to make between blocks lying in the origin volume and those lying on the snapshot volume.

The results of this experiment show that copy-on-write displacement of blocks across LVM2 volumes can lead to significant performance degradation of read workloads.

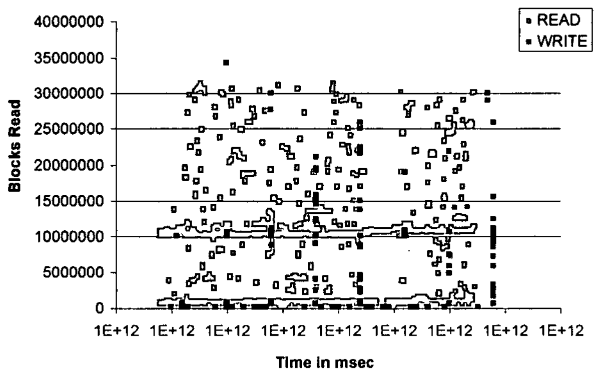


(a) Blocks Accessed during Kernel Build on the Plain Volume

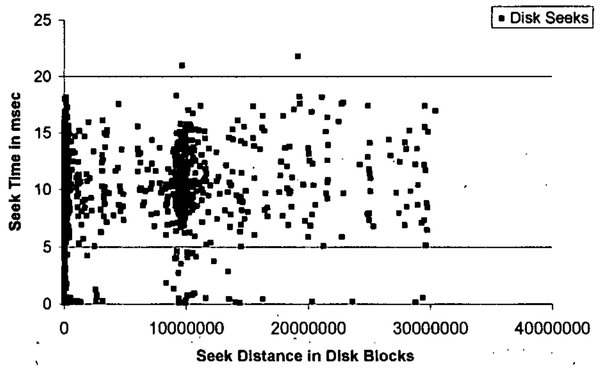


(b) Blocks Accessed during Kernel Build on the Origin Volume

Figure 2.2: Difference between 'kernel-build' disk accesses on plain and origin LVs.

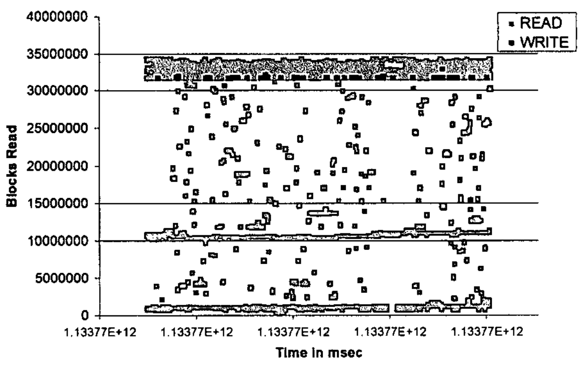


(a) Blocks Read during Kernel Grep on the Origin Volume

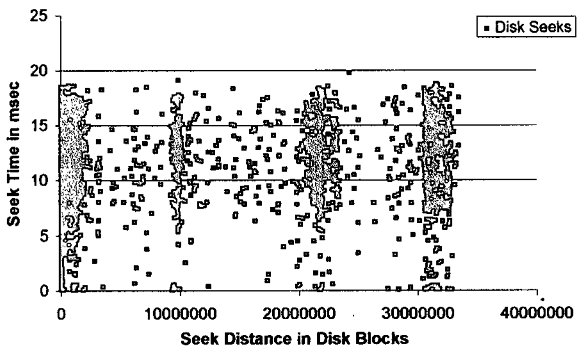


(b) Seek Profile during Kernel Grep on the Origin Volume

Figure 2.3: Disk Access Profile of the Kernel Grep Workload On the Origin LV



(a) Blocks Read during Kernel Grep on the Snapshot Volume



(b) Seek Profile during Kernel Grep on the Snapshot Volume

Figure 2.4: Disk Access Profile of the Kernel Grep Workload on the Snapshot LVs.

2.1.3 Spatial Density of Copy-On-Written Blocks

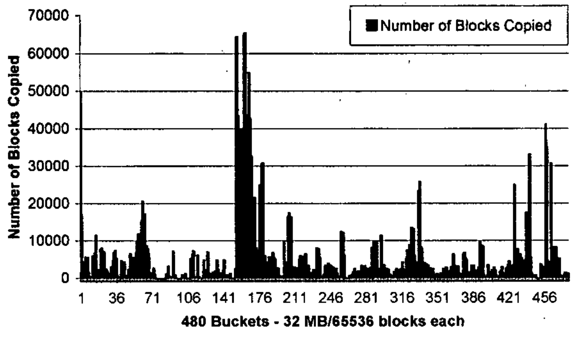
In order to understand the distribution of blocks which were copied from the origin volume to the snapshot volume, we extracted the 'exception-table' from the 'snap-cow' device and plotted the original location of copied blocks on the origin volume. This information is depicted in Figure 2.5(a). The x-axis represents the origin volume disk space divided into 480 buckets of 32MB each, while the y-axis represents the number of disk blocks which were copied from the origin volume to the snapshot volumes from each bucket. The graph in Figure 2.5(b) depicts the same information but sorted by the number of blocks copied rather than the slice index.

In these graphs, we see that the physical distribution of blocks copied from the origin volume to the snapshot volume is non-uniform, with most of the blocks copied from a few regions while the remaining regions are untouched or copied very little.

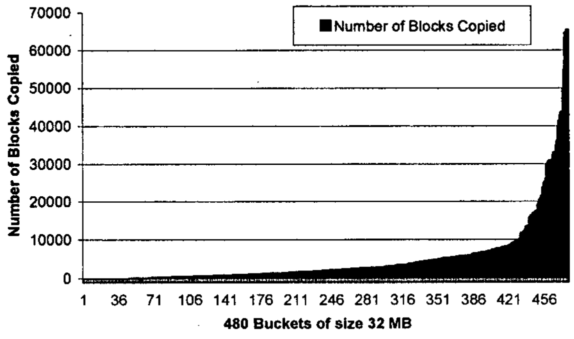
2.2 Design Decisions

Provisioning of Free Space near the Origin Volume: The results of the preliminary experiment, conducted on LVM2-based logical volumes show that the displacement of related blocks caused by copy-in-write can be large enough to cause a significant degrading effect on the disk read performance. We found out from the seek profile of the kernel grep workload on the origin and the snapshot volume that the increased seek distances in the case of the snapshot volume are one of the probable reasons of the degradation in performance. Earlier, we noted in Section 1.2.2, that the space allocation for the origin and snapshot volumes, in case of LVM2, is done completely independent of each other. We decided to modify the scheme of space allocation for logical volumes in LVM2, so that it provisions for some free space near these volumes for future allocation of snapshot data blocks. This will help in keeping displaced copy-on-written blocks near their original locations and thereby reducing the long disk seeks which we saw in our experiments.

Dynamic allocation of Snapshot Volume Space: At first, we thought of allocating space statically to the snapshot logical volumes, at the time of their creation, in seg-



(a) Density Distribution of Blocks Copied during Kernel Build on the Origin Volume - Sorted by Slice Index



(b) Density Distribution of Blocks Copied during Kernel Build on the Origin Volume - Sorted by Number of Copied Blocks

Figure 2.5: Spatial Density of Copy-On-Written Blocks

ments adjacent to the origin volume segments. This allocation scheme would perform very well in co-locating displaced blocks if the data writing and copying takes place uniformly across all origin volume segments. In Figure 2.5, we see that the block displacement from the origin volume due to copy-on-write occurs non-uniformly, with most of the blocks being copied from very few regions of the origin volume. Keeping this in mind, we decided to replace the LVM2's static space allocation scheme for the snapshot logical volumes to a dynamic one in which space gets allocated to the snapshot volume in small segments on-demand at run-time and at locations which best preserve the spatial locality of related blocks. This method will not only allow more flexibility in sharing the free space provisioned near the origin volumes for various snapshot volumes, but will also reduce space wastage in the snapshot volumes and co-locate related blocks in the best possible way.

2.3 Design

In this section we discuss the design of the new disk allocation and block placement schemes which we have developed for LVM2 in order to improve the disk performance of snapshot logical volumes. First, we describe the design changes we introduced in the static space allocation scheme of plain logical volumes in LVM2. Then, we describe the design of a dynamic space allocation scheme and an intelligent block placement scheme for LVM2 snapshot volumes.

2.3.1 'Snap-Aware' Static Allocation Policy for LVM2 Logical Volumes

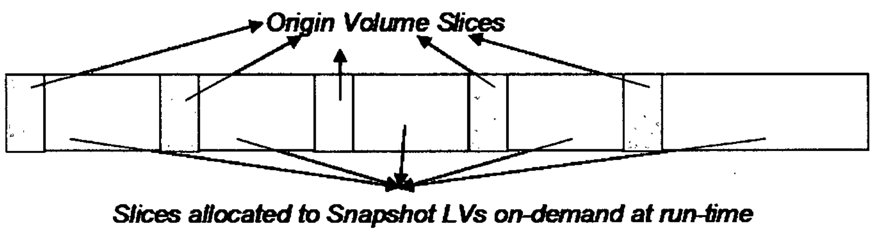
We have designed a new static allocation policy, called 'Snap-Aware', for LVM2 logical volumes, which leaves free space in between the volume segments for the volume's future snapshot logical volumes. According to this allocation scheme, a new logical volume will be allocated equal-sized segments called slices which are equi-distantly spaced from each other. The size of these slices and the gap between them is configurable by the user at the time of the creation of the logical volume.

The 'Snap-Aware' allocation policy accepts following parameters:

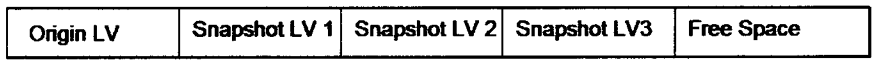
- Number of Slices: The user can specify the number of slices in which to divide the logical volume. The size of each slice can thus be calculated by dividing the size of the volume by this number.
- Size of a Slice: Optionally, the user can specify the size of a slice instead of the total number of slices.
- Number of Snapshots: The user can tentatively suggest a value for the number of snapshots he is provisioning for at the time of creating a logical volume. This will help the LVM2's space allocator in deciding the size of the inter-slice gap.
- Percentage Size of Snapshots: Along with the number of snapshots to provision for, the user can specify an approximate value of the size of a snapshot volume as a percentage of the origin volume size, so that the total size of the free space to be provisioned for can be calculated accordingly.

