Metadata services for the Parallax storage system

by

Gitika Aggarwal

B.Sc., The International Institute of Information Technology, 2004

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

July, 2008

Abstract

Parallax is a distributed storage system that uses virtualization to provide storage facilities specifically for virtual environments. In Parallax, fragmentation occurs when the block addresses visible to the guest virtual machine are sequentially placed, but the corresponding physical addresses are not. Because of the copy-on-write (CoW) nature of Parallax, as virtual disks are created, cloned, deleted, snapshotted and migrated, some fragmentation of the physical media can occur, potentially incurring seeks even when performing sequential accesses to the virtual disk. As the storage pool ages, performance issues due to unchecked fragmentation, unreclaimed storage space and duplicate data can cause significant concern. CoW snapshots also introduce sharing semantics between virtual disks and snapshots. The ability to create CoW clones of virtual disks from snapshots of other virtual disks leads to more sharing relationships. As a result block reclamation and allocation become non-trivial.

We have developed utilities for garbage collecting, de-fragmenting free disk space and virtual disks and reclaiming duplicate read-only blocks in the storage pool managed by Parallax. They work by updating and maintaining the metadata structures related to each virtual disk and its snapshots. They use very coarse grained locking on the metadata and work at the block level. They operate across the storage pool and are agnostic to the operating systems and file systems used by the virtual machines.

Table of Contents

List of Figures

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Norm Hutchinson for his guidance and kindness. I want to thank Norm for all his patience and for being my mentor. I would also like to thank my co-supervisor, Dr Andrew Warfield. It was an honour to work with him. Without Andy's brilliant ideas it is hard to imagine how this thesis would have shaped up. I would also like to thank Dutch, Geoffrey and Brendan. They have all helped me on a number of occasions when I needed help with my thesis work. I would like to thank all the members of the DSG for their constructive suggestions. I thank all my friends at the University of British Columbia, who made my stay here really memorable. Most importantly, I want to thank my family. They have been the guiding light in my life all along and my inspiration to pursue graduate studies. Their support, encouragement and wisdom have always helped me face my roadblocks. I can never thank them enough.

Chapter 1

Introduction

Abstracting storage for easier administration is a very powerful concept. Storage virtualization pools physical storage from multiple network or local storage devices into a single storage service that can be managed centrally. Typical functions include: adding and removing component storage systems, carving out logical partitions, resizing them, deleting them and performing storage-wide administrative jobs like consistency checks, snapshots, mirroring, replication and the like. As the number of features offered by a storage virtualization solution increases, so does the size (and complexity) of the metadata that is required to support the virtualization. In particular, the capability to create snapshots and clones can significantly complicate the metadata.

Parallax [16] is a distributed storage system that uses virtualization to provide storage facilities specifically for virtual environments. It uses copy-on-write(CoW) to snapshot and clone virtual disks. These CoW snapshots not only fragment the storage system, but also introduce sharing semantics between disk blocks. The ability to create CoW clones of disk objects from snapshots of other disk objects leads to more sharing relationships. As a result, operations like disk-space allocation and reclamation become non-trivial. As the storage pool ages, performance issues due to unchecked fragmentation, unreclaimed storage space and duplicate data can cause significant concern.

I have developed utilities for garbage collecting, de-fragmenting free disk space and reclaiming duplicate read-only blocks (de-duping) in the storage pool (blockstore) managed by Parallax. These utilities are provided at the block level so as to remove the overhead of managing them at a per virtual-machine level. They work across the blockstore and are agnostic to the operating systems and file systems used by the virtual machines.The remainder of this chapter briefly presents each of these three issues (garbage collection, defragmentation, deduping), and summarizes my work.

1.1 Garbage collection

A Garbage Collector (GC) is a process that reclaims disk blocks that are no longer referenced and will never be accessed or mutated again. Analogous GCs are also used in memory management. Depending on how the referenced blocks are tracked by the storage system, an appropriate GC can be designed. Broadly, there are two main approaches. One is the reference counting method and the other is the mark and sweep method. Almost all schemes are variations of these two approaches. A GC is generally invoked explicitly, either when the system is low on disk space or periodically.

We should first define reachability of a disk object [23]. A disk object can be an entity like a file, directory or even whole/part of a virtual disk. They can also hold pointers (and some metadata) to other disk objects. Examples of such disk objects are directories, virtual disks, clusters of virtual Disks, etc. A subset of disk objects is assumed to be reachable. These are called the Root objects. Any disk objects reachable from one of these roots are also reachable. Transitively, any disk objects reachable from another reachable disk object are also reachable.

In the *Reference Counting* method, disk objects also maintain a count of the number of references to the disk blocks/objects. When the reference count falls to zero, the object has become unreachable and can be freed. Obviously, maintaining these counts incurs extra storage overhead and updating them at every disk object allocation and deletion operation incurs performance overhead. It becomes especially expensive when disk objects are shared and/or when they are related to each other and are represented by graphs or hierarchies. This is because when disk objects are shared, there could be several processes competing to modify the value of the reference counter. In such situations, locking mechanisms are required for consistency which makes the process complicated and more expensive.

In the *Mark and Sweep* method, the most naive approach is to visit all disk objects starting from the root objects and keep track of all disk blocks that are reachable. The first stage of collection (mark phase) traverses all root objects, marking each accessible object as being "in-use". All objects transitively accessible from the root set are marked, as well. Thus, the disk is examined again and again until all disk objects/blocks have been marked. Finally, in the sweep phase, each object is again examined; those disk blocks that are not marked as reachable are freed. There are two disadvantages of this approach. The first is that it requires that while the GC is running, the process of creating new disk objects be halted. The second disadvantage is that it can require arbitrarily many passes over the disk, depending on the longest chain of pointers among disk objects.

However, depending on how the metadata is organised on disk, the mark and sweep method can be optimized to reduce the performance overhead and make

it more efficient. For example, one could use coloring schemes to differentiate allocated blocks, unallocated blocks and blocks whose status could have changed while the GC was executing. Then there would be no need to halt the system to execute the GC. The only side-effect to this scheme is that some unallocated blocks could escape being marked (blocks that became unreferenced while the GC was executing).

Besides reclaiming unused space, GCs can also perform several useful operations. Since they scan the metadata of the entire storage system, given sufficient additional information, they can also perform defragmentation tasks. In the absence of any additional information, they can at least trivially remap pages to create large contiguous pools of free space that would help with block allocation. If provided with enough semantic information, they can remap blocks of a file/virtual disk, so that they lie close to each other. One could also equip a GC to do sanity checks for the disk by calculating checksums of blocks and comparing them with already stored values and silently correcting the data from backups or replicas. These *smart GCs* are thus not limited to only free disk space management.

The GC for Parallax is an example of such a smart GC. It not only reclaims deleted blocks, but also helps with defragmenting virtual disks, with remapping pages to create large pools of free disk space and with deduping read-only blocks of data.

1.2 De-fragmentation

File-system fragmentation is not a new problem at all. As a file-system ages, it becomes progressively harder to write files sequentially , because the allocation and deletion of variable-sized data objects results in small regions of free space being spread across the surface of a disk, where it is preferable to have a single, continuous region of free space. File fragmentation increases disk head movement which makes disk I/O slower. Almost all file systems provide a utility to de-fragment the disk. These defrag utilities generally have a two-fold purpose. One is to rearrange the disk pages such that pages of a single large file are closer to each other on disk (read and in-place write locality) and the second is to merge small areas of free space to form larger pools of free space (for faster block allocation and sequential writes).

Parallax has a more complicated fragmentation problem. Parallax inserts an address translation mechanism below the file systems created by its clients. Hence, file-systems are only aware of the file layouts in terms of logical block numbers. Making file defragmentation decisions based on only logical block numbers could be totally misguided, because logical block numbers would have little semblance

with the actual layout of the files on disk. Added to that, blocks from files from one virtual disk could be interspersed with blocks from files of other virtual disks. This worsens the fragmentation problem.

Parallax also has the problem of metadata fragmentation. Parallax aims at easy extensibility, i.e., one should be able to add or remove physical disks or any other form of block storage easily. It also supports very large sizes of files/virtual disks and their numbers can be in the millions, with versioning, snapshots etc. To support all these features, considerable metadata needs to be maintained. This metadata cannot be stored in a small fixed area on disk. In fact, it is hard to predict how much disk space will be sufficient. Hence, it stores its metadata in an on-disk R-tree so that metadata blocks can be allocated from anywhere on the storage-system. A lot of other metadata is also distributed throughout the storage pool. However, this brings in the problem of metadata fragmentation.

A third kind of fragmentation is *related file fragmentation* [22]. It is caused by a lack of locality between related files and depends on the access pattern of specific applications. Basically, if related files are spaced out from each other, then there could be scenarios where in an application fetches a page from file1, then another one from file2 and then again file1 and so on. This would cause the disk head to move back and forth a lot.

A Parallax instance serving several virtual disks that are physically spaced out is very susceptible to related file fragmentation. Hence, we have written a defrag utility for Parallax that computes a remapping scheme so that disk blocks of popular virtual disks are laid out sequentially on disk. We call these specially laid out virtual disks *superpages*. A superpage can also be just a part of the virtual disk, instead of the whole disk. Details on superpages are discussed in Section 3.3. The defrag utility also merges small pools of free disk space and remaps interspersed allocated pages to create large pools of free disk space which can be used to create superpages, speed up block allocation and have more sequential write operations.

1.3 De-duping

Duplicate data due to frequent backups, file distribution over the network through emails, popular downloads, etc., can end up consuming enormous storage space that could be used otherwise. Not only does this incur increased cost in terms of disk to store the data, but also incurs additional costs in terms of data center space and power.

The idea behind de-duping is very simple. When writing out data to disk, if it is deemed as already present on-disk, then instead of writing it again, leave a pointer to the already present data. The duplicates can either be detected in-line

(i.e., before they have been written on disk) or in a post-process (i.e., sometime after the data has already been stored). Recently, a lot of work has surfaced in the area of de-duping. Generally, one of two approaches are taken : de-duping at a block level or de-duping at a file level. File level de-duping aims to remove duplicates of files either within a system or across the data-center itself. Another approach is to remove duplicate files in backups, for example, operating system files like Windows' .dll files. This could decrease the size of disk-based backups immensely.

Block-level deduplication works under the file-system. It computes a fingerprint of each block of data and if it finds two blocks with the same fingerprint then one of the duplicates is reclaimed. To avoid collision errors, generally the data is compared byte-by-byte before a duplicate is deleted. An important factor in de-duping is CPU usage. If the volume/ storage system that is being de-duped is constantly mutating, then the process of computing the fingerprints could cause some performance overhead. Also, if the blocks being reclaimed are still writable, then de-duping them would probably be of little value, because any in-place writes would trigger far more additional disk operations.

We have designed a post-process block level deduper for Parallax that remaps read-only pages across the blockstore. Our usage scenarios encourage the usage of gold masters and snapshots. With de-duping enabled, once VDIs branch off from a gold-master, any similar changes to these clones can be easily reclaimed after the first snapshot operation itself. This can lead to significant cost savings. De-duping can also reclaim any other duplicate read-only pages in the storage pool. A specific example of de-duping in a virtual environment is lots of common images (imagine 10,000 windows desktops) that have the same patch applied to them. Deduping the physical storage should be able to reclaim all duplicate copies of the patch.

1.4 Summary

We have designed and implemented storage reclamation and remapping techniques for the Parallax storage system. Using these techniques we have developed utilities to defragment free disk space and create superpages for better disk space allocation and seek time. The rest of the thesis is organised as follows. Chapter 2 provides background information about Parallax. Chapter 3 gives a detailed account of the design changes required in Parallax to implement block reclamation and remapping tooks. Chapter 4 gives the implementation details and evaluation of these utilities. Chapter 5 is a survey of how these tools have been implemented in several other storage systems. Chapter 6 includes some proposed future work and concludes this thesis.

Chapter 2

Background - Parallax

2.1 Introduction

This chapter gives some relevant information about Parallax. It provides the required context for the thesis and is central to understanding the issues addressed by the remapping operations. It includes a brief explanation of the design princples of Parallax and describes the system structure, the metadata and the data layout. At the end of this chapter, one should have a fairly good idea of how Parallax manages its metadata.

2.2 Parallax

Parallax provides block virtualization by interposing between individual virtual machines and the physical storage layer. The virtualized environment allows the storage virtualization service to be physically co-located with its clients. Figure 2.1 presents a high-level view of the structure of a parallax-based cluster: the storage system runs in an isolated VM on each host and is administratively separate from the client VMs running alongside it; effectively, Parallax allows the storage system to be pushed out to include slices of each machine that uses it.