Figure 2.6 shows an example of a logical volume which has been allocated space using the 'Snap-aware' allocation policy and how it may have looked if allocated using the 'Contiguous' allocation policy of LVM2.

Discussion: This scheme makes the space allocation of a logical volume anticipate that in future some snapshot logical volumes may be based on it and would benefit by being closer to it. The scheme has been desgined to offer flexibility to the user in deciding how much slicing and space provisioning he wants to do. At the same time it also encumbers the user with the burden of deciding suitable values for these parameters which are apt for the kind of workloads he expects to run on the system. There are certain tradeoffs which have to be considered while choosing a value for the above parameters. For example, the slice size should not be so big that it renders the snapshot blocks to be too far from their original allocations to be of any value. Similarily, the gap between the slices should be not be too much that it separates the original volume data so much that it hurts its performance. At the same time, the gap between the slices should be big enough to hold the difference data pertaining to its



(a) Snap-Aware Allocation Policy



(b) Contiguous Allocation Policy

Figure 2.6: Contiguous Vs. Snap-Aware Volume Space Allocation

neighbouring origin volume slices otherwise the difference data would spill over to other areas. But, as we saw in Figure 2.5, since the physical distribution of blocks copied from the origin volume is non-uniform and workload-dependent, it is difficult to anticipate the required free space provisioning to prevent spilling over of copied data from the intended physical disk area.

2.3.2 Dynamic Allocation Policy and Intelligent Block Placement for LVM2 Snapshot Logical Volumes

On-Demand Disk Space Allocation: For the LVM2 snapshot logical volumes, we have introduced two changes. The first change is in the way disk space is allocated to the snapshot volume. LVM2, in its original design, allocates space to a snapshot logical volumes statically at the time of its creation. We have modified this allocation scheme to make it dynamic. In the new LVM2 scheme, at the time of the creation of a snapshot volume, its blocks will be mapped to a single error segment equal in size to that of the snapshot volume size. Actual disk space will be allocated to the snapshot volume in terms of equal sized segments at run-time, and at a location near the disk chunk which is to be copied from the origin volume to the snapshot volume. This allocation will be done by a userspace-resident LVM2 space allocation daemon, which gets allocation requests from the kernel-resident device mapper module at run-time. The size of the segment which is allocated in one request is configurable and can be different for different snapshot volumes of the same origin volume.

Smart Block Placement: The second change which we have done is in the way data and metadata chunks are placed on the snapshot volume space. In the original LVM2 scheme, as we saw in Figure 1.7, the data chunks in the logical block domain are allocated on the snapshot volume in a sequential manner starting from the third chunk with single metadata chunks appearing between them after regular intervals. In the new LVM2 scheme, the logical block domain of the snapshot logical volume is divided into segments of equal size, each equal to the amount of space allocated by the LVM2 space allocator in one allocation request, and one error segment at the end mapping all the blocks which are yet unallocated. Instead of placing chunks contiguously in the logical domain, in the new LVM2 scheme, the device mapper, on getting an I/O request, first finds out the physical location of the chunk which is being copied from the origin logical volume to the snapshot logical volume. Then it tries to find an allocated segment whose physical location is within a desired range of the chunk's physical location. If such a segment is found, then the device mapper picks a free chunk from it, maps the I/O request to the new chunk and routes it to the target physical device. Free chunks are picked from the segment in a sequential manner. If such a segment is not available or does not have a free chunk, then the device mapper stalls the I/O request and sends a segment allocation request to the LVM2 space allocation daemon. Once the segment allocation is done, the device mapper adds this segment to its mapping table for the snapshot device and proceeds with the servicing of the I/O request.

Thus, in summary, we have sliced up the origin volume and left free spaces between the slices to be allocated to its future snapshot volumes. Additionally, we have made the space allocation to snapshot volumes dynamic so that space can be allocated to them as and when required and at a location which best preserves the spatial locality of snapshot volume blocks with their related blocks lying in the origin volume. Further, we have modified the block placement strategy within the snapshot volume segments so that the displaced blocks can be placed in that segment of the snapshot volume which is nearest to the original location of the blocks.

Discussion: There are certain concerns which ought to be discussed with regards to the above dynamic space allocation scheme. First concern is about the time overhead incurred at runtime for allocating space to the snapshot volume since the disk requests which need to write to the unallocated space are delayed for this amount of time. One obvious solution to minimize the impact of this overhead is to increase the amount of space allocated to the snapshot volume in one allocation request, thereby reducing the number of such requests. But this approach may lead to large amount of space wastage in those segments of the snapshot volume which hold very little copy-on-written data. This may also negatively affect the space allocation of other snapshot volumes which may not get free space in the desired regions because of the extra allocation done by previous snapshots. In our experiments (refer Section 3.1), we find that the overhead of dynamic space allocation is minimal and does not effect the overall performance of the snapshot volumes significantly. But, this overhead may become large if the size of the allocated segments is reduced considerably leading to frequent requests for space allocation. In short, this scheme calls for striking a balance between the tradeoffs of write time efficiency versus fine grained data placement and efficient free space utilization by snapshots.

2.4 Implementation

In the last section, we described the new space allocation and block placement schemes which we have introduced for LVM2 logical volumes. In this section, we discuss the software architectural changes, datastructure modifications and control flow changes which we made in the LVM2 software in order to implement these schemes.

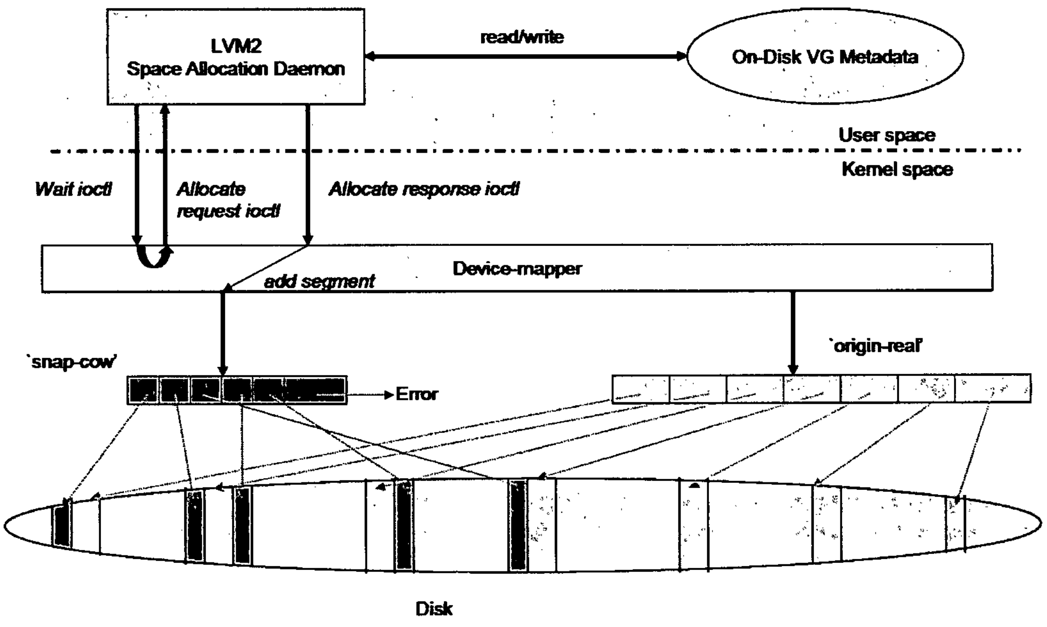


Figure 2.7: LVM2 Space Allocation Daemon

2.4.1 LVM2 Disk Space Allocation Daemon

We implemented a user-space resident disk space allocation daemon for dynamically allocating disk space to the snapshot logical volumes. It has a multi-threaded design. It interacts with the device-mapper using a set of ioctl commands as shown in Figure 2.7.

- Each thread within the daemon calls the 'wait' ioctl of the device-mapper and gets blocked till it reads a pending disk segment allocation request. This request has the following format :
  - The snapshot device id: This field uniquely identifies the snapshot volume which needs disk space to be allocated.
  - The lower physical address range: This field specifies the lower end of the physical disk address range within which to allocate the segment.
  - The upper physical address range: This field specifies the upper end of the

physical disk address range within which to allocate the segment.

- On getting the request, one of the daemon threads reads the current volume group metadata from the disk, prepares a list of free spaces on the VG, arranges them in the order of their physical location on the disk, and then attempts to find a free segment of the required size within the physical address range specified by the request. If such a segment is not found within the desired range, it tries to allocate a segment which is nearest to the specified address range.

- After allocating a free segment, the daemon thread updates the on-disk VG metadata to reflect the newly added segment and sends a segment allocation response back to the device mapper using the 'dev-alloc' ioctl with following details:
  - The snapshot device id
  - The lower and upper physical address range
  - The allocated segment's size.
  - The allocated segment's start address in the logical address range.

- On getting this response, the device mapper adds this segment to the mapping table of the 'snap-cow' device and proceeds with the servicing of pending disk I/O requests. In order to prevent new incoming requests from waiting for a long time till the above processing takes place, the mapping table datastructure is locked only for the time when the new segment is being added to it. After segment addition, the mapping table is unlocked. Subsequently, for each pending or freshly arriving request for this segment, the lock is reacquired in order to process the request.

2.4.2 Device Mapper Datastructures

In this section, we describe the datastructure changes we have made in the device mapper kernel module to implement the new block placement schemes for snapshot devices.

Snapshot Device Metadata: As we saw in Figure 1.7, in the original scheme new chunks are allocated sequentially from the snapshot volume space. The device mapper maintains a 'next-free' chunk counter for each snapshot device which is increased every time a new chunk is allocated on the 'snap-cow' device. In the new LVM2 scheme, the new chunks on the snapshot volume are not allocated sequentially on the 'snap-cow' device but are allocated on that logical segment of the 'snap-cow' device which is physically closest to the chunk's original location. Therefore, in the new LVM2 scheme the 'next-free' chunk counter is maintained within each logical segment of the 'snap-cow' device. Whenever a new chunk needs to be allocated, first an appropriate segment is chosen. Then, the next free chunk within that segment is allocated.

Mapping Tables: In the original scheme, the device mapper maintains a linear mapping table for each of the 'snap-cow' device and the 'origin-real' device. These mapping tables are indexed by the logical start block numbers of the segments allocated to these devices. In the new LVM2 scheme, the device mapper maintains an additional mapping table, one each for these devices, indexed by the physical start block number of the segments allocated to these devices. This kind of mapping table is useful for locating segments of a device based on their physical location on the disk. The usage of these tables is discussed futher in the next section where we describe the control flow within the device mapper module.

2.4.3 Device Mapper Control Flow

In this section, we will discuss the control flow of write requests which result in copying of data blocks from the origin to the snapshot volume. Such writes may either be directed to the origin device or to the snapshot devices. Figure 2.8 depicts a typical scenario of adding a new segment to the snapshot volume and copying a block from the origin volume slice to an adjacent snapshot segment.

- Step1: For an incoming write request, the device mapper looks into the completed exception table of the snapshot device to check if the block to be written has already been copied from the origin to the snapshot device. If yes, then the

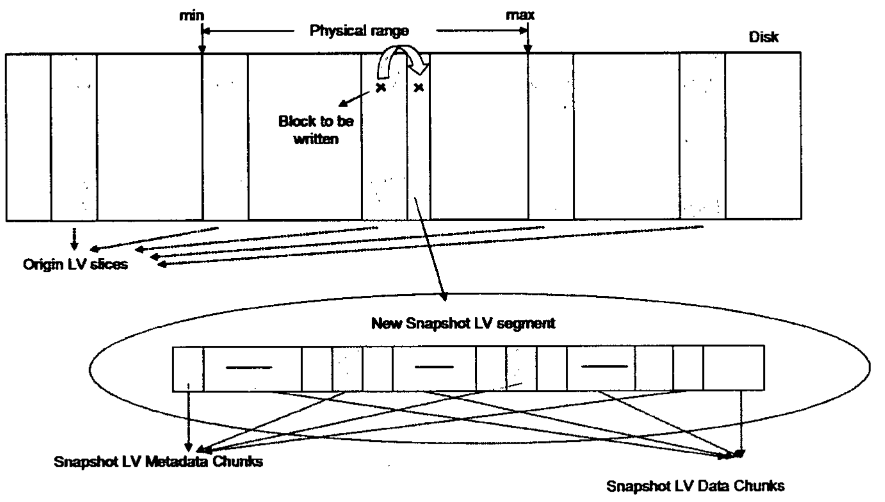


Figure 2.8: Allocating new Snapshot LV Segment and Copying Blocks to it

write request is simply routed to the target device with the correct block mapping. If no, then the device mapper follows Step2.

- Step2: Now, the device mapper first finds the physical location of the block to be written on the origin device. Then, using the 'origin-real' device's mapping table which is indexed by physical disk location, finds out the segment which contains this block and the neighbouring segments. By using the physical start location of the neighbouring segments, the device mapper defines a range of physical disk addresses within which the block should be copied.
- Step3: Using this physical addresses range, the device mapper looks into the 'snap-cow' device's mapping table indexed by physical disk location to find a segment which lies in that range. If such a segment is found which has a free chunk, then this chunk is allocated for the new exception and the chunk copy request is issued to the 'kcopyd daemon'. If no such segment with a free chunk is found, then the device mapper goes to Step4.
- Step4: The device mapper maintains a list of pending segment allocation re-

quests which have been sent to the userspace LVM2 allocation daemon. Each of these requests contains the range of physical addresses within which they have requested a segment to be allocated, and a list of pending exceptions which are waiting to allocate a free chunk from the segment. Each pending exception in turn has a list of pending disk I/O requests to be serviced once a free chunk has been allocated on the snapshot and the old data chunk has been copied to it. The device mapper looks into the list of pending requests to see if there is a segment allocation request already pending for this range of physical disk addresses. If yes, then it appends the pending exception to this request's list of pending exceptions. If no such request is present, then the device mapper creates a fresh segment allocaton request and sends it over to the LVM2 allocation daemon.

- Step5: On receiving a response to its segment allocation request, the device mapper adds this newly allocated segment to the mapping tables of the 'snap-cow' device. Then, it services all of the pending exceptions which were waiting to allocate free chunks from this segment.

Summarizing this chapter, we first presented the results of some of our early experiments with LVM2 snapshot logical volumes. These experiments not only verified our concerns about the performance degradation of the snapshot logical volumes due to copy-on-write, but also provided some guidelines to remedy this effect. Next, we described the design decisions we took and the actual design followed by its implementation.

Chapter 3

Evaluation

In this chapter, we shall describe the various experiments we conducted in order to evaluate our new disk space allocation and block placement schemes for the LVM2 snapshot logical volumes. The primary goal of these experiments is to determine if the new block placement scheme, in which the copy-on-written blocks are placed as close to their original locations as possible, improves read performance on the snapshot logical volumes or not. Other goals are to measure the effect of slicing on the disk I/O performance of the origin volumes, and to determine if the overhead of allocating disk space to the snapshot volumes dynamically significantly effects the runtime performance of these systems. These experiments have been conducted on two different sets of hard disk subsystems from different vendors in order to see how disk-specific characteristics like on-disk cache buffer, average seek times, etc., impact the results of our experiments.

3.1 Kernel Compile and Grep Workload

In the Section 2.1 we described a kernel compile and grep workload and presented the results of running this workload on LVM2 logical volumes in a Xen-based virtual machine environment. In this experiment, we use the same workload and compare the results obtained with the original LVM2 scheme and the new LVM2 scheme.

3.1.1 Platform Configuration

We did two runs of this experiment, each on a different machine. Table 3.1 compares the hardware configuration of the two machines. The first machine has a 2.8GHz CPU and a 40GB Western Digital IDE disk with 2MB of on-disk cache buffer, while the second machine has a 3.2GHz CPU and a 80GB Maxtor SATA disk with 8MB of on-disk cache buffer. Each machine has Xen-2.0 installed on it and the guest virtual machines, running on top of Xen, have Xen-ported Fedora Core 4 as their operating system.

The disk is prepared by creating a physical volume on it and then creating a volume group containing this physical volume. The extent size, which is the basic unit of space allocation for logical volumes, is configured to be 4MB for this volume group. After creating the volume group, a logical volume is created which is 15GB in size and the root file system containing an uncompiled kernel source tree is copied on it. Then a guest operating system is run on this volume which compiles the kernel source tree. Now a snapshot of the volume is taken. The snapshot's chunk size, which is the amount of data copied from the origin to the snapshot volume as a result of copy-on-write of a disk block, is configured to be 8K. At this point, both origin and snapshot volumes point to the same compiled kernel source tree. After that, we clean the source tree on the origin volume and run a kernel build workload which recompiles the kernel source tree and writes object files. Once the kernel compilation is complete, we do a grep on the entire kernel source tree including object files. After that, we run another guest operating system on the snapshot logical volume and run the grep workload on its kernel source tree. For all these workload runs, we log the disk read and write requests and record the running time.

In the case of the new LVM2 scheme, the 15GB origin volume is sliced to provision for 6GB of free space for three snapshots between its slices, in the following way:

- Number of Slices = 30
- Size of Each slice = 15GB/30 = 512MB (128 extents)
- Interslice Gap = 6GB/30 = 200MB (50 extents)

The segment allocation size per allocation request is configured to be 2 extents (8MB) in the LVM2 allocation daemon.

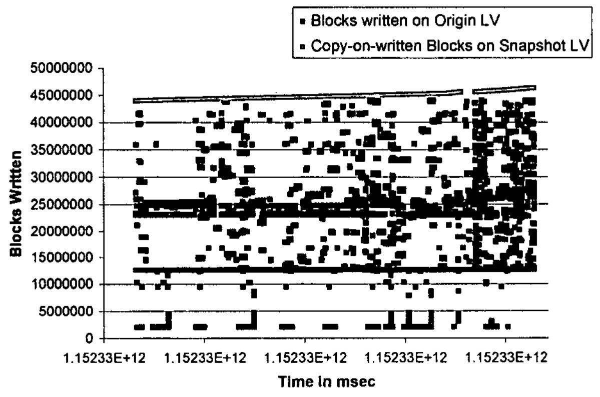
Features	Machine 1	Machine 2
CPU Speed	2.8GHz	3.2GHz
CPU L1 cache	8K	16K
CPU L2 cache	512K	1024K
RAM	512MB	1GB
Hard Disk Model	WD400BB-23FJA0, ATA	Maxtor 6Y080M0, ATA
Interface	EIDE	SATA
Rotational Speed	7200 RPM	7200 RPM
Hda max req. size	128KiB	128KiB
Hda capacity	40GB	80GB
Cache Buffer	2MB	8MB
Average Seek time	8.9ms(read seek time)	<= 9.3ms
Write seek time	10.9ms	-
Track to track see time	2ms	.9ms
Full Stroke seek	21ms	<= 20ms
Transfer rate (buffer to disk)	400 Mbit/s(max)	-
Transfer Rate (buffer to host)	100Mbit/sec	150Mbit/sec
Number of Platters	2	-
Number of Cylinders	-	158816
Data Zones per surface	-	16
Data Sectors /track	-	610/1102
Track Density per Inch	-	89 ktpi

Table 3.1: Comparison of Hardware Configuration of the two Machines (source: [6] and [11]

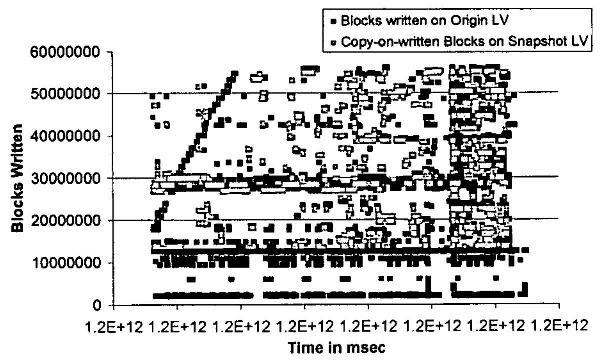
3.1.2 Observations

We have plotted the disk access graphs for the kernel build workload on the origin volume with the old and the new LVM2 scheme. These graphs are shown in Figure 3.1. The x-axis in these graphs represents the time during the compilation and the y-axis represents the disk blocks written during the kernel build on the origin volume. We observe in the disk access graph for the old LVM2 scheme that the blocks which are written to during the kernel build workload are copied from the origin volume to the snapshot volume in a linearly increasing disk segment at the top end of the graph. While for the new LVM2 scheme, we observe that the blocks are copied from the origin volume to the nearby snapshot segments on the disk. These segments are allocated on-demand at kernel build time by the LVM2 space allocation daemon.