The next section describes the set of specific design considerations that have guided the implementation of Parallax and its associated services, and then present an overview of the system's structure.

2.2.1 Design considerations

Designing a system to provide VM-specific functionality involved a series of decisions that shaped the resulting implementation:

Agnosticism and Isolation. Parallax is implemented as a collaborative set of storage *appliances*; as shown in Figure 2.1, each physical host in a cluster contains a *storage VM* which is responsible for providing storage to other virtual machines running on that host. This VM isolates storage management and delivery to a single container that is administratively separate from the rest of the system. This design

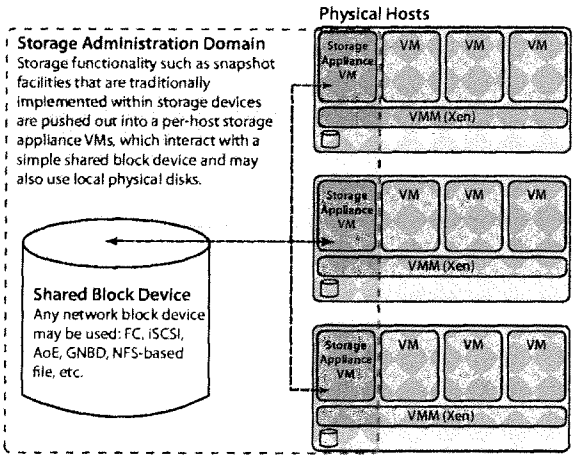

Figure 2.1: Parallax is designed as a set of per-host storage appliances that share access to a common block device, and present virtual disks to client VMs.

has been used previously to insulate running VMs from device driver crashes [5, 13], allowing drivers to be transparently restarted. Parallax takes this approach a step further to isolate storage virtualization in addition to driver code.

Isolating storage virtualization to individual per-host VMs results in a system that is agnostic to both the OSes that run in other VMs on the host, and the physical storage that backs VM data. A single cluster-wide administrator is capable of managing the parallax instances on each host, unifying the storage management role.

Blocks not Files. In keeping with the goal of remaining agnostic to OSes running within individual VMs, Parallax operates at the block, rather than file-system, level. Block-level virtualization provides a narrow interface to storage, and allows Parallax to present simple virtual disks to individual VMs. While virtualization at the block level maximizes agnosticity and simplifies implementation, it also presents a set of challenges: the "semantic gap" introduced by virtualizing the system at a low level obscures higher-level information that could aid in identifying opportunities for sharing, and complicates request dependency analysis for the disk scheduler.

Minimize the DLM. Parallax's design is premised on the idea that data sharing in a cluster environment should be provided by application-level services with clearly defined APIs, where concurrency and conflicts may be managed with application semantics in mind. Therefore, it *explicitly excludes* support for write-sharing of individual virtual disk images. The system maintains the invariant that each VDI

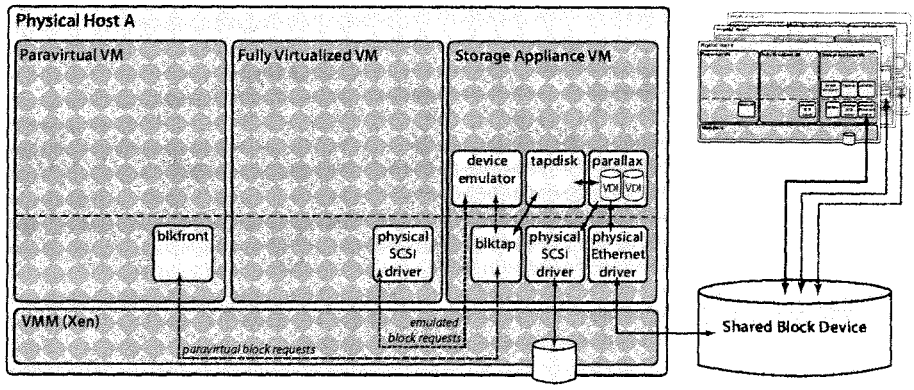

Figure 2.2: Overview of the Parallax system architecture.

has at most one writer, greatly reducing the need for concurrency control. Some degree of concurrency management is still required, but only when performing administrative operations such as creating new VDIs, and in very coarse-grained (multi-gigabyte) allocations of writable areas on disk. Locking operations are explicitly not required as part of the normal data path or for snapshot operations.

Snapshots as a primitive operation. Parallax has been designed to incorporate snapshots from the ground up, representing each virtual disk as a set of radix-tree based block mappings that may be chained together as a potentially infinite series of copy-on-write (CoW) instances.

2.2.2 System structure

Figure 2.2 shows a detailed overview of Parallax's architecture and allows a brief discussion of the relevant components that are presented in more detail in later sections.

As discussed above, each physical host in the cluster contains a storage appliance VM that is responsible for mediating accesses to an underlying block storage device by presenting individual virtual disks to other VMs running on the host. This storage VM allows a single, cluster-wide administrative domain, and effectively allows functionality that is currently implemented within filers, arrays, and storage switches to be pushed out and implemented on individual hosts. The result is that advanced storage features, such as Parallax's snapshot facilities, may be implemented in software and provided to the cluster over simple, inexpensive network storage.

Parallax itself runs as a user-level daemon in the Storage Appliance VM, and

uses Xen's *block tap* driver [20] to handle block requests. The block tap driver provides a very efficient interface for forwarding block requests from VMs to daemon processes that run in user space of the storage appliance VM. The user space portion of block tap defines an asynchronous disk interface and spawns a `tapdisk` process when a new VM disk is connected. Parallax is implemented as a tapdisk library, and acts as a single block virtualization service for all client VMs on the physical host.

Each Parallax instance shares access to a single shared block device. There are no restrictions as to what this device need be, so long as it is shareable and accessible as a block target in all storage VM instances. In practice we most often target iSCSI devices, but other device types work equally well. We have chosen that approach as it requires the lowest common denominator of shared storage, and allows Parallax to provide VM storage on the broadest possible set of targets.

Virtual machines that interact with Parallax are presented with entire virtual disks. Xen allows disks to be accessed using both emulated and paravirtualized interfaces. In the case of emulation, requests are handled by a device emulator that presents an IDE controller to the client VM. Emulated devices generally have poor performance, due to the context switching required to emulate individual accesses to device I/O memory. For performance, clients may install paravirtual device drivers, which are written specifically for Xen-based VMs and allow a fast, shared-memory transport on which batches of block requests may be efficiently forwarded. By presenting virtual disks over traditional block device interfaces as a storage primitive to VMs, Parallax supports any OS capable of running on the virtualized platform, meeting the goal of agnosticism.

The storage VM is connected directly to physical device hardware for block and network access.

Virtual disk images (VDIs) are the core abstraction provided by Parallax to virtual machines. A VDI is a single-writer virtual disk which may be accessed in a location-transparent manner from any of the physical hosts in the Parallax cluster. There are three core operations, allowing VDIs to be created, deleted, and snapshot. These are the only operations required to actively manage VDIs: once created, they may be attached to VMs as would any other block device.

2.2.3 VDIs as block address spaces

A Parallax VDI is effectively a single *block* address space, represented by a radix tree that maps virtual block addresses to physical block addresses. Virtual addresses are a continuous range from zero to the size of the virtual disk, while physical addresses reflect the actual location of a block on the shared blockstore. The current Parallax implementation maps virtual addresses using 4K blocks, which are

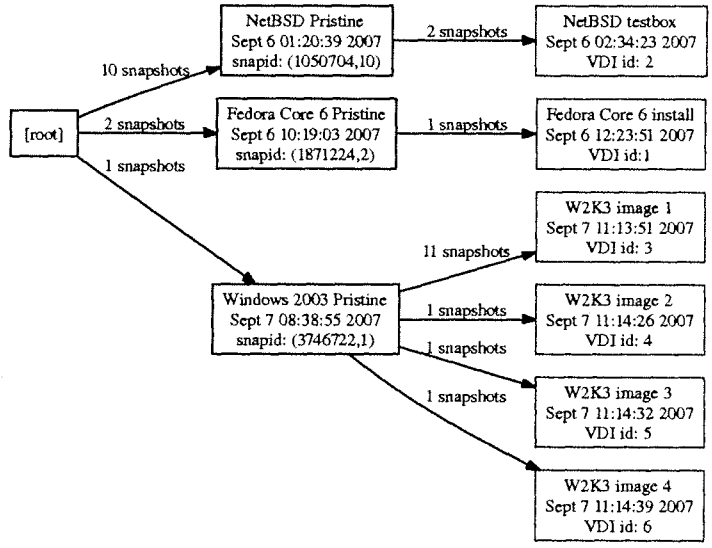

Figure 2.3: VDI tree view—visualizing the snapshot log

chosen to intentionally match block sizes used on x86 OS implementations. Mappings are stored in 3-level radix trees, also based on 4K blocks. Each of the radix metadata pages stores 512 64-bit global block address pointers, and the high-order bit is used to indicate that a link is read-only. This layout results in a maximum VDI size of 512GB (9 address bits per tree-level * 3 levels * 4K data blocks = $2^9 * 3 * 2^{12} = 2^{39} = 512$GB). Adding a level to the radix tree extends this by a factor of 2^9 to 256TB, involves a small configurable change to the code, and has a negligible effect on performance for small volumes (less than 512GB) as only one additional metadata node per active VDI need be cached. Parallax's address spaces are sparse. Zeroed addresses indicate that the range of the tree beyond the specified link is non-existent and must be allocated. In this manner, the creation of new VDIs involves the allocation of only a single, zeroed, root block. Parallax will then populate both data and metadata blocks as they are written to the disk. In addition to sparseness, references are shared across descendant radix trees in order to implement snapshots.

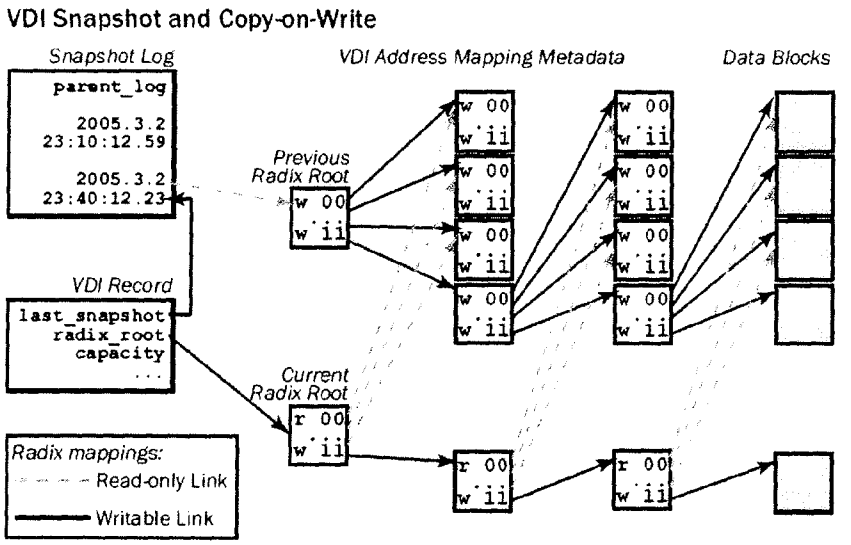

Figure 2.4: Parallax radix tree (simplified with short addresses) and COW behavior.

2.2.4 Snapshots

As mentioned, the high-order bit of block addresses in the radix tree is used to indicate that the block pointed to is read-only. All VDI mappings are traversed from a given radix root down the tree, and a read-only link indicates that the entire subtree is read-only. To take a snapshot, Parallax simply copies the root block of the radix tree and marks all of its references as read-only.

This is illustrated in Figure 2.4. The figure shows a simplified radix tree mapping six-bit block addresses with two address bits per radix page. The example shows a VDI that has had a snapshot taken, and subsequently had a block of data written at virtual block address `111111` *(binary)*. The snapshot operation copies the radix tree root block and redirects the VDI record to point to the new root. All of the links from the new root are made read-only, as indicated by the "`r`" flags and the dashed grey arrows in the diagram.

Copying a radix tree block always involves marking all links from that block as read-only. A snapshot is completed using one such block copy operation, following which the VM continues to run using the new radix tree root. At this point, data writes may not be applied in-place as there is not a direct path of writable links from the root to any data block. The write operation shown in the figure copies

every radix tree block along the path from the root to the data (two blocks in this example) and the newly-copied branch of the radix tree is linked to a freshly allocated data block. All links to newly allocated (or copied) blocks are writable links, allowing successive writes to the same or nearby data blocks to proceed with in-place modification of the radix tree. The active VDI that results is a copy-on-write version of the previous snapshot.