Figure 3.2 shows the disk access graphs for the kernel grep workload on the origin and the snapshot volume with the old LVM2 scheme. Note that in case of kernel grep on the snapshot volume, the workload reads the unchanged data from the origin volume and the copy-on-written data from the snapsot volume. This causes disk seeks between the snapshot volume and the origin volume. Figure 3.3 shows the disk access graphs for the kernel grep workload on the origin and the snapshot volume with the new LVM2 scheme, where the two graphs look similiar to each other. The disk seeks in case of kernel grep on the snapshot volume are greatly reduced as the copy-on-written data lies adjacent to the unchanged data in this case. These observations are verified from the graphs in Figure 3.4 depicting the disk seek times as a function of disk seek distance during the grep workload on the snapshot volume with the old and new LVM2 scheme. The x-axis in these graphs denotes the seek distance which the disk travels for accessing blocks during the kernel grep workload and the y-axis denotes the corresponding time taken for these disk seeks. Comparing the two graphs in this figure, we note that the disk seek distances on the snapshot LV are reduced greatly with the new LVM2 scheme as compared to the old LVM2 scheme.

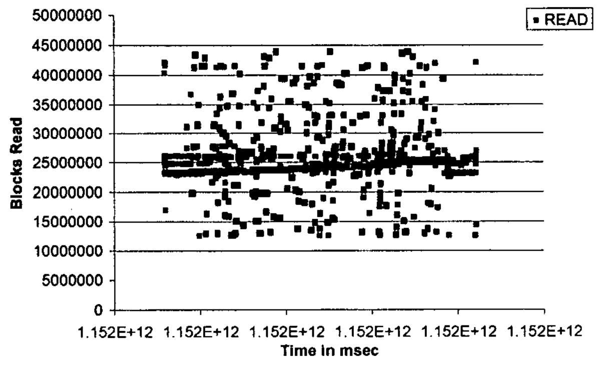


(a) Blocks Written during Kernel Build on the Origin Volume - Old LVM2 Scheme

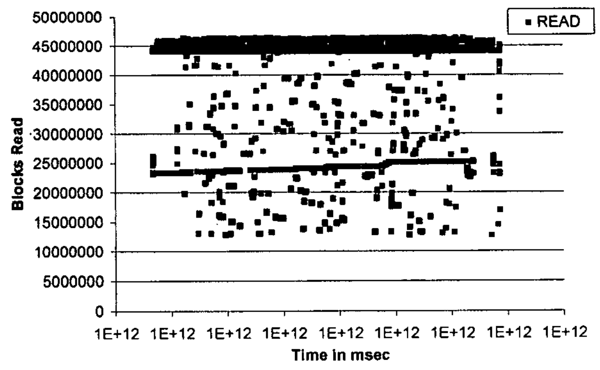


(b) Blocks Written during Kernel Build on the Origin Volume - New LVM2 Scheme

Figure 3.1: Blocks Written during 'Kernel-Build' on the Origin LV for the Old and New Schemes

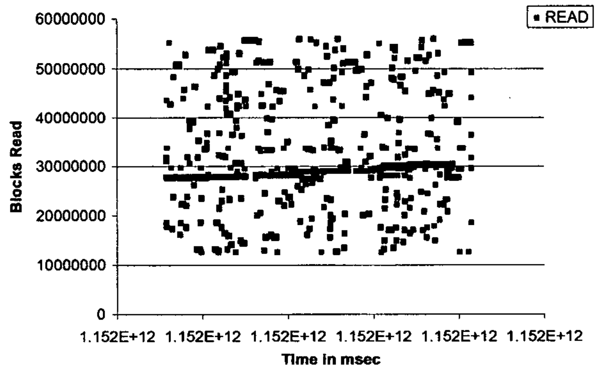


(a) Blocks Read during Kernel Grep on the Origin Volume - Old LVM2 Scheme

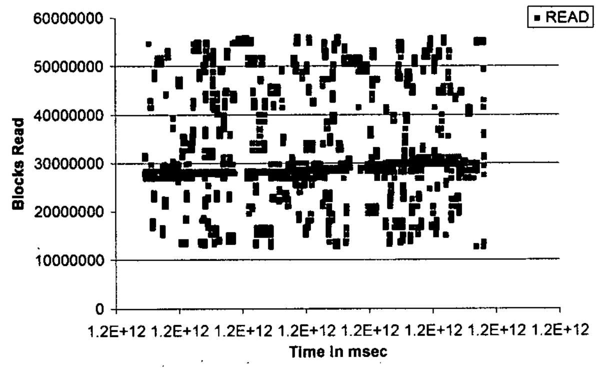


(b) Blocks Read during Kernel Grep on the Snapshot Volume - Old LVM2 Scheme

Figure 3.2: 'Kernel-Grep' Disk Accesses on the Origin and the Snapshot LV - Old LVM2 Scheme.

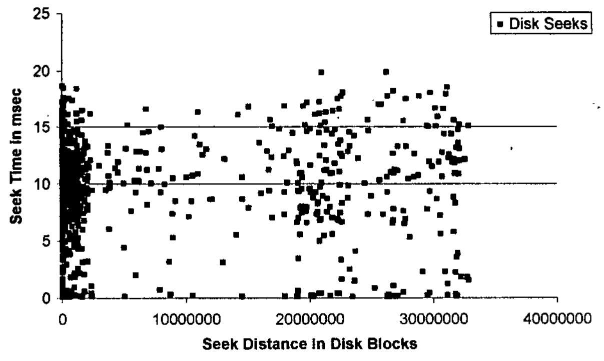


(a) Blocks Read during Kernel Grep on the Origin Volume - New LVM2 Scheme

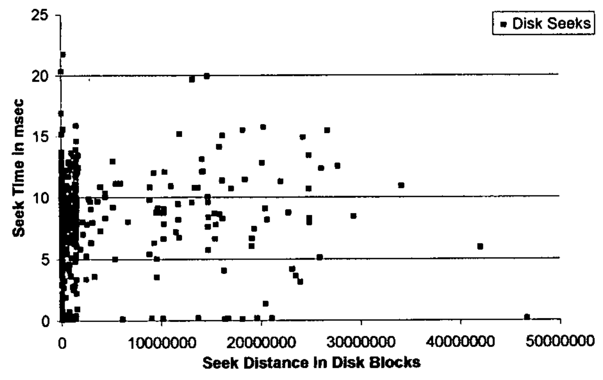


(b) Blocks Read during Kernel Grep on the Snapshot Volume - New LVM2 Scheme

Figure 3.3: 'Kernel-Grep' Disk Accesses on the Origin and the Snapshot LV - New LVM2 Scheme.



(a) Seek Profile during Kernel Grep on the Snapshot Volume - Old LVM2 Scheme



(b) Seek Profile during Kernel Grep on the Snapshot Volume - New LVM2 Scheme

Figure 3.4: 'Kernel-Grep' Seek Profile on the Snapshot LV for the Old and the New LVM2 Scheme.

3.1.3 Results

Table 3.2 shows the timing results for the experiment run on Machine 1. With the old LVM2 scheme, we observe 68% performance degradation for the kernel-grep workload on the snapshot volume as compared to the origin volume. Additionally, the extra copy-on-write chunk copies during the kernel build workload on the origin volume costs 8% performance degradation as compared to kernel build on the plain LV. With the new LVM2 schemes, we observe that the performance degradation in the kernel grep workload over the snapshot volume is 50% as compared to the kernel grep on the origin volume. while the percentage degradation in kernel build performance is the same as with the old LVM2 scheme. Overall, the new LVM2 scheme leads to an 18% improvement in read performance.

Workload (M/C #1)	Time (min:sec) OLD LVM2	Time (min:sec) NEW LVM2
Kernel build on Plain LV	42:59	42:39
Kernel grep on Plain LV	2:59	02:54
Kernel build on Origin LV	46:24	46:21
Kernel grep on Origin LV	03:00	02:58
Kernel grep on Snapshot LV	05:03	04:27

Table 3.2: Kernel Build and Grep Timings on the Plain, Origin and Snapshot Volumes for Old and New LVM2 - Machine 1

On comparing the kernel build time over the origin volume in case of the new LVM2 scheme with the old LVM2 scheme, we observe that the overhead of dynamic segment allocation in case of the new LVM2 scheme has no effect on the workload performance. Also, if we compare the kernel build time and grep time over the plain volume in case of the new LVM2 scheme with the old LVM2 scheme, we will see that the slicing of the origin volume in case of the new LVM2 scheme too has no noticeable impact on performance. Of course, these results are dependent on the degree of slicing in the origin volume, the free space provisioned between the slices, and the segment allocation unit for dynamic space allocation to the snapshot volume.

Table 3.3 shows the timing results for the experiment run on Machine 2. During this run, under the old LVM2 scheme, copy-on-write causes the kernel build time to increase by 26% and the kernel grep time by 15%. The new LVM2 scheme performs almost equally to the old LVM2 scheme. In this case the degradation in the snapshot volume's kernel grep performance due to copy-on-write is significantly less than in the previous case. We attribute this to the four times larger on-disk cache buffer. As shown in Table 3.1, the machine for this run has an 8MB on-disk cache buffer as compared to the 2MB cache buffer on the previous machine. As a result, the disk is able to cache more data and more effectively reduce number of disk accesses, which in turn improves read performance. The disk writes do not benefit from this cache buffer and therefore there is no improvement in the kernel build time which involves 43% writes.

Coming to the overall performance improvement for the kernel build and grep workload with the new LVM2 scheme, one may wonder if performance can be further improved by even more fine grained data placement. We can adjust the granularity of data placement by tuning the volume slicing and segment allocation parameters in the new LVM2 scheme, but it involves balancing some tradeoffs. For example, one may want to have thinner origin volume slices in order to reduce the distance between origin and snapshot data, but at the same time increasing the number of origin LV slices also implies increasing the number of gaps at the cost of their size thus leading to large number of dynamic allocations for the snapshot volume which could increase the write overhead on these volumes. In order to evaluate this aspect of the new LVM2 scheme, we conducted a series of experiments with increasing number of slicing while keeping the relative ratios of all other factors constant. These experiments are described in the next section.

Workload (M/C #2)	Time (hr:min:sec) OLD LVM2	Time (hr:min:sec) NEW LVM2
Kernel build on Plain LV	56:27	56:44
Kernel grep on Plain LV	10:16	10:23
Kernel build on Origin LV	1:11:15	1:12:9
Kernel grep on Origin LV	10:17	10:24
Kernel grep on Snapshot LV	11:53	12:15

Table 3.3: Kernel Build and Grep Timings on the Plain, Origin and Snapshot Volumes for Old and New LVM2 - Machine 2

3.2 Performance Impact of Origin LV Slicing

In this set of experiments, we aimed to explore the impact of slicing on the performance of the origin volumes and the snapshot volumes and see if there is any value in having more fine-grained slicing of the origin volume. In the last section we saw 18% performance improvement in the grep workload over the snapshot volume by slicing the 15GB origin volume into 30 slices. In these experiments we gradually increase the origin volume slicing from 30 to 8192 while keeping other factors constant and record their performance for the 'kernel compile and grep' workload. The configuration parameters for these experiments are shown in Table 3.4. In this table, we see that we have kept all other factors constant like the origin LV size, total space provisioning for the snapshot volumes, the snapshot LV size and the ratio of the gap between the slices to the dynamic allocation size.

During our experiments, we observe that increased slicing decreases the distance between the origin and the snapshot volume data blocks and increases kernel-grep per-

# of Origin LV Slices	Origin LV Size	Snapshot Provision	Extent Size	Origin LV Slice Size	Gap between Slices	Dynamic Allocation Size	Snapshot LV Size
30	15GB	6GB	4MB	512MB	196MB	8MB	2GB
512	15GB	6GB	2MB	30MB	12MB	4MB	2GB
1024	15GB	6GB	2MB	15MB	6MB	2MB	2GB
2048	15GB	6GB	512K	7.5MB	3MB	1MB	2GB
4096	15GB	6GB	256K	3.75MB	1.5MB	512K	2GB
8192	15GB	6GB	128K	1.875MB	.75MB	256K	2GB

Table 3.4: Configuration Parameters for the Slicing Experiments

formance on the snapshot volumes. But at the same time, the overhead of dynamically allocating space to the snapshot volumes increases during the kernel-build workload as the amount of space allocated per request decreases with increased slicing thereby increasing the number of allocation requests. These trends are depcited in the Figures 3.5 and 3.6.

The graph in Figure 3.5 shows the trend in kernel grep time on the origin volume and the snapshot volume with an increasing number of origin volume slices. The 'origin LV grep' data points in the graph depict the kernel grep time on the origin volume for different number of origin volume slices. We observe that there is not much variation in the kernel grep time over the origin volume as the number of slices are increased. The 'snapshot LV grep' data points in the graph depict the kernel grep time on the snapshot volume for different number of origin volume slices. The percentage value printed above each data point denotes the percentage by which the kernel grep workload takes more time on the snapshot volume than on the origin volume. We see that the kernel grep time on the snapshot volume shows a decreasing trend as the number of origin volume slices are increased. The percentage gap between the kernel grep time on the origin and the snapshot volume decreases from 68% in case of one slice(equivalent to

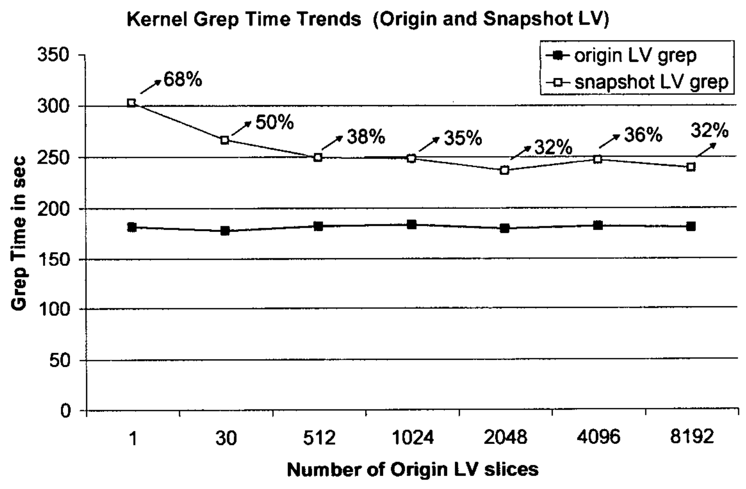


Figure 3.5: Trends in Kernel Grep Time on Origin and Snapshot LVs with increasing number of slices

the old LVM2 scheme) to approximately 35% with more slices. Thus we see that the idea of slicing up the origin volume and carefully placing snapshot data in between these slices does prove beneficial in improving the read performance over the snapshot volume without sacrificing the origin volume's read performance.

The graph in Figure 3.6 shows the trend in kernel build time on the plain and the origin volume with an increasing number of origin volume slices. The 'Plain LV Build' data points in the graph depict the kernel build time on the plain sliced volume and the 'Origin LV Build' data points depict the kernel build time on the origin sliced volume for different numbers of slices. The percentage value printed above each 'Origin LV Build' data point depicts the percentage by which the kernel build workload takes more time on the origin volume than on the plain volume. We observe an increasing trend in the kernel build time on the origin volume with increasing numbers of slices. This is because of the increasing overhead of dynamic space allocation for the snapshot volume during the kernel build on the origin volume. In this case, this overhead almost

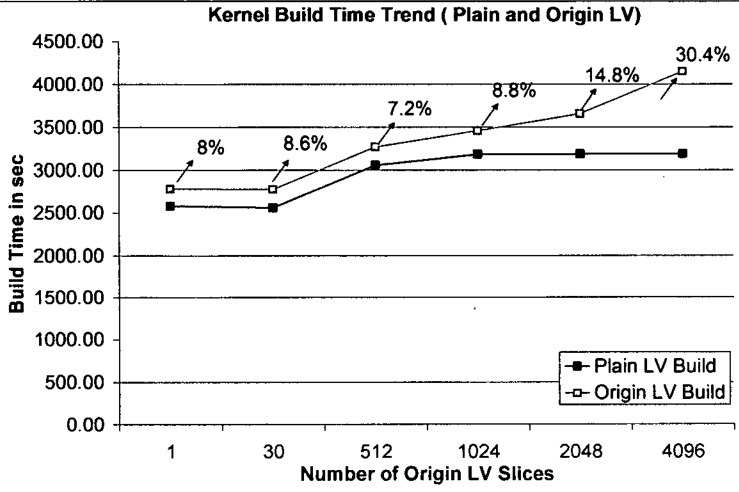


Figure 3.6: Trends in Kernel Build Time on Origin LV with increasing number of slices

starting doubling after a point as the amount of space allocated per dynamic allocation request is halved.

Thus, we observe that as we go on slicing the origin volume more and more, the snapshot volume blocks get further closer to their original locations, thereby improving the snapshot read performance. But, as the slicing is increased beyond a point, the overhead of dynamic space allocation for snapshot volumes becomes considerable.

3.3 Partial Filesystem Rewrite Workload

This is an artifical workload which ages the origin volume by writing random parts of the filesystem in order to generate copy-on-write data and then greps on the filesystem to measure the read performance of the snapshot volume. It differs from the kernel compile and grep workload in the sense that it does not allocate new data blocks on the origin volume but rewrites the existing data blocks.

3.3.1 Workload Description

In this workload, first a script is run which finds all the files present on the origin volume and writes this list into a file. Then a snapshot of the origin volume is taken. After that, another script is run on the origin volume which randomly picks files from the above list and partially rewrites them. During this rewriting process, the rewritten disk blocks get copied from the origin volume to the snapshot volume. The files are rewritten partially in order to generate copy-on-data which is related to unchanged file data. For the rewriting, the script starts from the beginning of each file, skipping few blocks of data in the process periodically till it reaches the end of the file. For our experiments, we have configured the script to skip 8K of data periodically. Once this script is over, a grep is run on the snapshot volume's filesystem and the origin volume's filesystem and time recorded. In order to maintain uniformity across the two workload runs, the list of random files picked up for partial rewriting in the second step is kept same for the two runs.

3.3.2 Experiment Setup

For our experiment, we performed two runs of the above workload once using the old LVM2 scheme and once the new LVM2 scheme. In both the cases, same machine was used(Machine 1 in Table 3.1). The configuration parameters for the two runs are shown in Table 3.5.

3.3.3 Results

The results of the above experiment runs are shown in Table 3.6. We see in the table that while the new LVM2 scheme takes 22% more time to partially rewrite the filesystem than the old LVM2 scheme, it fairs better in the read performance of the snapshot volume by almost 40%. With the old LVM2 scheme, the snapshot volume's read performance is 84% degraded as compared to the origin volume. On the other hand, the new LVM2 scheme brings down this degradation in the read performance to 45%.