The address of the old radix root is appended, along with the current timestamp, to a *snapshot log*. The snapshot log represents a history of all of a given VDI's snapshots. Parallax enforces that radix roots refered to by snaplogs are immutable. However, they may be used as a reference to create a new VDI. The common approach to interacting with a snapshot is to create a writeable VDI clone from it and to interact with that. A VM's snapshot log represents a chain of dependent images from the current writable state of the VDI, back to an initial VDI. When a new VDI is created from an existing snapshot, its snapshot log is made to link back the the snapshot on which it is based. Therefore, the set of all snapshot logs in the system form a forest, linking all of the radix roots for all VDIs, which is what Parallax's VDI tree operation generates, as shown in Figure 2.3. This aggregate snaplog tree is not explicitly represented, but may be composed by walking individual logs backwards from all writable VDI roots.

From a single-host perspective, the VDI and its associated radix mapping tree and snaphot logs are largely sufficient for Parallax to operate. However, these structures present several interesting challenges that are addressed in the following sections: Section 2.3 explains how the shared block device is managed to allow multiple per-host Parallax instances to concurrently access data without conflicts or excessive locking complexity.

2.3 The shared blockstore

A major challenge in VM-based systems is the tendency of virtualization to *increase* the exposure of VMs to system failure. In Parallax, distributed locking has been avoidedwherever possible, with the intention that even in the face of disconnection[1] or failure, individual Parallax nodes should be able to continue to function for a reasonable period of time, while an administrator resolves the problem. This approach has guided the management of the shared blockstore both in terms of how data is laid out on disk, and where locking is required.

[1]This refers to disconnection from other hosts. A connection to the actual shared blockstore is still required to make forward progress.

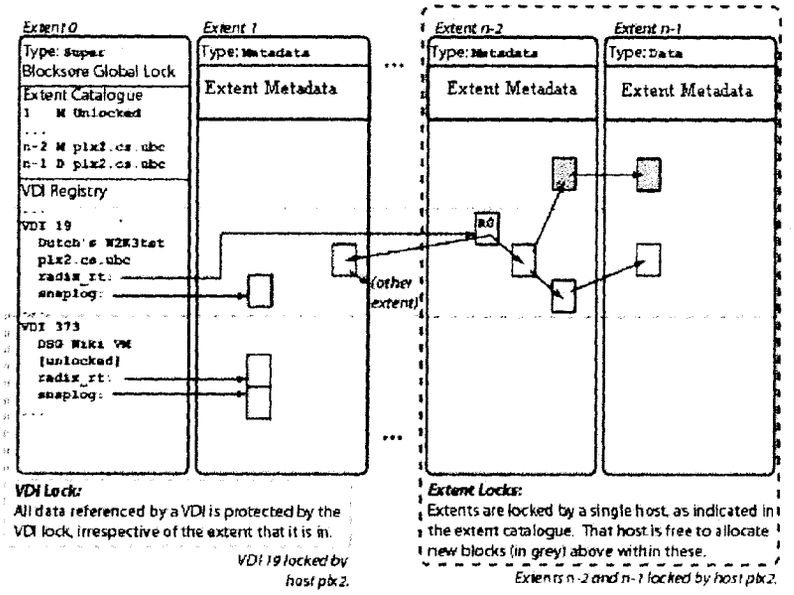

Figure 2.5: Blockstore layout.

2.3.1 Extent-based access

The physical blockstore is divided, at start of day, into fixed-size extents. These extents are reasonably large, 2GB in the current implementation, and represent a lockable single-writer region. "Writers" at the blockstore level are physical hosts—Parallax instances—rather than the consumers of individual VDIs. These extents are typed; with the exception of a special system extent at the start of the blockstore, extents either contain data or metadata. Data extents hold the actual data written by VMs to VDIs, while metadata extents hold radix tree blocks and snapshot logs. This division of extent content is made to clearly identify metadata, which facilitates garbage collection. Both data and metadata extents start with an allocation bitmap that indicates which blocks are in use.

When a Parallax-based host attaches to the blockstore, it will exclusively lock a data and a metadata extent for its use. At this point, it is free to modify unallocated regions of the extent with no additional locking.[2] In order to survive disconnection from the lock manager, Parallax nodes may lock additional unused extents to allow

[2]This is a white lie – there is a lock on the allocation bitmaps to coordinate with the garbage collector, see Section 5.6.

room for additional allocation beyond the capacity of active extents.

The system extent at the front of the blockstore contains a small number of blockstore-wide data structures. In addition to system-wide parameters, like the size of the blockstore and the size of extents, it has a catalogue of all fixed-size extents in the system, their type (system, data, metadata, and unused), and their current lock-holder. It also contains the VDI registry, a tree of VDI structs, each stored in an individual block, describing all active VDIs in the system. VDIs also contain persistent lock fields and may be locked by individual Parallax instances. Locking a VDI struct provides two capabilities: First, the locker is free to write data within the VDI struct, as is required when taking a snapshot where the radix root address must be updated. Second, with the VDI struct locked, a Parallax instance is allowed to issue in-place writes to *any* blocks, data or metadata, referenced as writable through the VDI's radix root.

Figure 2.5 illustrates the structure of Parallax's blockstore, and demonstrates how extent locks allow a host to act as a single writer for new allocations within a given extent, while VDI locks allow access to allocated VDI blocks across all extents on the blockstore. All extents, except the system extent, maintain extent-metadata at the head of the extent. Various metadata services use the extent metadata, which is maintained accordingly. The block allocator (Section 5.2), uses the block allocation map (BMap). The BMap is a bitmap that maintains information about whether a block in the extent is allocated or not. A BMap can also be modified by the garbage collector. The garbage collector unsets bit positions corresponding to blocks that are deemed as deleted. Hence access to the BMap is regulated by a lockmaster as explained in Section 2.3.2 .

2.3.2 Lock management

The protocols and data structures in Parallax have been carefully designed to minimize the need for coordination. Locking is required only for infrequent operations: to claim an extent from which to allocate new data blocks, to gain write access to an inactive VDI, or to create or delete VDIs. Unless an extent has exhausted its free space, no VDI read, write, or snapshot operation requires any coordination at all.

The VDI and extent locks work in tandem to ensure that the VDI owner can safely write to the VDI irrespective of its physical location in the cluster, even if the VDI owner migrates from one host to another while running. The Parallax instance that holds the VDI lock is free to write to existing writable blocks in that VDI on *any* extent on the shared blockstore. Writes that require allocations, such as writes to read-only or sparse regions of a VDI's address space, are allocated within the extents that the Parallax instance has locked. As a VM moves across hosts in the

cluster, its VDI is managed by different Parallax instances. The only effect of this movement is that new blocks will be allocated across multiple extents.

Because it is unnecessary for data access, the lock manager can be very simple. We call this as the lockmaster. In the current implementation, a single node is designated as the lockmaster. When the lockmaster process instantiates, it writes its address into the special extent at the start of the blockstore, and other nodes use this address to contact the lockmaster with lock requests for extents or VDIs. Failure recovery is not currently automated, but the system's tolerance for lockmaster failure makes makes manual recovery feasible.

2.4 Summary

We now have a good idea of how the blockstore is laid out and how Parallax manages it. This was a description of Parallax in its pristine state before any block reclamation and remapping services were added to it. In the next chapter we will motivate the need for these services and see how they have been implemented.

Chapter 3

Motivation

3.1 Introduction

This chapter explains the motivation to develop block reclamation and remapping mechanisms for Parallax. In Parallax, fragmentation occurs when the block addresses visible to the guest virtual machine are sequentially placed, but the corresponding physical addresses are not. Because of the copy-on-write (CoW) nature of Parallax, as virtual disks are created, cloned, deleted, snapshotted and migrated, some fragmentation of the physical media will occur, potentially incurring seeks even when performing sequential accesses to the virtual disk. CoW snapshots also introduce sharing semantics between virtual disks and snapshots. The ability to create CoW clones of virtual disks from snapshots of other virtual disks leads to more sharing relationships. As a result block reclamation and remapping become necessary and non-trivial.

3.2 Tendency of Parallax to fragment data

Parallax has a tendency to fragment VDIs for reasons we explain in the remainder of this section. As explained in Section 2.3.1, a Parallax instance uses separate extents to allocate the data and metadata pages respectively. There are several observations to be made here. Firstly, separating the data extent from the metadata extent avoids interspersing metadata blocks within data blocks in the blockstore and thus reduces metadata fragmentation. Metadata fragmentation is not fully eliminated though. Initially, when an extent is all empty and is newly allocated as a metadata extent, it is very likely that radix tree pages of a VDI will get allocated close to each other in the extent. Every Parallax instance maintains a cache of metadata pages called the RadixNodeCache. This cache stores all radix tree pages that were read in from the blockstore to serve I/O requests of the VMs running on the Parallax instance. Since it is known that the metadata is localized in a particular extent, one can take advantage of that and do some read-ahead to improve I/O performance. The clean separation of the location of metadata pages from the data pages is very useful to tasks like garbage collection and deduping. One can easily browse

through the entire metadata of a Parallax cluster by simple sequential passes of the superblock and the metadata extents. It is often argued that storage systems that offer storage virtualization face I/O streams from a number of clients and hence, any benefit that one might get from locality is mostly lost. It is generally up to the disk scheduler, that given a list of I/O requests, it should be able to come up with a sequence that will cause minimum disk head seek. However, we argue that having locality is capable of making a considerable difference, especially when one is doing sequential scans. One can then take advantage of read ahead policies that pre-fetch metadata blocks and that will speed up the Parallax utilities (GC, remapper etc.) requesting these pages.

As the blockstore ages, several things can happen:

1) Parallax instances could be restarted. Every time a Parallax instance is restarted, it can end up with a different data and metadata extent, which will result in fragmenting the data as well as metadata pages of the VDIs.

2) When the initially allocated extents fill up, a Parallax instance has to move on to newer extents. However, in due time if VDIs or their snapshots are deleted and reclaimed, that could create empty blocks in the old extents. When these extents are again allocated, it could be to any Parallax instance, resulting in the pages of the VDIs of the two Parallax instances being interspersed.

3) A user could shutdown a VM on one Parallax instance A and then use the same VDI to boot another VM on another Parallax instance B. Or a user could simply migrate a VM from one Parallax instance to another Parallax instance in the Parallax cluster. Booting on another Parallax instance will cause all future block allocations to happen from the extents owned by the new Parallax instance, which will again result in VDI fragmentation, both at the data as well as metadata levels.

4) In our usage scenarios (backups for disaster recovery, system replay etc.), we imagine our users taking a large number of snapshots of the VDIs at a very frequent rate. Every time a VDI is snapshotted, it renders the entire state of the VDI read-only (data as well as metadata). Any writes to any of these pages will trigger a CoW operation to preserve the snapshot. In such cases, if the VDI is mutating even at a reasonable rate, then coupled with the snapshot operations, the VDI could end up highly fragmented. A similar situation occurs when a VDI is created from a snapshot of another VDI. If both VDIs are being managed by different Parallax instances, then again all future allocations to the new VDI will be in a different extent.

Thus, Parallax has a tendency to fragment VDIs. Since OS agnosticity is one of the chief design principles of Parallax, the guest OSs running in the VMs have little idea about the kind of fragmentation below the logical address space. For example, consider Figure 3.1. Any defragmentation attempts by these OSs are

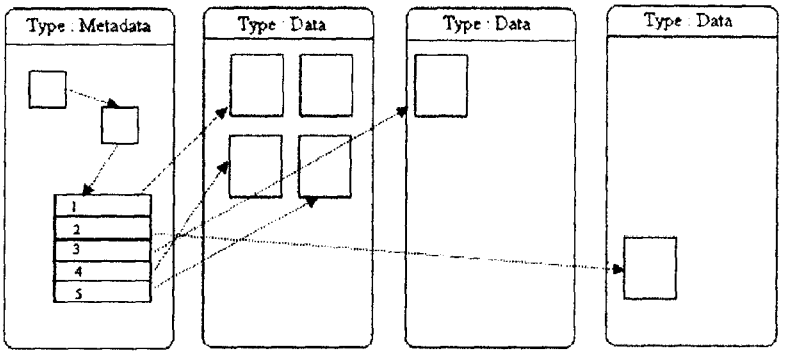

Figure 3.1: Example of a fragmented VDI

more likely to backfire and worsen the I/O performance. As we can see, as far as the guest file system is concerned, the pages allocated to the VDI will all seem contiguous to it. It is only when some files are deleted and the guest file-system starts reusing the disk blocks, that it will notice any fragmentation. And even then, it is most likely that any remapping schemes that the file system will come up with will be faulty and might only worsen the fragmentation problem. If the guest file-system had some hints about the physical layout of the pages of the disk, then it could take smarter decisions about remapping its pages. But that would violate the OS agnosticity design principle. Added to that, even if we compromised on this principle, a guest file-system can only know about the layout of the pages of its VDI. Hence, any remapping decisions that it will take can only involve juggling the pages available to it. On the other hand, if a Parallax cluster-wide service can take on the task of defragmenting VDIs (or at least part of the VDIs), then it can look into entire blockstore and look for better remapping schemes (e.g., look for larger chunks of contiguous disk space).