We have plotted the disk accesses and seek profile of the filesystem grep workload for the old and the new LVM2 scheme in order to understand what the disk is doing

Parameter	Old LVM2	New LVM2
Origin Volume Size	15GB	15GB
Snapshot Volume Size	2GB	2GB
Provision for Snapshots	-	2GB
# of Slices	-	2048
Slice size	-	7.5MB
Slice Gap	-	1MB
Dynamic Allocation Size	-	1MB

Table 3.5: Configuration Parameters for the Partial Filesystem Rewrite Experiments

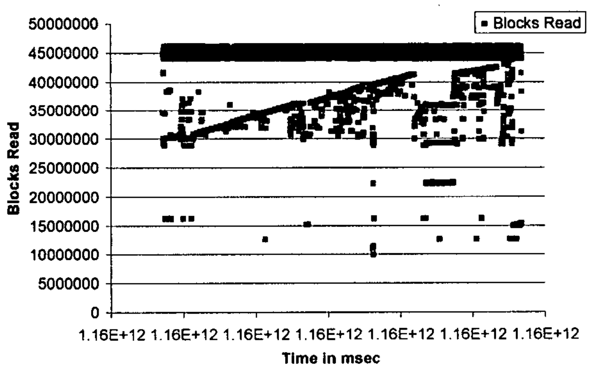
Workload (M/C #1)	Time (min:sec) OLD LVM2	Time (min:sec) NEW LVM2
Partial File system Rewrite on Origin LV	19:40	24:02
Grep on Origin LV's File System	04:29	04:26
Grep on Snapshot LV's File System	08:14	06:26

Table 3.6: Results for the Partial Filesystem Rewrite Experiments - Old and New LVM2

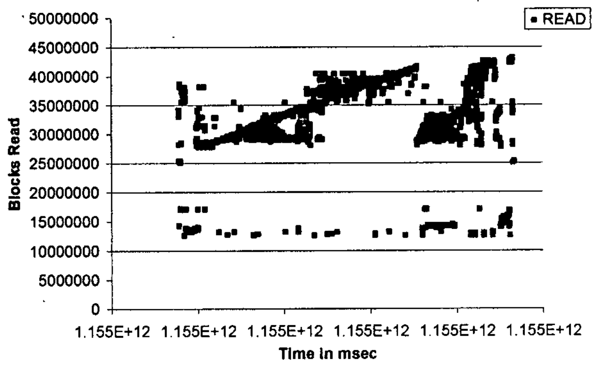
during this workload. Figure 3.7 shows the disk blocks read with time during the file system grep for the old and the new LVM2 scheme. For the old LVM2 scheme we observe that the copy-on-written blocks lie in a dense horizontal band on the top belonging to the snapshot volume while the unchanged data blocks lie below in the origin volume. For the new LVM2 scheme, the copy-on-written blocks occupy space in the snapshot segments which lie near their original locations in the origin volume slices.

Figure 3.8 shows the disk seek profile of the filesystem grep workload for the old and the new LVM2 scheme. The x-axis in these graphs denotes the seek distances which the disk travels to access the data blocks during the grep and the y-axis denotes the corresponding disk seek time for these accesses. We can clearly see in these graphs that the seeks are greatly reduced in case of the new LVM2 scheme as compared to the old LVM2 scheme.

In this chapter, we presented the results of our experiments which compared the new LVM2 design with the old one. We evaluated the new scheme with two different workloads and on two different machines and found out that it performs better than the old scheme by 18% to 40% depending upon the configuration parameters like the degree of origin volume slicing, etc. Although it incurs extra overhead during first time writes due to its dynamic space allocation policy, the read performance of the snapshot logical volumes with the new scheme can be improved considerably while keeping this overhead low.

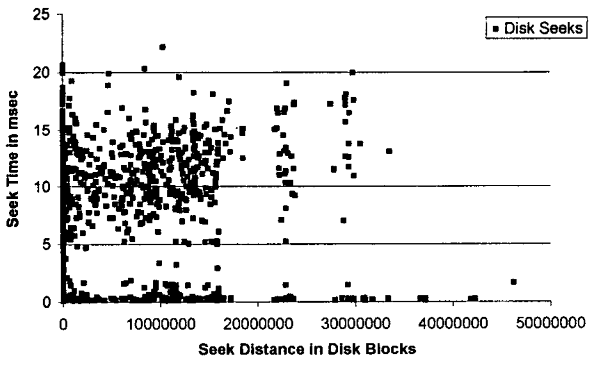


(a) Blocks Read during File System Grep on the Snapshot Volume - Old LVM2 Scheme

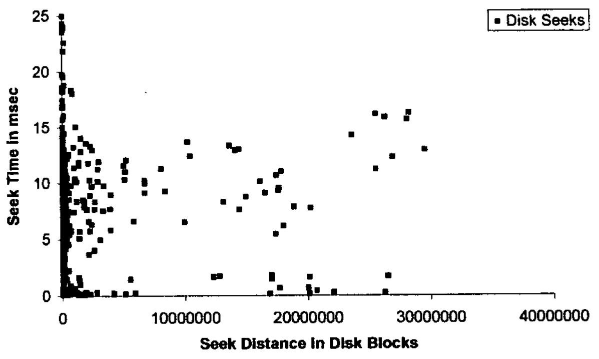


(b) Blocks Read during File System Grep on the Snapshot Volume - New LVM2 Scheme

Figure 3.7: Disk Accesses during File System Grep on the Snapshot LV for the Old and the New LVM2 Scheme.



(a) Seek Profile during File System Grep on the Snapshot Volume - Old LVM2 Scheme



(b) Seek Profile during File System Grep on the Snapshot Volume - New LVM2 Scheme

Figure 3.8: Disk Seek Profile for the File System Grep on the Snapshot LV for the Old and the New LVM2 Scheme.

Chapter 4

Related Work

Data Snapshot Technology is a key component of commerical-grade storage products and has been implemented by most hardware and software storage vendors today. In Section 4.1, we discuss some of the commerical and research-based snapshot technology solutions, focusing on the copy-on-write aspect of their implementations and the related I/O performance issues. Section 4.2 provides a survey of previous research work on improving performance of copy-on-write data. In this section, we particularily focus on the data placement techniques explored by other researchers.

4.1 Survey of Snapshot Technologies

Most of the today's storage systems provide some kind of snapshotting facility to capture data as it appears at some point of time and use it for backup, recovery and other purposes like data mining and data cloning. A snapshot can be defined as a consistent point-in-time image of data. Depending on the way snapshots are created, they can be classified along several lines.

One of the important feature of snapshots is their writability. Some storage systems support only read-only snapshots which are mainly used for backup and error recovery purposes. Other systems provide the facility to create writable snapshots which can support live applications running on them. One such use of writable snapshots is in the OS-virtualized environments where a snapshot is created from a base image and then mounted to support live filesystems of individual guest operating systems.

Another way in which snapshots may differ is the manner in which they are created. A snapshot can be either a full-copy of the original data or a log of changes on the original data since the time of the snapshot creation. While the full-copy method removes any dependency on original data and reduces recovery time, its preparation time and disk space requirement is proportional to the original data size. Long creation time implies equivalent down-time for applications while the snapshot is being taken. On the other hand, the differential log method utilizes copy-on-write technique to generate snapshots instantaneously while using minimal disk space. In the copy-on-write technique, the snapshot initially points to the original data. Whenever some data is changed in the original data store, it is first copied to the snapshot area to preserve data as it appeared at the snapshot creation time. Thus, the copy-on-write snapshot requires only a fraction of the original datastore disk space size to store the changed data blocks. With all these benefits of disk space savings, zero application down-time, and instantaneous snapshot generation, copy-on-write techniques have been widely used in snapshot technology design. We shall see some of these designs in the subsequent subsections.

Besides the above mentioned classifying features, snapshots can also be differentiated based on the level of storage hierarchy at which they are created, the granularity of data which they represent and the frequency with which they are generated. At the application level, software version control systems, like CVS [3] and RCS [21], provide a facility to create and manage various point-in-time versions of software. They give user the flexibility to choose whichever parts of the file system need to be versioned. Also, they allow users to concurrently checkout old versions and make changes to them and return them back to the repository.

At the file system level, the versioning semantics may be implemented inside the file system. While several file systems like Plan9 [23], Network Appliance's 'Write Anywhere File Layout' file system [12], and Ext3cow [5] allow periodic snapshots of entire file systems, others like Elephant [22], VersionFS [15], WayBack [2], Comprehensive Versioning File System CVFS [8] provide individual file and directory versioning. Filesystem-based snapshot systems understand filesystem semantics and thus enable greater control and finer granularity in the snapshotting process. For instance, the Elephant [22] file system implements support for user-specified retention policies for individual files, groups of files or directories. Also, since most of these systems are implemented inside the file systems, they provide snapshot feature as an easy extension to the existing file system interface.

On the other hand, there are systems which implement a snapshot facility at the storage block virtualization level. The advantage of this approach is that such mechanisms provide a common, filesystem-agnostic snapshot facility which can be transparently used by all higher system layers. By moving the data management functionality near the disk, such systems reduce the complexity of higher layers and take advantage of powerful processing capabilities of the disk subsystems. However, since information about the content of data is not available at this level, most block-level snapshot systems provide snapshotting of entire volumes and not individual files or directories. Examples include volume management systems like Logical Volume Manager [13], Parallax [1], Petal [20] and block-level versioning systems like Clotho [4] and Peabody [10].

In the following sections we shall discuss various data snapshot systems, primarily focusing on the algorithms, datastructures and data placement mechanisms implemented by them.

4.1.1 Versioning File Systems

At the file system level, we can find a wide variety of snapshot and versioning systems designed with a varying set of goals, priorities and requirements. In these systems, the snapshot granularity varies from single file versions to entire filesystem snapshots. Nevertheless, saving disk space in storing various different versions of filesystem data is one of the primary concerns for all these systems. So, most of these systems use block-level copy-on-write techniques to replicate metadata and data blocks for different versions or snapshots. Some systems, like CVFS [8], go further in saving space for metadata replication by maintaining metadata versions in a log or in a multi-version btree. Other systems have to optimize for their use-specific requirements like Network Appliance's WAFL file system [12], which is optimized for writes for providing fast NFS service.

The problem of arranging copy-on-write data of versioned filesystems on the disk which optimizes read performance has largely been ignored or not investigated. Copy-on-write versioning inherently destroys contiguity of metadata and data blocks in cases where file versions, which share some blocks but not all of their blocks, not all versions can be laid out contiguously on the disk. Let us look at some of the snapshotting and file versioning system designs and understand how they optimize their data placement on the disk.