3.3 Superpages

As discussed in Section 3.2, Parallax has a tendency to fragment VDIs across the blockstore which can cause significant performance overhead as the blockstore ages. The solution is that the VDIs be defragmented from time to time. In order to defragment a VDI, we have to remap its pages to a contiguous piece of available disk space in the blockstore. Sometimes, it may not be feasible to find a contiguous chunk of diskspace large enough to be able to remap a whole VDI. Hence, we should be able to remap at least a part of the VDI. As explained earlier, Parallax

allows the sharing of VDI pages. These pages can be data pages or radix-tree pages (metadata pages). If there is more than one pointer to a radix-tree page, then that indicates that the entire subtree below it is also shared. Hence, remapping at the granularity of VDI subtrees will extend its advantages to more than one VDI. We call these defragmented VDI subtrees *Superpages*.

Once the superpages are created, they should remain so. They should not get remapped as a side-effect of defragmenting another VDI subtree. Hence, the remapping mechanism should be able to identify superpages. For this, we modify the address representation in the radix tree nodes. Earlier, only the 64th bit in the global block-address was a reserved bit. This bit is used to indicate whether the disk-page pointed to is writable or is read-only. Now we also reserve the 63rd bit and use it to indicate if the disk block pointed by the address is the base address of a superpage. The format of a global block-address is as shown in Figure 5.1. The size of the superpage depends on which level of the radix tree the address occurs in. As explained earlier, VDI block addresses are stored in a 3-level radix tree, each node of which is of size 4K. Each of the radix metadata pages stores 512 64-bit global block addresses, the two highest bits of which are reserved. If the root node address in the vdi-registry is a superpage address, then it indicates that the whole VDI is a superpage and 512GB of contiguous space has been set aside for it (a very unlikely case). Similarly, if the superpage address occurs in level 1 of the radix tree, then it points to a contiguous space of size 1GB and if it is at level 2, it points to a contiguous space of size 2MB. Level 3 nodes point to 4K blocks, hence superpage addresses won't make any difference there yet unless the number of levels in the radix tree is increased to accomodate VDIs larger than 512GB. Thus, superpage addresses can be at any level of the radix tree, as shown in Figure 5.2. This addressing scheme also eliminates some levels of address translation. During the address translation process, if a superpage address is encountered, then the logical block number is simply added to the superpage address to get the physical block address. Hence, superpages not only help with presenting defragmented VDI subtrees, but also help with reducing the address translation overheads.

The pages have to be remapped irrespective of whether the VDI is in use or not. This is take care of by the Remapper process as explained in Section 5.5. In order to create a superpage, a contiguous chunk of disk space is required. As the blockstore gets increasingly fragmented, these chunks will have to be made available. An *Extent Defragmenter* process uses the Remapper to remap disk blocks of an extent so that all free diskspace gets merged into one large free pool. This is explained in Section 5.4. At the outset it might seem that creating superpages is just a one time overhead. One can create VDIs from any Parallax host, access them and migrate them and remapping techniques can take care to see that the VDIs are eventually defragmented. However, that is not always the case. Firstly, because Parallax

is designed to support millions of VDIs and blockstores of size in the range of Terabytes. Creating superpages does incur CPU overhead. Also, because there could be significant sharing among VDIs, it may not be possible to convert all VDIs into superpages. To convert a VDI subtree into a superpage, it is important that the subtree should be largely filled first. Because, creating the superpage will involve setting aside a significant chunk of disk space. For example, if a VDI subtree is only half filled and it is converted to a superpage, then when eventually more allocations are done, if there is no pre-allocated disk space, the efforts gone into creating the initial superpage will be wasted. Pre-allocating disk space in anticipation could also prove wasteful. Hence, we give preference to creating superpages starting from the lower levels of the radix tree. Also, having superpages at lower levels of the tree is likely to benefit many more sharing VDIs.

3.4 Summary

Parallax has a tendency to fragment data and metadata blocks over time across the blockstore. This fragmentation can lead to longer seek times. Snapshot and cloning operations lead to sharing relationships amond VDIs that need to be taken into account before any blocks can be remapped or reclaimed. Fragmentation can be reduced by remapping frequently accessed parts of VDIs so that they are physically sequential on disk and thus creating superpages. This can also reduce some of the address translation overhead. In the next few chapters we will see the design and implementation of techniques to achieve this.

Chapter 4

Design

4.1 Introduction

In this chapter we will look at the design of block remapping schemes that can help solve the problems of garbage collection and fragmentation in Parallax. We will first look at a naive approach to transparently remap blocks and discuss its limitations. We will then derive a better approach and discuss the design changes to Parallax's metadata that are required to implement it.

4.2 Naive approach

Conceptually, the problem of remapping and reclaiming blocks in the blockstore is not a hard one. Given the metadata of the entire blockstore and enough time and memory resources, one can find all block locations that have alteast one reference to them. All remaining blocks are either unallocated blocks or need garbage collection. Using the block allocation maps of each extent, one can separate the two and thus do garbage collection. Similarly, given memory resources large enough to store an arbitrarily large address translation table, one can find all references to an old block address and modify them so that they point to the new block address. Implementing these approaches is also relatively trivial and require no changes to the current metadata layout of Parallax.

There are several practical problems with the approaches mentioned above and they all spring from the fact that they work on a blockstore-wide scale. As the size and usage of the blockstore increases, the size of the metadata also increases. Accordingly, the time required to scan all this metadataa and trace all block references to find unreferenced or remapped blocks increases. But since these operations are not really high priority, time is one of the smaller concerns. As far as garbage collection is concerned, the main resource required is memory to hold a copy of the blockstore's block allocation maps. For a blockstore of size 1 TB, the total size of block allocation maps is 256MB (1 bit for every 4K size block), which needs to be stored in memory. This seems manageable. However, for a blockstore of size 1 Petabyte, the size of all block allocation maps is 32GB. One cannot dedicate 32GB

memory for garbage collection.

An address translation table for remapping consists of a number of two-column tuples. Column 1 of the table is the old block address (that is currently recorded in the radix nodes) and is of size 64 bits. Column 2 of the table is the address to which the block content has to be remapped to and is also of size 64 bits. Hence every row is of size 128 bits. For the process of remapping blocks, one can simply keep the entire address translation table in memory and then compare the entries of each radix node to each entry in the table to check if it needs to be changed. In the worst case, the address translation table could be remapping half the blockstore. For a blockstore of size 1 TB, the size of the address translation table could then be as large as 32GB ($\frac{Number\ of\ 4K blocks\ in\ extent}{2} * 128bits$). Hence, the global approaches are not scalable in memory as the size of the blockstore increases.

4.3 Per-extent approach

To solve the above explained scalability problem, an obvious solution is to break up the problem into smaller sub-problems that can be dealt with individually. For example, instead of garbage collecting the entire blockstore, one can instead try to garbage collect/remap only a part (of fixed size) of the blockstore. This would put a cap on the maximum resource required to run the utility. An obvious choice is to do this on an extent basis, since the blockstore(and its metadata) is already divided into extents. The main advantage of this approach is that the utility should not have to scan all the metadata of the blockstore, but only that part of the metadata that concerns the blocks in the extent. To facilitate this, we introduce some extent specific metadata that needs to be maintained, which is described next.

4.3.1 Per extent metadata

VDI journal

The main problem with writing block reclamation and remapping tools for only a given extent, is finding all references in the blockstore to any physical block address in the extent. In Parallax, CoW snapshots introduce sharing semantics between virtual disks and snapshots. The ability to create CoW clones of virtual disks from snapshots of other virtual disks leads to more sharing relationships. An extent could have been allocated to several Parallax instances since it was first initialized. Hence, the blocks in an extent could be referred by any VDI in the registry, by its snaphots, its clones and so on.

For the utilities to work on one extent at a time, we need to know which VDIs in the VDI registry could have references to blocks in the extent. For this, a VDI

journal is maintained for every extent. The ids of all VDIs that have pages allocated from the extent are recorded in the VDI journal. When an extent is allocated to a parallax instance, the VDI journal is read into memory by the Parallax instance and maintained as a sorted list of VDI ids. When the parallax instance recieves a request to open a VDI, it first checks if the VDI id is already recorded in the VDI journal. If it is not recorded in the journal, then the VDI id is appended to it. A parallax instance could have opened several VDIs before it is allotted this extent. These VDIs could subsequently allocate pages from this extent and hence their VDI ids also have to be recorded in to the VDI journal. To facilitate this, the parallax instance maintains a list of VDI ids of all VDIs that it has currently open. Every time a VDI is mounted, its VDI id is appended to this list. When the VDI is closed, its id is removed from the list. When an extent is newly allocated to an extent, the parallax instance adds these VDI ids to the VDI journal of the extent.

The VDI journal will only provide a list of all VDIs that have allocated pages in the extent and written to them. But as the VDI ages, it could be snapshotted and cloned. Even if the VDI is deleted later, its snapshots and clones still have references to these blocks. This has to be taken into account when reclaiming/remapping blocks. We assume that on an average each VDI recorded in the journal would have allocated at least 2 pages from the extent and reserve disk space accordingly for writing the journal in the extent metadata.

Remap space

Processes that want to remap blocks for creating a superpage or to defragment free space in an extent have to record the remappings of old locations to the new locations. These remappings are recorded in the Remap Space(RSpace). For every block in an extent that has to be remapped, its new location is recorded in its corresponding 64 bit space in the RSpace. The remapper process looks at the RSpace and does the required remappings. Since more than one process can modify the RSpace, access to it is regulated using by the lockmaster.

ROMap

The deduper needs to know which blocks in an extent are read-only so that it can calculate the fingerprint of these blocks to look for potential matches. It looks for these read-only blocks by reading the Read-only Map (ROMap). The ROMap is a bitmap in which every bit represents whether the corresponding block in the extent is read-only or not. The deduper has only read access to an ROMap. The garbage collector is the only process that can modify the contents of the ROMap. While the garbage collector is executing, it has access to system wide metadata and in-

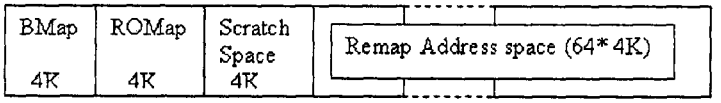

Figure 4.1: Extent metadata layout for every 4096 blocks.

formation regarding which blocks are read-only and which are writable. Hence the garbage collector sets the respective bits in the ROMaps. Since only one process can modify an ROMap, it does not need lock access.

Scratch space

Some scratch space is also reserved. The garbage collector uses this scratch space to make an on-disk copy of the BMaps before it starts executing. The size of the scratch space is equal to the size of the BMap.

All this metadata is laid out in the extent in such a way so that it can be accessed mostly linearly. For every 4K contiguous blocks of the extent, the related metadata i.e., its BMap, ROMap, scratch space and remap space are laid out together as shown in Figure [4.1]. This helps the metadata services linearly access all metadata related to the blocks in an extent.

Serialized block pool

Parallax instances allocate blocks from a pool of unallocated block addresses as explained in Section 5.2. Although these blocks are not yet allocated, every block-address recorded in the blockpool is recorded as allocated in the BMap. To avoid inconsistencies, the garbage collector needs to know the contents of the block pool from an extent so that it can take them into account when reclaiming blocks of the extent. Hence, the block allocator writes the contents of the block pool in the extent metadata, which is then used by the garbage collector. This is explained in more detail in Section 5.2 and Section 5.6. We assume that the size of the block pool will never be more than half the extent size and hence reserve disk space accordingly for writing the block pool in the extent metadata.

The metadata layout of an extent is as shown in Figure (4.2).

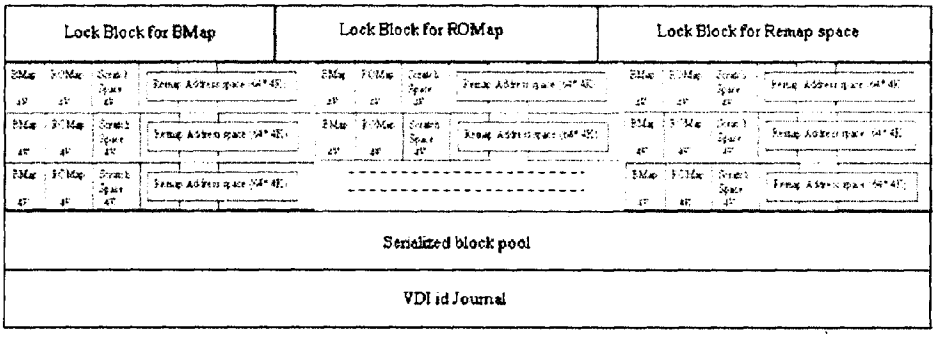

Figure 4.2: Extent metadata layout.

4.4 Summary

In principle, block reclamation and remapping are very trivial operations. But if they are implemented the trivial way (blockstore-wide), they can be unreasonably memory intensive. Hence they have to be implemented such that they can run on one extent at a time. To facilitate this, some more metadata needs to be stored in the extent headers. We have seen why each of this is required and how it is laid out in the extent for efficient access. In the next chapter we will see the implementation details.

Chapter 5

Implementation and evaluation

5.1 Introduction

In this chapter we are going to look at the implementation details of the block allocator, garbage collector, block remapper, extent defragmenter and superpage remapper. We will also take a brief look at the performance of each of these utilities. The memory and disk resources used by all these tools is directly related to the size of the extent. Hence we shall see how they behave on the time scale as the size of the extent varies.

5.2 Block allocator

When a Parallax instance boots, it grabs two extents for its use: a data extent and a metadata extent. These extents are allocated by the *Block Allocator* using a simple algorithm. To allocate a metadata extent, it goes through the extent catalog in the Superblock extent and looks for the first unlocked extent that is either unused or is a metadata extent. Each block in the extent catalog gives the following information about an extent:

1) Extent id

2) Type of extent: Data / Metadata / Unused

3) Lock status: Locked / Unlocked

4) Number of free blocks in the extent. If this value is zero, it indicates that the extent does not have any more free blocks.

5) Block Number at which the Block Allocator had last stopped looking for free blocks. When the extent is next allocated, the Block allocator will start looking for new blocks from this point onwards.

The *Block Allocator* locks the extent with the Parallax Id of the Parallax instance that requested the metadata extent. If it is an unused extent, the type of the extent is set to type metadata and the id of the extent is passed on to the Parallax instance. A data extent is allocated similarly.

When the Parallax instance needs to allocate a block from either the data or the metadata extent, it first needs to know which blocks in the extent are free and hence

can be allocated. Each extent (except the superblock extent) has a Block Allocation Map (BMap) and its associated lock. They are stored in the beginning of the extent. The BMap is a simple bitmap, in which each bit indicates whether the corresponding block in the extent is allocated or not. When a Parallax instance needs new blocks, the block allocator locks the BMap of the extent. This is necessary because the BMap of an extent can also be modified by the garbage collector. Hence, both processes, the block allocator and the GC have to first get a lock on the BMap before they can read/write to it. If the extent had been previously allocated, then its header information in the extent catalog will indicate its last allocated block. If this value is non-zero, the block allocator looks for unallocated blocks starting from this block. A fixed-size list of the addresses of the unallocated blocks is compiled by the block allocator. In the current implementation, this size is set to 10,000 blocks, but it is a tunable value. For every block address that is included in the list, its corresponding bit in the BMap is set indicating that it is no longer free. If a garbage collector is executed at this time, then it would detect that these blocks are not really in use and mark them as free in the BMap. Since the Parallax instance using the extent is oblivious of the garbage collector, it could end up finally allocating the block from the pool although it is still recorded as available in the BMap. To avoid such inconsistencies, the garbage collector needs to know the contents of the block pool from an extent so that it can take them into account when reclaiming blocks of the extent. Hence, every time a new pool of blocks is created, the block allocator writes the contents of the block pool in the extent metadata, which is then used by the garbage collector.

The Parallax instance stores the free-list in memory and uses it to allocate disk blocks until the list is exhausted. When the free-list is close to getting exhausted, the block allocator repeats the above process starting from locking the BMap, and looks for free blocks from the point where it had stopped the last time. If there are no more free blocks in the extent, then the extent is marked as full, unlocked and another extent is allocated to the Parallax instance.

When the Parallax instance is shutting down, it first unlocks its allocated data and metadata extents. This involves the following:

1) Lock the BMap of the extent and unmark all blocks that were not used from the free list.

3) Write the address of the last allocated block in the extent catalog.

4) Calculate the number of free blocks in the extent by scanning the BMap and note it in the extent catalog.

5) Unlock the BMap.

4) Reset the lock status of the extent in the extent catalog.

If a Parallax instance crashes, it will not be able to unlock the extents and because the free-list is only maintained in memory, any unused blocks will remain

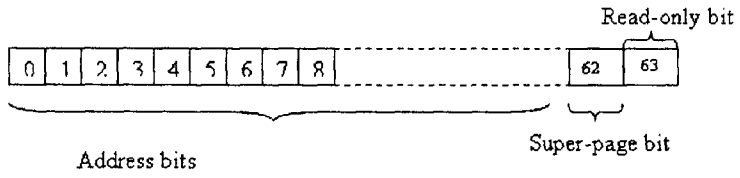

Figure 5.1: Address bits in a radix tree node

marked on the BMap of the extent. The extents can be unlocked manually using a simple script. The unallocated blocks will be reclaimed eventually by the garbage collector as explained in Section 5.6.

5.3 Superpages

As explained in Section 3.3, *superpages* are specially laid out virtual disks. It is easier to defragment a part of VDI rather than degragmenting the whole VDI. Hence, we start by defragmenting just the lower levels of the VDI radix tree, also called as VDI subtrees. Thus, a superpage can also be just a part of the virtual disk, instead of the whole virtual disk. We use two heuristics to choose candidate VDI subtrees. The first heuristic is to choose VDI subtrees with a large number of references to them. Another heuristic is to choose those VDI subtrees that are accessed relatively more often and hence will benefit with reduced overhead in address translation and disk seek.

Given the root of a VDI subtree, there are two steps in converting it to a superpage. In the first step, as described in Algorithm 1, we find a contiguous piece of the blockstore into which the pages of the VDI subtree can be remapped. The address to which a block is to be remapped is stored in the metadata of the extent in which the block occurs. After the pages are copied to their new locations, their remap addresses in the extent metadata are set to -1. This is to help the free space defragmenter recognize superpage blocks in an extent. The third step is to change all references to the old locations so that they point to the new locations. This is accomplished by the Remapper as described in Section 5.5.

5.4 Free space defragmentation

This module defragments the free space in an extent. Ordinarily, one can try to come up with a scheme to remap pages in an extent, such that one has to do mini-

Algorithm 1 Parallax's Superpage Remapper

1. MaxSize = Calculate maximum size of VDI subtree
2. Superpage_Extent = Scan Extent_Catalog for an unused extent
3. If (Superpage_Extent == NULL)
   Superpage_Extent = Scan extent catalog for an unlocked extent with MaxSize empty blocks
Else (Report Failure)
4. if (Superpage_Extent)
   Lock Superpage_Extent
   if (MaxSize blocks are not contiguous)
         Response = Call FreeSpace_Defragmenter to Defragment Superpage_Extent
         if (Response == Failure)
           Go to Step 3.
5. List_Extents = Create a list of extents spanned by the pages of the VDI subtree.
6. Write remap addresses in the remap space of extents in List_Extents.
7. Copy pages from VDI Subtree to Superpage_Extent.
8. Mark remap addresses = -1 for each superpage block.
9. Execute Remapper for every extent in List_Extents.

mum number of remap operations to get the largest contiguous chunk of the extent. However, if any of the pages in the extent belong to a superpage, then reampping them would mean that we are destroying the original superpage to make another new one. The benefits from the new superpage could be more than the old one. However, as of now, we do not have any means to compare such benefits. Hence, by default, superpages are not remapped to make a new superpage. If a block is a superpage block, then its remap address is set to -1. The free space defragmenter will not remap these blocks. As of now, we follow a naive approach to come up with a remapping scheme. The defragmenter looks at the block allocation map and remaps allocated blocks (except superpage blocks) starting from the end of the extent to any available blocks from the beginning of the extent. Thus, it tries to create a contiguous piece at the end of the extent. The address to which a block is remapped is stored in the remap space of the extent metadata and the block contents are copied to the new location as well. If the block has a writable pointer to it, then it will be recopied during the remapping process. The algorithm for free space defragmentation is described in Algorithm 2.

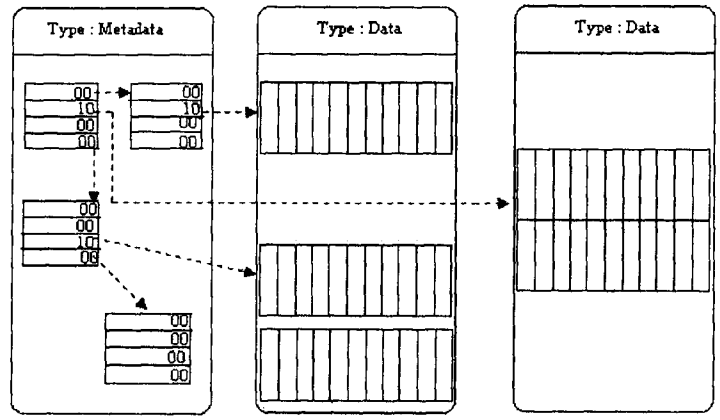

Figure 5.2: Example of superpages

5.5 Remapper

As discussed in Section 5.4 and Section 3.3, once the block remappings have been recorded in the extent headers, the next step is to change all references to the old locations so that they point to the new locations. This is done by the Remapper. It works on a per extent basis. If a block is writable, then it will have only one radix page pointing to it, whose entry will have to be modified for correct future accesses. But, if a block has at least one read-only pointer to it, then there is a distinct possibility that there may be many other such pointers. Once all these pointers have changed, the remapped page will be reclaimed by the garbage collector (even-

Algorithm 2 Parallax's Free Space Defragmenter

1. Lock Defrag_Extent
2. Lock the BMap of Defrag_Extent.
3. Scan the BMap from the end for allocated blocks.
4. For every allocated block
   If remap address != -1
   - 4.1 Write remapping to remap address.
   - 4.2 Copy block contents to new location.
   - 4.3 Set the corresponding bit in the BMap.
5. Execute Remapper for Defrag_Extent.

tually).

Finding writable pointers to blocks is relatively trivial. Given the VDI journal in the extent, the Remapper only has to search for references to blocks in the radix pages of the VDIs recorded in the journal. Since these are writable blocks, they cannot be modified trivially by the Remapper. If the VDI is already owned by a Parallax instance, then the remapper delegates the actual modification of the blocks to the Parallax instance. It provides the Parallax instance with the following information: address of the radix node, value of the entry to be changed, new value of the entry, indication if the disk block needs to be copied to the new address. If the Parallax instance loses ownership of the VDI, it returns an appropriate error code. The remapper then rechecks the ownership of the VDI. If the VDI is unowned, it takes ownership itself and modifies the radix blocks. Else, it contacts the respective Parallax instance.

Finding all read-only pointers involves a little more work. A block that has a read-only pointer can only occur in a snapshot. A block in a snapshot will have pointers in radix blocks of the snapshot, the VDI of the snapshot and any VDIs that were created by cloning any snapshots of the VDI (and their snapshots).

Hence we have to scan the following for pointers:

1) VDIs that had allocated pages from the extent (VDI id journal). 2) Snapshots of all VDIs included in 1. 3) All VDIs that were cloned from snapshots included in 2. 4) Snapshots of VDIs included in 3. 5) Any VDIs that were cloned from the snapshots included in 4 and so on.