Write Anywhere File Layout(WAFL): Network appliance's Write Anywhere File Layout file system [12] is specifically designed for an NFS filserver appliance. It provides online read-only snapshots of entire file systems. WAFL's design has been optimized for write-performance. They adopt a write anywhere design in which all metadata is kept in files, which can be allocated anywhere on the disk. This design choice gives more flexibility in write allocation policies as metadata and data can be arranged on the disk more creatively. WAFL uses non-volatile RAM to collect write requests and send them to the disk in one 'write episode'. This not only reduces the response time of write requests but also allows WAFL to do block allocation for a large number of requests at once. WAFL implements snapshots at the entire file system level. It maintains a 32-bit entry for each 4K block, with each bit indicating if the block is being used by the corresponding snapshot. Whenever a disk block is updated, WAFL makes a copy of it and its metadata and updates the corresponding block mappings of the active file system. In summary, WAFL, by virtue of its write-anywhere design and ability to schedule writes in bulk, allows for a wide variety of intelligent block placement strategies. But, no effort has been made to co-locate newly written copy-on-write data with its previous versions in case of WAFL.

Elephant File System: The Elephant [22] file system works like a version control system, storing all version of files or group of files automatically and managing their storage based on user-specified retention policies. It differs from checkpointing systems like WAFL, as described above, in the sense that entire filesystems need not be versioned, instead users can specify versioning at the level of individual files. In Elephant, a file can have multiple inodes, one for each of its versions. These inodes are stored in an inode log indexed by the time each inode in the log was closed. Elephant does allow some flexibility in the location and size of the inode log by introducing a level of indirection from inode number to the inode log, but it is not clear how it manages the physical organisation of various versions of files on the disk. For systems like Elephant, which maintain different versions of files in a copy-on-write fashion on the disk, it is important to prevent disk fragmentation for performance reasons.

Ext3cow File System: The ext3cow filesystem [5] provides both filesystem-level snapshots and file-level versioning. It implements these features through copy-on-write of file system blocks and inodes on the disk. Different versions of files have their own copy of the inode and are identified by the 'epoch' in which they were created. Inodes belonging to different versions of a file are arranged as a chain of inodes with the most recent file version's inode heading the list. The ext3cow paper [5] mentions the problem of optimizing read performance in versioning file systems. It identifies that duplicating inodes for file versions reduces efficacy of inode clustering and block-level copy-on-write destroys contiguity. The authors of this paper have proposed the concept of Virtual Contiguity [19] [18] in which related blocks belonging to different versions of the same file are kept as near as possible on the disk to prevent long seeks between them. We shall discuss their work on virtual contiguity in Section 4.2.

4.1.2 Block-level Snapshot Systems

Block-level snapshot systems work on the principle of virtualizing the target disk and exposing logical disks to the higher layer. Internally, they manage the block mappings for different versions of the logical disk or volume. Most of them employ copy-on-write techniques to cut down on the disk space required to store different data versions. Some systems, like Peabody [10], which perform fine-grained data versioning also perform content-based block coalescing to save space. The major advantage of moving snapshot functionality to the block layer is that it provides a common, filesystem-independent mechanism of versioning and backing up data. In the following paragraphs, we shall briefly describe the design of some block-based versioning systems.

Petal Storage System: Petal [20] is a distributed storage management system which manages a pool of physical disks in a way which exposes an easy to manage, highly available block-level storage system consisting of virtual disks. By virtualizing the physical disk resources, Petal is able to provide transparent component and site failure recovery, re-configuration, load-balancing and backup. Petal's virtual disk provides 64-bit byte storage space and is allocated disk space on demand. Petal translates the client-supplied virtual disk block addresses into physical disk addresses by using a set of local and global address mapping datastructures. It provides read-only snapshots of virtual disk in a copy-on-write fashion and identifies each snapshot by the 'epoch' in which it was created. The placement of virtual disk blocks on the underlying physical disks is governed by the redundancy requirements of the virtual disk as specified by the user, and the load-balancing algorithms in Petal which aim at equally distributing storage load on all physical disks. Clearly, in the Petal system's design, requirements of high-availability and load-balancing due to the distributed nature of the system take precedence in deciding data organisation on the physical disks.

Parallax storage system: The Parallax storage system [1] is a distributed storage system designed to manage storage for live and dormant virtual machine (VM) images in large cluster environments. Like Petal [20], Parallax servers manage the disks of the cluster machines and present an abstraction of virtual disks to their clients (the virtual machines). A virtual disk, in case of Parallax, represents both the current state and the set of read-only snapshot images of a virtual machine. The mapping from virtual disk block number to physical address is stored in a radix-tree datastructure which represents the copy-on-write block sharing between the various snapshots of the virtual machine. Similiar to Petal, a Parallax virtual disk is accessible from any node in the cluster and is replicated for high availability and failure-protection. On each machine in the cluster runs a 'Parallax server VM', which manages local disk, servicing virtual disk requests coming from host VMs, and participating in the distributed sharing and replication schemes. From the block placement point of view, each Parallax VM explicitly manages block allocation on its local disk for its volumes but also contributes a part of its disk space to host blocks belonging to volumes from other cluster machines. Parallax design does not make any effort to manage physical block placement for snapshot virtual disks and here too, similiar to Petal, the data replication requirements across cluster machines would affect any such considerations.

Clotho : Clotho [4] is a block-level volume versioning system, similiar to volume managers in design. It provides read-only snapshots of data volumes each identified by the timestamp at which it was created. In order to reduce the size of the metadata holding mappings from logical addresses to physical disk addresses, Clotho uses extent size, which is bigger than single block size, as a unit of mapping. At volume initialization time, Clotho partitions the volume capacity into two logical segments: a primary data segment and a backup data segment, and reports the volume size as the size of primary data segment to the higher layer. This partitoning is only logical and is actually enforced by Clotho by partitioning its metadata table into two segments: a primary extent mapping table and a backup extent mapping table. When the backup mapping table becomes full, Clotho simply returns an error to its higher layer, in which case, the higher layer has to reclaim, compact or move some of the backup versions. Clotho maintains the version history of each extent as a chained list of mapping entries in the extent mapping table. The most recent version of an extent has its mapping in the primary extent mapping table. Inorder to access subsequent versions of the extent, Clotho needs to follow the chain of mapping entries till it reaches the right entry with the corresponding version timestamp. New extents, whether for current or archive data, are allocated on the target volume device by the extent allocator, which follows a sequential disk allocation policy, in which extents are allocated sequentially from the beginning of the volume till its end. Any extents freed on the disk shall be scanned in the next pass of the disk by the extent allocator. Thus, on the target volume device, extents belonging to the current and backup versions shall be arranged in a mixed way.

Peabody: Peabody [10] is a network block storage device that exposes virtual disks. It provides an 'undo' mechanism at the block-level to recover any previous state of the exposed virtual disks. For doing so, it logs the content of all the writes to the virtual disk blocks in a write-log and records metadata for writes in a transaction-log. The metadata includes the location being written and the time stamp which can be used in rolling back to a particular point in time. Recovery is managed by Peabody's virtual disk manager, Sherman, which allocates a new virtual disk and traverses the write log in reverse order, writing the old contents back to the virtual disk. Since Peabody records the content of all writes going to the virtual disks, it needs to have some mechanism to minimize storage consumption. Peabody authors found out through experiments that there is considerable amount of identical content sharing between blocks within same virtual disk and across different virtual disks. Based on all these findings and the requirement to reduce space consumption, Peabody allows for coalescing of identical content blocks. Preliminary experiments with the Peabody prototype show 20% lower bandwidth for both read and write requests than the normal iSCSI target. This reduction in throughput is because of the overhead of fine-grained write logging and the processing overhead of content-based block coalescing logic.

Peabody [10] attempts to combine both content-based block coalescing and fine-grained write logging to achieve a balance between space requirements and fine-grained data recovery. In the Peabody paper, the authors explore different strategies to implement the on-disk datastructures for write logging and content-sharing. The run-time performance of read and write requests on Peabody virtual disks and the rollover performance shall depend on these implementation choices.

4.2 Disk Block Placement Schemes for Copy-on-Write Snapshot Systems

In the last section, we saw that storage snapshot technology has been extensively researched and applied both at the logical file system level [5] [22] [12] and the physical block level [4] [20] [1] [13] with varying granularity. It is used to provide entire volume as well as file-level versioning. Although a lot of documentation is available on the design and implementation of these techniques, there has been lesser emphasis on analyzing the effects of copy-on-write on the I/O performance of such systems. In most cases, the problem of efficiently laying down data on the disk, in the face of the fragmentation caused by copy-on-write, has largely been ignored or superseded by other concerns.

We could only find the 'Virtual Contiguity' work done by Randal et. al. [19] [18], which deals with this problem at the file system level. In the following paragraphs, we shall present the concept of 'Virtual Contiguity', their design, and important results and conclusions drawn by their work.

Concept of Virtual Contiguity: The idea of Virtual Contiguity is to relax the requirement of strict physical contiguity of data blocks and to allocate them in dense regions on the disk so that they are close enough to be read in a single disk head movement, but at the same time leaving space between them for reallocation. By breaking a file into densely allocated segments, virtual contiguity attempts to reduce the number of disk seeks required to read the file. And by leaving space within these segments, the copy-on-written file blocks can be placed near their original allocation, thereby keeping data close and maintaining read performance for copy-on-write data.

They adopt a randomized dynamic storage allocation policy which instead of consuming space sequentially for initial file allocations, randomly selects a start offset and begins searching forward from that offset for a contiguous allocation. By doing so, they hope to keep the density of allocated blocks uniform across the disk.

Results and Conclusions: In his work [18], Zachary found out that although a very small percentage of total files are copy-on-written, such files fragment heavily across the disk, and that the read performance of these files is important because a large part of these files belong to the active working set of files. He also found out that traditional file system allocation schemes, like the Next-Fit and Best-Fit, fail to maintain the contiguity of copy-on-written files. Even in non-cow file systems, Zachary found out that almost 20% of all writes have to be reallocated because they are unable to grow in place.

In the Virtual Contiguity work [19] [18], the authors note that the benefits of co-locating overwritten blocks near the original allocations for copy-on-written files get nullified by the losing of spatial inter-file and inter-segment locality. In their results, they discover that the random selection of start offset for segment allocation, instead of resulting in a uniform density of allocated blocks, causes hotspots to be created, which exhibit a high degree of variance in region density. Also, it affects the spatial locality of related files, whose segments are initially allocated randomly on the disk. Thus, Virtual Contiguity scheme results in poorer disk performance as compared to the standard schemes like Best-Fit and Next-Fit.