The algorithm for remapping read-only pages in an extent is in Algorithm 3. In Step 1, the remapper compiles a list of all VDIs that might have pointers to pages in the extent being remapped. In Step 2, using a single sequential pass of the VDI id journals of all metadata extents, the remapper compiles a list of extents spanned by these VDIs. A Reachability Map (RMap) records which pages belong to the VDIs (or their snapshots) that were recorded in Step 1. The remapper can only modify entries of radix pages that are read-only themselves. Hence to keep track of which radix pages are read-only, it uses another bitmap called the Read-Only bitmap (ROMap). This is an in-memory bitmap that is maintained for every extent included in the list compiled in Step 2. If a bit in the ROMap is set, it implies that the corresponding block in the extent is read-only. In Step 3 the RMaps and ROMaps are initialized to zeros. In Step 4, the Remap-space in the extent header is read into memory. Using the RMaps and the ROMaps, the remapper changes all old locations to point to the new locations. Initially, only the radix roots of VDIs and their snapshots are marked as in the RMaps and only the snapshots are maked in the ROMaps. Subsequent passes mark blocks that are reachable from these radix roots and so on. For every page that is marked in the RMap, we check if any of its entries are within the address range covered by the extent being remapped. If

an entry does belong to the extent, then we can index into its corresponding entry in the Remap space to check if it needs to be remapped. If any matches are found, they are suitably modified and the modified page is written back to disk. All final entries in the radix page are marked in the RMaps and ROMaps accordingly. This process is repeated for every level of the radix tree and each time the entire RMap and ROMap is scanned. At the end of the process, all read-only radix pages point to the new locations.

Algorithm 3 Parallax's Read-Only Remapper

Remap_Extent = Id of extent to be remapped.
1. VDI_List = List of all VDIs that could have pointers to blocks in Remap_Extent.
    1.1 VDI_List = All VDI ids in VDI id journal of Remap_Extent.
    1.2 Repeat : VDI_List_Length1 = length of VDI_List.
        VDI_List += All VDIs cloned from VDIs in VDI_List.
        VDI_List_Length2 = length of VDI_List.
    Until($VDI\_List\_Length2 > VDI\_List\_Length1$)
4. For each VDI in the VDI_List:
  If the VDI is not marked as deleted :
    Mark the position of radix root in the RMap.
    For each snapshot that is not marked as deleted:
      Mark its radix root in the RMap.
      Mark its radix root in the ROMap.
5. Remap_Table = Remap space of Remap_Extent
6. Mark all New_Locations in Remap_Table in respective RMaps.
7. For each marked entry in an RMap
  If the corresponding entry in ROMap is marked :
    Check if any entries in the page need to be remapped
      If matches are found, the entries are modified
        and the page is written back to disk.
    Mark all pages (on RMap) that it points to.
    Mark all pages (on ROMap) to which it has read-only pointers.
8. Repeat step 7 for each level in the radix tree.

5.6 Garbage collection

Parallax nodes are free to allocate new data to any free blocks within their locked extents. Combined with the copy-on-write nature of Parallax, this makes deletion

Algorithm 4 Parallax's Writable Remapper

```
Remap_Extent = Id of extent to be remapped.
1. VDI_List= List of allVDIs in VDI Journal of Remap_Extent.
2. Remap_Table = Remap space of Remap_Extent
3. For each VDI in VDI_List:
        If VDI is not owned by any Parallax instance, lock it.
        Traverse only writable links in the radix tree depth-first order
        Read the radix page into memory.
            Compare all entries in the page with the entries in Remap_Table.
            If matches are found,VDI owner executes:
                If pointer is read only :
                    UpdateParent(ParentNode, FromAddress, ToAddress)
                If pointer is writable :
                    UpdateParentAndRelocate(ParentNode, FromAddress, ToAddress)
4. Repeat step 3 for each level in the radix tree.
```

a challenge. Our approach to reclaiming deleted data is to have users simply mark radix root nodes as deleted, and to then run a garbage collector that tracks metadata references across the entire shared blockstore and frees any unallocated blocks.

Parallaxs garbage collector is described as Algorithm 5. It is similar to a mark-and-sweep collector, except that it has a fixed, static set of passes. This is possible because the maximum length of any chain of references in the VDI is equal to the height of the radix trees (which is currently 3). As a result we are able to scan the metadata blocks in disk order rather than follow them in the arbitrary order that they appear in the radix trees. The key data structure managed by the garbage collector is the Reachability Map (RMap), an in-memory bitmap with one bit per block in the blockstore; each bit indicates whether the corresponding block is reachable. A significant goal in the design of the garbage collector is that it interfere as little as possible with the ongoing work of Parallax. While the garbage collector is running, Parallax instances are free to allocate blocks, create snapshots and VDIs, and delete snapshots and VDIs. Therefore the garbage collector works on a checkpoint of the state of the system at the point in time that it starts. Step 1 takes an on-disk read-only copy of all block allocation maps (BMaps) in the system. Initially, only the radix roots of VDIs and their snapshots are marked as reachable. Subsequent passes mark blocks that are reachable from these radix roots and so on. In Step 5, the entire RMap is scanned every time. This results in re-reading nodes that are high in the tree, a process that could be made more efficient at the cost of additional memory. Every Parallax instance grabs a pool of

Algorithm 5 Parallax's Garbage Collector

1. Checkpoint Block Allocation Maps (BMaps) of extents.
2. Initialize all Reachability Maps (RMaps) to zeros.
3. For each VDI in the VDI registry :
   If VDI is not marked as deleted :
      Mark the position of radix root in the RMap.
      For each snapshot in its snaplog
         If snapshot is not marked as deleted:
            Mark its radix root in the RMap.
4. For each metadata extent :
   Scan its RMap, if a page is marked:
      Mark all pages (on RMap) that it points to.
5. Repeat step 4 for each level in the radix tree.
6. For each VDI in the VDI registry:
   If VDI is marked as not deleted :
      Mark each page of its snaplog in its RMap.
7. For each extent:
   Lock the BMap.
   If the extent is locked by a Parallax instance
      Read the block pool from the extent metadata.
      Mark the block addresses in the pool as reachable.
   For each unmarked bit in the RMap:
      If it is marked in the BMap as well as in the
      checkpointed copy of the BMap :
         Unmark the BMap entry and reclaim the block.

blocks (currently 10,000) for future allocations. Although these blocks are marked as allocated in the BMap, at any given time, several of them would be in fact waiting allocation and hence not reachable. The block addresses in the free pool are written to extent metadata after every run of the block allocator as explained in Section (5.2). To avoid reclaiming these blocks, the garbage collector reads their addresses from the extent metadata and marks them as reachable. Hence, even though these blocks are marked as allocated in the BMap and its checkpoint and are also not reachable, they are still not reclaimed. All other blocks that were marked as allocated in the checkpoint taken in Step 1 are considered as candidates for deallocation by the collector (see Step 7). The only time that the collector interferes with ongoing Parallax operations is when it updates the (live) allocation bitmap for an extent to indicate newly deallocated blocks. For this operation it must coordinate with the Parallax instance that owns the extent to avoid simultaneous updates, thus the BMap must be locked in Step 7. Parallax instances claim many free blocks at once when looking at the allocation bitmap (currently 10,000), so this lock suffers little contention.

5.7 Evaluation

In this section, we will look at the performance of garbage collection, extent defragmentation, read remapping and write remapping. In all tests, I used IBM eServer x306 machines, each node including a 3.2 GHz Pentium-4 processor, 1 GByte of RAM, and 3 Intel e1000 GbE network interfaces (only one interface is active during the tests). Storage is provided from a NetApp FAS3070 4 exporting an iSCSI LUN over gigabit links. The filer is accessed in all cases using the Linux openiSCSI software initiator (v2.0.730, and kernel module v1.1- 646) running in domain 0. All development was done against Xen 3.1.0 as a base.

5.7.1 Garbage collector

Since the Parallax garbage collector works via sequential scans of all metadata extents, the performance of the garbage collector is determined by the speed of reading metadata and the amount of metadata, and is independent of both the complexity of the forest of VDIs and their snapshots and the number of deleted VDIs. Weve run the garbage collector on full blockstores ranging in size from 10GB to 50GB, and its performance is perfectly linear at a rate of 1.03GB/sec as show in Figure 5.3. The performance of the per-extent garbage collector is also mostly linear as shown in Figure 5.4.

Given a blockstore, the relative cost (time per GB) is expected to be higher for

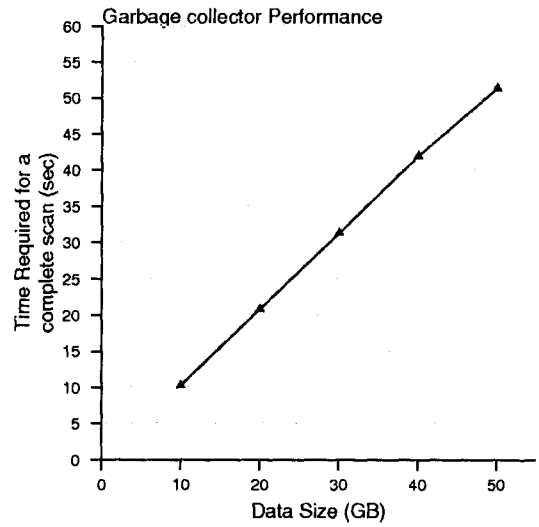

Figure 5.3: Performance of global garbage collector

the per-Extent GC. This is because the per-Extent GC has to do the extra work of calculating the subset of VDIs that need to be scanned. Also, it is possible that several VDIs that are scanned have very few pages in the extent that is being garbage collected. In spite of that, all blocks of the VDI will be checked for reachability. Whereas, In the case of the global garbage collector, every block that is marked contributes to the progress of the process.

5.7.2 Extent defragmentation

The defragmentation process involves copying disk pages from one end of the extent to a free slot on the other end of the extent. In the worst case, the extent could be fragmented such that the first half is unused and the other half is allocated. This would require copying the allocated half of the extent to the unallocated half, one disk block at a time. This will incur a number of disk seeks back and forth across the extent. This is the test case for evaluating the defragmentation process and the results are as shown in Figure 5.5. It has been evaluated for different extent sizes (2GB - 10GB). It only shows the worst case performance. Defragmentation can be optimised in a number of ways like implementing smarter schemes for rearranging disk blocks such that it requires minimum copying and batching reads and writes

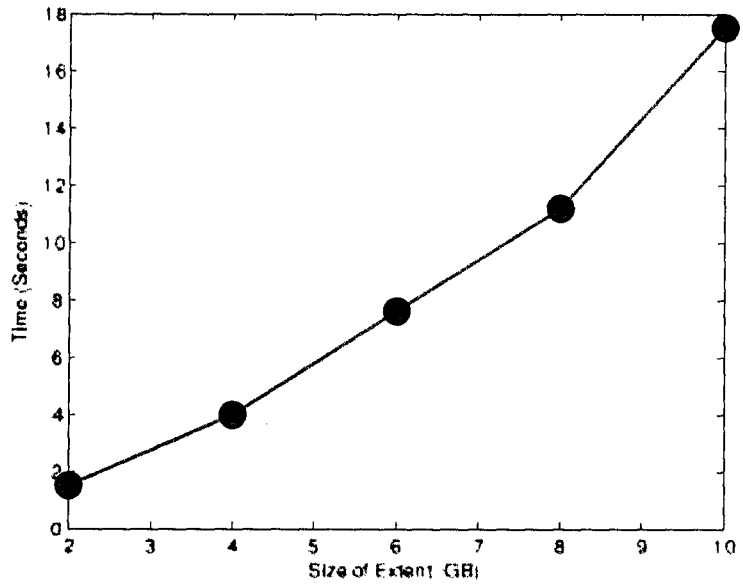

Figure 5.4: Performance of per extent garbage collector

of disk blocks.

5.7.3 Read remapping

The workings of Read remapping are very similar to that of garbage collection. Read remapping gathers a list of all VDIs that could have pointers to blocks in an extent. However, unlike the garbage collector, it only traverses read-only links and checks the third level of the radix tree to see if it points to a block in the extent being remapped and if it needs remapping, then it changes the entry in the radix page to the new entry. If any changes are made to a radix block, it is written back to disk after all its entries have been checked. Hence, read remapping involves linear scans of a few metadata extents. It was evaluated by remapping extents whose remap tables were generated by defragmenting the extents as mentioned in Section 5.7.2. After defragmenting the extent, all VDIs with pointers in the extent were snapshotted to render all their pages read-only. Read remapping is then executed to change the pointers in the radix block of these VDIs to the new addresses recorded in the remap table. The performance is again mostly linear as shown in Figure 5.6.

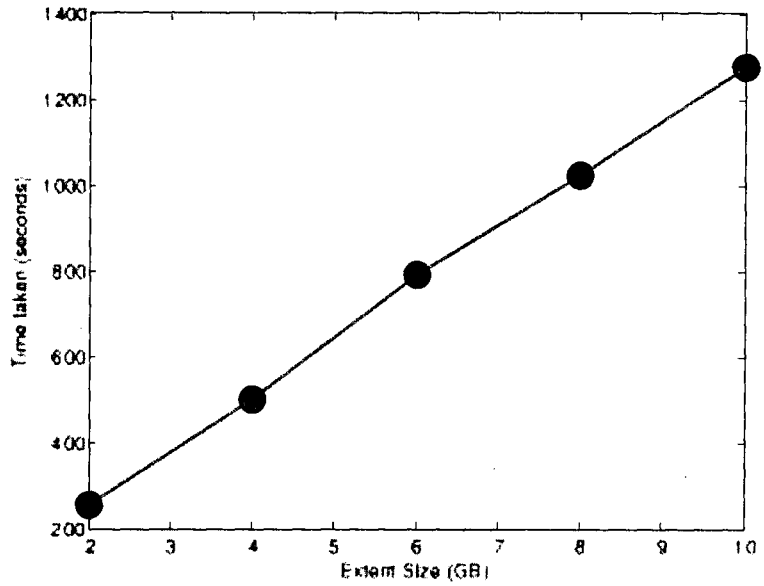

Figure 5.5: Performance of extent defragmentation

5.7.4 Write remapping

The remap tables generated from defragmenting extents in Section 5.7.2 were used to test write remapping. The write remapper gathers a list of all VDIs that have pointers to blocks in the extent and then traverses only the writable links and checks if they need remapping. If an entry needs to be changed, it is changed in the radix block and the corresponding disk block is also copied to the new location. In case the VDI is locked by another parallax instance, these changes have to be delegated to the parallax instance. However, the current evaluation assumes that the VDI is not locked. The performance of the write remapper is as shown in Figure 5.7.

5.8 Summary

The metadata services implemented thus far can successfully defragment and remap extents in the blockstore. Better defragmentation schemes should definitely help in reducing the rearrangement of disk blocks and it would help if the extents are defragmented regularly. Schemes for defragmenting storage have been in the works since a long time. In the next section we will survey some existing storage systems and their mechanisms for defragmentation and remapping.

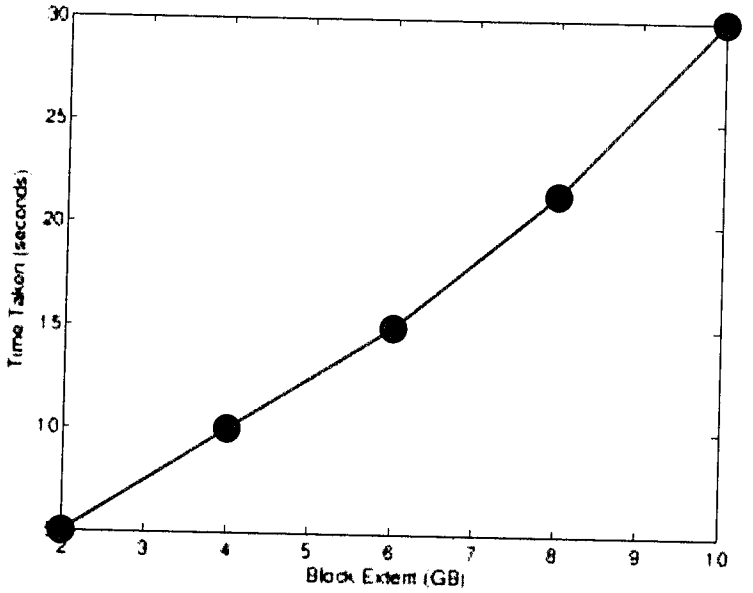

Figure 5.6: Performance of read remapper

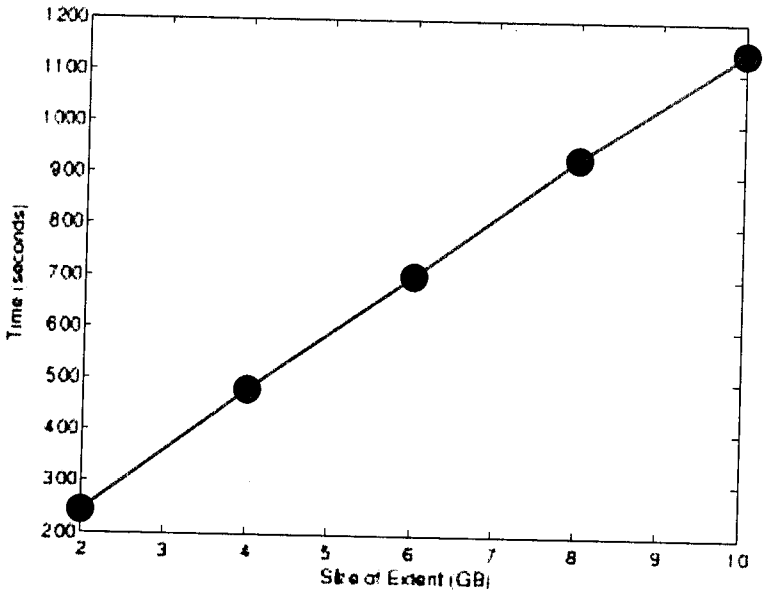

Figure 5.7: Performance of write remapper

Chapter 6

Related work

This chapter is a brief survey of existing storage virtualization systems and their methods to garbage collect, defragment and dedup the storage pool. Broadly, they can be categorised into volume managers and file systems. For each system, we will look at how they store and maintain their metadata. In particular we will look at how they implement delete (deleting a file/virtual disk or garbage collection) and defragmentation, since both are crucial operations in any storage system. While deduping is not exactly a must-have feature, it is certainly a huge cost-cutter and any competing backup system today is trying to implement it [11]. Hence, we will also look at some popular storage solutions that offer deduping.

6.1 Volume managers

The Logical Disk (LD) Interface [21] uses the notion of separating file management from disk management. The file system would manage files and interact with LD via logical block numbers and *Block lists*. LD translates the logical block numbers to physical block numbers using a *Block-number Map* which is kept in memory. If a file system puts the blocks of a file (including file metadata) on a block list, then LD can do file defragmentation by placing these blocks physically together on disk. The process would be totally file system agnostic, because the logical block numbers would remain unchanged. Garbage collection, on the other hand, would be dictated by the file system. For example, in a *SpriteLFS* [18] implementation of LD, the disk is divided into fixed-size logical segments. *Segment Cleaning* is SpriteLFS's version of garbage collection and is carried out by LD.

The Petal [12] system aims at separating the view of a distributed storage system from the management of physical resources that implement it. Petal virtualizes a given pool of commodity disks and servers, and presents virtual disks to the distributed file system. One can add or remove physical disks/servers and take snapshots of these virtual disks. These operations are all file system agnostic. Petal clients work with virtual disk addresses of the form < *vitual-disk identifier, offset*> which are translated to <*serveridentifier,diskidentifier,diskoffset*>. The translation is carried out with the help of three important data structures: a virtual disk direc-

tory (VDir), a global map (GMap), and a physical map (PMap). The maintenance of these data-structures is designed to tolerate server/disk failures. The VDir gives the location of the server that has the GMap for the virtual disk. The GMap has information about which servers are spanned by the virtual disk and the PMap gives the actual translation of a virtual diskoffset to its corresponding physical disk id and physical diskoffset.

Petal has two schemes to take snapshots. The first scheme creates consistent disk images, but this requires that all client applications be paused for a period less than a second. If the pause is unacceptable, then Petal can create a snapshot that would create a disk state similar to a disk-image that would be left after an application has crashed. Running a utility like *fsck* should bring the disk back to a consistent state. All read requests are then translated to the latest epoch number of the virtual address. To do this, the address translation mechanism adds a new piece of information, the *epoch number*. The epoch number records the version of each disk page that ends up associating it either with a particular snapshot of the virtual disk or the virtual disk itself. All writes are done to the current epoch number of the virtual disk, using Copy-on-write operations. These CoWs can fragment the virtual disks badly, since every write to a disk page from an earlier epoch will cause Petal to write the page with the new content to a new location with the current epoch number. Petal does not have a garbage collector to take care of deleting snapshots and/or virtual disks.

Peabody [9] was the next step in storage virtualization. It is a network block storage device that exposes virtual disks. Unlike Petal, Peabody is unconcerned with the management of physical resources. Peabody uses an iSCSI initiator as the backend storage and carves out virtual disks whose sectors are continuously versioned so that any previous state can be recovered. However, maintaining transaction logs of every disk-write and versions of every sector imposes a huge storage overhead. To reduce this overhead, Peabody maintains a *sector store*. The sector store is the actual storage for the virtual disk. There are several methods to implement the sector store. If it is implemented as a contiguous physical piece of the LUN, then there would be no fragmentation (at the level of virtual disks) and the file-system can address the sectors by their physical addresses. However, it also excludes any opportunity for sharing sectors among virtual disks. Another alternative is to implement an address translation mechanism for each virtual disk using a BTree. The would result in fragmenting the virtual disks across the LUN and also incurs address translation costs. However, it does allow sector sharing among virtual disks which would be a significant cost saver. An MD5 hash of each sector is stored to identify sectors with similar content. A hybrid scheme would probably be better. Some virtual disks could be implemented as contiguous pieces of the LUN, and if there was sufficient benefit, then some other virtual disks could

be implemented using the virtual addressing scheme. We don't know if this was ever implemented and/or evaluated, but it did set the stage for deduping. Peabody also had a garbage collector that could reclaim all sectors that belonged to version numbers that were no longer required. However, it could leave the disk in an inconsistent state, since it was not integrated with file-system consistency checks.

Clotho [4] is a block storage abstraction layer that provides data versioning. It can be plugged in the Linux block I/O heirarchy in a single machine, a clustered I/O system, or a SAN. Higher layers (e.g., file system, volume managers etc.) interact with it like it is a standard block device driver. Other block abstraction layers (e.g., RAID) could be plugged under Clotho. Clotho divides the block device into two logical *Segments*, the *Primary Data Segment* and the *Backup Data Segment.* Versioining is done at an *extent* level. The size of an extent could be larger than the standard block size. File systems refer to data by logical block numbers that are converted to physical block numbers using a textitLogical Extent Table(LXT). The primary data segment stores the latest version of the disk, while the backup data segment stores all earlier versions. The data in the backup data segment is strictly read only and would be required only in case of data recovery or historical analysis operations, which are less frequent. Hence, in order to reduce address translations for reading or writing the primary data segment and to avoid fragmenting it, the LXT is divided logically into the primary LXT and the backup LXT. In the primary LXT, a 1-1 correspondence is maintained between the logical extents and the physical extents in the LXT. Thus, only one lookup of the primary LXT is required to convert logical block numbers to physical block numbers. Every time a COW operation is done to preserve an older version of an extent, the older copy is moved to the backup data segment and the corresponding metadata is updated in the backup LXT.

Clotho has automatic garbage collection in the form of the *DeleteVersion()* function. Clotho traverses the primary LXT segment and for every entry that has a version number equal to the delete candidate, changes the version number to the next existing version number. It then traverses the backup LXT segment and frees the related physical extents. One can also dedup or compress extents belonging to different versions in the backup data segment in order to reduce the storage overhead.

The *FAB (Federated Array of Bricks)* [19] project provides a distributed disk array with reliable access to logical volumes. It is built from commodity hardware like disks, CPU and NVRAM, all connected by standard networks such as Ethernet. One can start building the system with a few bricks and increase it to several hundred bricks. Since it all requires just commodity hardware, the cost is kept minimum even with 3-way replication. However, due to high risk of com-

ponent failure, they have implemented a quorum system to co-ordinate operations like snapshot creation/deletion [10], replication, erasure coding, etc. To facilitate reconfigurations and recovery from failures, even the quorum system is designed to be dynamic.

In FAB, the unit of data distribution is a *segment* and its size is 256MB. A *segment group* is the unit of redundant storage. A *volume* is defined as a collection of segment groups. A volume can span a number of bricks. Snapshots are called *versions* and they represent the state of a volume as it was at a certain time. It is a consistent state. Each snapshot and current version of a volume has its own map, which defines the logical address to physical address mapping. All maps are connected by bi-directional links and are used for address-translation and for merging maps in deletion scenarios. When a write is attempted to a data block in a snapshot, the new data block is written to a new location and the physical map of the current version is updated to reflect the change. Thus, as more and more blocks are modified, the volume also gets increasingly fragmented.