Comparison with Our Work: We share most of the Virtual Contiguity ideas in our research but at a different plane. We adopt the basic idea of Virtual Contiguity, that is to initially allocate the blocks in dense groups with some space left between them for receiving the copy-on-written blocks. That way, the snapshot volume blocks would reside near the origin volume blocks to which they are related. But, we also take note of the failure of the random initial allocation scheme in the Virtual Contiguity work, and therefore allocate space to the origin volume in configurable fixed size slices with configurable fixed space between them. However, the fixed scheme has its own disadvatanges as it is difficult to anticipate the amount of copy-on-written data which will be generated per origin volume slice for different workloads. We have discussed the tradeoffs of our scheme in Section 2.3.

Coming to the differences between the two schemes, the virtual contiguity work is applied at the file system allocation level, where the system has semantic understanding of file data and metadata. Whereas, our research focuses on volume-level copy-on-write snapshots at the block level, a level which is file-system agnostic and unaware of any associations between the data which is stored in the blocks. So, the task of co-locating related blocks in our case is a bit more challenging as it becomes more difficult to justify meaningful relation between copy-on-written blocks at the block level. The only intuition which we have at the block level is that the block which is written to is somehow related to its neighbouring blocks. So, the reallocation for this block should be done near its original location in order to preserve the spatial proximity of the reallocated block with its related blocks.

In this chapter, we provided a survey of various data snapshot technologies implemented at the file system level and the disk block level. We discussed their space allocation and block placement policies for copy-on-write data. Then we described the Virtual Contiguity work and drew some similarities and differences with our work.

Chapter 5

Conclusion

In this thesis, we investigated the disk space allocation, data placement and disk I/O performance of LVM2 logical volumes. We found out that the disk I/O performance of the LVM2 snapshot volumes is gravely effected by the out-of-place data writing due to copy-on-write. While some degree of performance degradation is expected for the first time writes on the origin or the snapshot volume after the creation of the snapshot volume due to the copying of data from the origin to the snapshot volume, it is the read performance of the snapshot volume which is a subject of concern. Because of the discontiguity created in the physical placement of the snapshot volume blocks due to copy-on-write, the disk has to make long seeks while reading data on the snapshot volume. Our experiments verified this phenomenon in case of LVM2, showing that these disk seeks lead to significant lowering of the read performance of snapshot volumes as compared to that of plain volumes. The performance of snapshot volumes is an important issue in OS-virtualized environments where such volumes are used to support live applications running inside Guest virtual machines. Even in case of systems which apply copy-on-write at other data planes, for example filesystems which provide versioning of files and directories (refer Section 4.1), the time taken to access snapshotted data can be a crucial performance factor for purposes like data mining etc.

In order to reduce the physical distance between the snapshot data blocks, we experimented with the physical placement of this data on the disk. We changed the way LVM2 allocated space to the origin volume such that it leaves free space between the origin volume segments so that copy-on-written snapshot data can be written near its original location on the origin volume. In order to ensure that the snapshot volume data is allocated the closest avaliable free space near its original location, we replaced the static space allocation scheme of LVM2 with a dynamic one which looks for an appropriate location to copy data on the snapshot volume at run-time. This new LVM2 scheme is more deterministic in placing snapshot volume data near the origin volume data as compared to the original LVM scheme which allocates space to the origin and the snapshot volume independently.

From our experiments with the new LVM2 scheme, we found out that carefully placing snapshot data near the original data does give us performance improvement by reducing the seek distance between the snapshot and the origin data blocks. This performance gain increases with increased origin volume slicing as it ensures more fine-grained snapshot data block placement. Thus, our work clearly shows that the idea of placing copy-on-written data blocks near their original locations holds value and can lead to disk I/O performance gain for the snapshot volumes.

In our implementation, we adopted a dynamic space allocation scheme for the snapshot volumes in order to seek maximum benefit at run time in co-locating the related blocks. Dynamically allocating space to the snapshot volume at runtime keeps disk I/O requests waiting till the time space is allocated, thereby adding delay in the I/O processing path. This leads to performance overhead during the first time writes to the origin or snapshot volumes. This overhead becomes significant when the dynamic allcoation requests become more frequent (as we saw in Section 3.2). With our scheme, we need to balance the need to co-locate data finely and the need to reduce the processing delays caused by this dynamic allocation. We do observe that it is possible to attain such a balance with our scheme but it is difficult to decide the suitable configuration values like the number of origin volume slices, dynamic segment allocation size etc., which lead to an optimal balance.

In our experiments with machines having disks with large cache buffers, we observed that such disks can cache multiple different tracks simultaneously and can thus improve the read performance of physically scattered data on the disk. In this case, we observed that the degradation in snapshot volume's performance as compared to the origin volume performance is reduced to a small percentage due to these caching effects. Thus the snapshot volumes with old LVM2 scheme perform well and those with the new LVM2 scheme also perform equivalent to the origin volume.

In summary, we studied the disk placement and I/O performance of copy-on-written data in case of LVM2 and implemented an alternative scheme to co-locate related on-disk data for improving performance. We noted some performance overheads in the first time writes with our scheme, but also saw the new scheme perform better than the old LVM2 implementation by 18% to 40%. We believe that this idea of co-locating copy-on-written with its original location on the disk holds value and can be taken further to improve the performance of data snapshot systems. We would like to explore the possibilities of applying this idea to other block-based data snapshotting systems in future.

Glossary

Chunk : A chunk is a contiguous collection of disk blocks. It is the amount of data copied from a origin logical volume to a snapshot logical volume during first time writes., 14

Device Mapper Library: A device mapper library provides a programming interface to the user-space applications for accessing the device mapper's ioctls., 10

Device Mapper: A device mapper is a kernel module which maintains the mapping tables for logical volumes and maps their I/O requests., 9

Error Mapping: In Error Mapping the disk I/Os directed to the logical volume are failed by the device mapper., 11

Linear Mapping: In Linear Mapping the logical extents are mapped sequentially to extents on a physical volume., 11

Logical Extent: A logical extent is the basic unit of space allocation for logical volumes and is equal to the volume group's physical extent size., 5

Logical Segment: A Logical Segment is a collection of contiguous logical extents., 7

Logical Volume (LV): A Logical Volume is an abstraction of a physical hard disk. This abstraction may either consist of a portion of the physical drive, known as a partition, or it may consist of a set of disks, such as a RAID volume or array., 5

LVM2 Mapping Table: An LVM2 mapping table maps a logical volume's logical segments to physical segments., 9

Mapping Type: A mapping type defines how the blocks from a logical segment will be mapped to blocks in a physical target segment., 9

Origin Logical Volume: An origin logical volume is a volume which has one or more snapshot logical volumes based on it., 12

Physical Extent: A physical extent is the quantum of storage space that LVM uses when sizing logical volumes., 5

Physical Segment: A Physical Segment is a collection of contiguous physical extents on a disk., 7

Physical Volume (PV): A hard disk or a hard disk partition which has been prepared to be used by LVM2., 5

Slicing: A logical volume is made up of multiple equal sized and equally spaced physical segments., 29

Snapshot Logical Volume: A snapshot logical volume is a volume which maintains volume data frozen at some point of time., 12

Volume Group (VG): A Volume Group (VG) is the highest level abstraction used within the LVM. It gathers together a collection of Logical Volumes (LV) and Physical Volumes (PV) into one administrative unit., 5

Bibliography

[1] R. Ross K. Fraser C. Limpach A. Warfield and S. Hand. Parallax: Managing storage for a million machines. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS-X)*, 2005.

[2] B. Cornell And, P. Dinda, and F. Wayback: A user-level versioning file system for linux. *CORNELL, B., DINDA,P.,AND BUSTAMANTE, F. Wayback: A user-level versioning file system for linux. In Proceedings of USENIX 2004.*, 2004.

[3] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 341–352, 1990.

[4] Michail D. Flouris Bilas and Angelos. Clotho: Transparent data versioning at the block i/o level.

[5] Zachary N. J. Peterson Burns and Randal C. Ext3cow: The design, implementation, and analysis of metadata for a time-shifting file system.

[6] Maxtor Corporation. Diamondmax plus9 60/80/120/160/200gb at product manual, 2003.

[7] Marshall K. McKusick Fabry, William N. Joy, Samuel J. Leffler, and Robert S. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[8] C. Soules Ganger, G. Goodson, J. Strunk, and G. Metadata efficiency in versioning file systems. *Conference on File and Storage Technologies (San Francisco, CA, 31 March–02 April 2003.*, 2003.

[9] A. Whitaker Gribble, M. Shaw, and S. Denali: Lightweight virtual machines for distributed and networked applications. *A. Whitaker, M. Shaw, and S. D. Gribble.*

Denali: Lightweight virtual machines for distributed and networked applications. In Proceedings of the USENIX Annual Technical Conference, Monterey, CA, June 2002. 10, 2002.

[10] Charles B. Morrey III Grunwald and Dirk. Peabody: The time travelling disk. In *IEEE Symposium on Mass Storage Systems*, pages 241–253, 2003.

[11] Western Digital Technologies Inc. Wd caviar eide hard drives specification sheet, Feb 2006.

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

[13] Heinz Mauelshagen. Linux volume manager, 2001.

[14] Heinz Mauelshagen. Logical volume manager(lvm2), 23 Sep, 2004.

[15] Kiran-Kumar Muniswamy-Reddy. VERSIONFS: A versitile and user-oriented versioning file system. 2003.

[16] B. Dragovic Neugebauer, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Xen and the art of virtualization.

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

[18] Z. Peterson. Data placement for copy-on-write using virtual contiguity. *Z. N. J. Peterson. Data placement for copy-on-write using virtual contiguity. Master's thesis, University of California, Santa Cruz, September 2002.*, 2002.

[19] Randal Burns Robert. Allocation and data placement using virtual contiguity.

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

[21] Walter F. Tichy. RCS — a system for version control. *Software — Practice and Experience*, 15(7):637–654, 1985.

[22] Douglas J. Santry Veitch, Michael J. Feeley, Norman C. Hutchinson, and Alistair C. Elephant: The file system that never forgets. In *Workshop on Hot Topics in Operating Systems*, pages 2–7, 1999.

[23] Rob Pike Winterbottom, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.