In order to create and delete snapshots, a voting system is used in which a majority of the bricks have to agree on whether the snapshot exists in the system and on a timing order for the operation to appear in the globally serializable sequence of snapshot/data operations. To create a snapshot, each participating brick creates a new map with the current timestamp and makes the current volume point to it. Any new writes after that will now be reflected in this new map. To delete a snapshot, each brick merges the content of the map to be deleted (excluding its private map) into its next map and removes the map from the linked list. If a quorum can't be reached, the operations are aborted. One need not pause applications to create/delete snapshots.

The *Logical Volume Manager (LVM)* [24] is another volume management system. They are popular with both home and production systems. There are two versions of LVM: LVM1 and LVM2. One can create volume groups (VG) online from an existing set of physical volumes (PV). These VGs can be resized by absorbing or deleting physical volumes. The resizing is done at the granularity of extents (concatenating or truncating extents), whose size has to be defined. One can also move a VG across the physical volumes and spilt or merge two VGs. To manage these operations, the LVM keeps a metadata header in the head of every physical volume. Each physical volume header has complete information about all volume groups, the identity of other physical volumes, logical volumes, and allocation maps of logical extents to physical extents. Hence, even if one physical volume is lost, all the system metadata can be recovered.

Using LVM1 one can create read-only snapshots of logical volumes. Read-only snapshots are implemeted by creating an exception table. It is used to keep track of the blocks that have been changed. If a block is to be changed at the origin, it is

first copied to the snapshot, marked as copied in the exception table, and then the new data is written to the original volume. If the exception table is as large as the original volume, then the snapshot will always be consistent. Otherwise once the table fills up, the snapshot becomes inconsistent. In order to delete the snapshot, one only needs to delete the exception table.

Using LVM2, one can also create read-write snapshots. That is, if a block is changed, then it is marked in the exception table as used, and it never gets copied from the original volume. Thus, a read-write snapshot cannot be used for backup and recovery. However, it does have other uses. One example is that one can mount a read-write snapshot, and try an experimental program that changes files on that volume. In case the changes don't work, one can unmount the snapshot, discard it, and mount the original volume in its place. It is like creating a gold template that can be used to create volumes for use with Xen [3]. One can create a disk image, then snapshot it and modify the snapshot for a particular domU instance. The only storage used by a snapshot is blocks that were changed in the origin or in the snapshot (exception table). Hence, the majority of the volume will be shared by the domUs.

6.2 File systems

In this section, we will look at two lines of file systems research. The first kind are those that snapshot the entire file system itself and the second kind are file systems that maintain explicit versions of individual files and directories.

In order to snapshot an entire file system, the snapshotting/versioning mechanism has to be incorporated into the basic design of the file system. It cannot be plugged into an already existing file-system design. Example of such file-systems are the Log structured file system (LFS), Write Anywhere File Layout (WAFL) by NetApp and ZFS by Sun.

LFS [18] tries to optimize writes by ensuring that all writes are done sequentially to a massive log-file. The entire disk is divided into *segments*, only one of which is active at any given time. The log is written to this active segment. The files and directories are identified by inode blocks and indirect inode blocks. This concept has been borrowed from the FFS [15]. The difference is that, unlike FFS, in LFS the inode blocks are not written to a fixed location on disk. Any changed data blocks are written to the log. Subsequently, the inode blocks and any indirect inode blocks that need updating are also written to the log. The locations of these inodes are tracked using an inode map. A *checkpoint* is scheduled as frequently as every 30 seconds, during which LFS writes the last known location of the inodes into the inode map. This inode map is kept at two separate fixed locations on disk.

They are updated and used alternatively. Once written, the checkpointed inode map represents the latest consistent image of the disk.

Unless the disk starts running out of segments, LFS will ideally have all versions of data and metadata. Hence, one could think of implementing some kind of recovery system. However, LFS does not offer any such services. It is imperative that segments should be available for re-use in case the disk starts filling up. A garbage-collector called the *LFS cleaner* does the job of reclaiming sparse segments for reuse. To do so, the cleaner needs to know which blocks of the segment are live and also which file these blocks belong to and where they occur in the file, so that the appropriate inode blocks can be updated. The cleaner gets this information from a *segment summary* that is maintained at the end of every segment. The segment summary maintains information about each block in the segment, that is which file it belongs to and its block number in the file. The cleaner reads several segment summaries into memory, gauges which segments have maximum free space and cleans them. Basically it copies out all live blocks into empty spaces in other segments and updates the corresponding inode blocks and then marks the cleaned segments as ready for re-use.

Thus, the only criteria for reclaiming blocks for the cleaner is that the segment in which they occur should be largely free. Hence, there is no guarantee regarding which block versions are maintained and which are removed. One could possibly think of more refined reclamation heuristics, basically designed such that important versions of blocks are kept for a relatively longer time. However, this has not been implemented in LFS as the primary aim was to free segments. Also, writing out files in a log fashion results in a highly fragmented file-system. Even if files were initially written sequentially, as they are updated, the updated blocks will be written to possibly totally different segments and even the cleaner could move them around, further away from the rest of the file. There are no utilities in LFS to solve this problem.

Write Anywhere File Layout (WAFL) [8] [7] by NetApp is a file-system layout that is designed specifically to work in an NFS appliance. The primary focus of its design was intended to provide easy and fast snapshots of the entire file-system. Like FFS, WAFL stores its file and directory metadata in inode blocks. However, unlike FFS, these inode blocks are not stored on fixed locations on disk, but in metadata files. It has three kinds of metadata files, those which store inodes, a block-map file which identifies free blocks and an inode-map file that identifies free inodes. Since the metadata is kept in files, they can be written anywhere and can be of any size. This makes adding and removing disk capacity a trivial operation.

WAFL is organized as a tree of blocks. At the root of this tree is a special inode called the root inode. This root inode describes the inodes of the rest of the file

system, block-map files and inode-map files all included. This root inode is the only entity that has a fixed position since WAFL needs it to boot the system and find files/directories. To create a snapshot, WAFL simply replicates the root inode. The new duplicate root inode will have read pointers to all inodes that the old root inode pointed to. Any writes to old blocks will result in CoW operations on the blocks. Thus the snapshot is preserved. When it is first created, it is created fast and it occupies no more additional space than the old root inode. With more and more writes, the snapshot diverges from the current filesystem.

Earlier, each block in WAFL was represented by 32 bits. When the block was free, all bits were unset. If it was in use by the current file system, bit 0 was set. If it was in use by snapshot 1, bit 1 was set. Similary if it was in use by the second snapshot, then bit 2 was set and so on. Hence, WAFL could not support more than 31 snapshots. However, recently they released a new version of WAFL that can support upto 255 snapshots. They do this using the method of reference counting [14]. Inodes store not only a pointer to the block, but also its associated reference count. This count cannot exceed 255. Using this method also enabled them to incorporate deduping. NetApp calls this their Advanced Single Instance Storage (A-SIS) deduplication. Basically, they maintain a database of fingerprints(checksum) of every data block. All disk writes are intercepted and the new fingerprints of the written disk blocks are recorded in a log. At a later time, these logged fingerprints are compared with the ones in the database, and if any matches are found, then the corresponding data blocks are deduped. Before deleting a block as a duplicate, A-SIS does a byte-by-byte comparison to make sure that the data is indeed the same. A-SIS can also be turned off anytime. However, A-SIS does not differentiate between writable and read-only data. Hence, if a volume is rapidly changing, then A-SIS could cause considerable performance overhead.

NetApp claims that WAFL fragments far slower than other file-systems and in fact handles it better too. Whenever possible, WAFL writes adjacent blocks of a file close to each other. As the disk gets used up, adjacent blocks may not be available. It will still try to place them as close as possible. To facilitate this, WAFL reserves 10% of extra disk space to increase the probability of finding adjacent blocks. Writing data over a network generally has the disadvantage that it is broken into smaller chunks anyways, but WAFL deals with this by first writing out the data in NVRAM, and hence it can group writes together efficiently.

However, these optimizations should work well to avoid fragmentation so long as no snapshots are taken. If snapshots are taken at a reasonable frequency, the disk is bound to get fragmented anyways, in spite of all attempts to allocate close to other file disk blocks. NetApp does provide a defrag utility, wafl scan reallocate, that scans a volume and rewrites the latest version of the file blocks close together. But it is recommended that the system be offline and have no snapshots in order

for the utility to be useful.

ZFS (Zettabyte File System) [17] [2] by Sun is a filesystem that claims to have been designed from a scratch with explicit support for transactions and snapshots. However, their tree like data-structure representation of the file-system and inode representation for files and directories seem very similar to those in WAFL. Every disk-block in use by ZFS can be reached from the root node of the file-system tree. This root node is called the *uberblock*. To take a snapshot, a copy of the uberblock is made with all read-only pointers to the inode blocks below it. Every write operation in ZFS results in a CoW operation. Hence, if one has ample disk space, then one can rollback the whole filesystem as far as possible. ZFS is not a distributed or a parallel file system. ZFS is a local file system and cannot be accessed concurrently from multiple hosts. ZFS has several good features like replication of data blocks, storing a checksum with the block pointer which are then used to recover data in case of corruption.

Recently, some concerns have been expressed regarding their garbage collection method [1]. Every block is tracked by its birth and death - the first snapshot in which it is referenced and the last one. When all snapshots between those two times have been deleted, the block can be reclaimed. Every snapshot maintains a list of blocks that were deleted from it, that is they occur in earlier snapshots, but not in this one. Every time a block is deleted from the main-line file-system, a routine checks to see if it occurs in any of its snapshots. If it does occur in any of the snapshots, then it is added to list of killed blocks. Otherwise the block is reclaimed. The fact that it is reclaimed is written to a log-file that is maintained explicitly to record block allocations and final-deletions. This log is maintained for logical partitions of the disk-space. Every once in a while, the log is replayed on block bitmaps and thus updated to reflect any freed disk space.

The linear sequence of the snapshots is central to the management of disk space in ZFS. A block can only be referenced by the main file-system or by one of its snapshots. However, ZFS also gives the option to create a *clone file-system* from an existing snapshot. Since ZFS supports only one local file-system, there can be only one main-line file system. Creating the clone will create references to blocks which violate the linear nature of references. If this clone is now made the main-line file-system, then there will be no accurate way to track block lifetime. In fact, there is a distinct possibility that one can end up with an undeletable snapshot that one can't get rid of until all clones have been backed out. It is not clear whether ZFS has any utilities to combat long-term fragmentation. Although the claim is that ZFS's block allocation policies are designed so as to write file blocks as close as possible. Yet, always writing out of place is certain to cause significant fragmentation.

6.3 Summary

As discussed above, each storage system has its own scheme of managing metadata. Depending on this scheme, each has its own way of managing the issue of storage reclamation and fragmentation. Each scheme has its own advantages and disadvantages. Parallax also has its own scheme of managing metadata. Accordingly we have developed utilities for block reclamation and remapping to counter fragmentation.

System	Block / File Level	Reclamation	Defragmentation	Deduping
Logical Disk	Block	Yes	Yes	Yes
Petal	Block	No	No	No
Peabody	Block	Yes	Yes	Yes
Clotho	Block	Yes	Yes	Yes
FAB	Block	Yes	No	No
LVM1	Block	Yes	Yes	No
LVM2	Block	Yes	Yes	No
LFS	File	Yes	No	No
WAFL	File	Yes	Yes	Yes
ZFS	File	Yes	Yes	No

Chapter 7

Future work and conclusion

Techniques to remap and reclaim disk blocks have been implemented for the Parallax storage system. Using these tools, utilities to create superpages and defragment free disk space in the blockstore have been implemented. All these operations work only with the Parallax metadata and hence they have very little interference with the normal operations of the Parallax instances. This along with the fact that utilities work with disk blocks, makes them totally OS and file-system agnostic. Depending on the size of the blockstore, available memory resources and time constraints, a system administrator can either use these tools on a blockstore-wide scale or run them one extent at a time. Although we have a design for a de-duping mechanism for the blockstore, due to time constraints it has not been implemented yet. Implementing this mechanism will definitely be an added benefit. Added to that the fact that the global garbage collector gets to see the metadata of the entire blockstore, can be used to more advantage. One can develop a fingerprinting method that will facilitate de-duping as well as data restoration. While scanning the blockstore for unreferenced blocks, the garbage collector can calculate the fingerprint of each block. When the blocks become read-only these fingerprints can be used for de-duping duplicate read-only blocks. The fingerprints can also be used to detect data-corruption (for read-only disk blocks). Corrupted data can be restored from backups. The existing utilities to defragment extents can be improved to use more intelligent means of rearranging blocks in the extent so that minimum remapping is required. Remapping writable disk blocks requires some co-operation from the Parallax instances. One can work to make these requests to be of low priority to the Parallax instances, so that the remapper does not interfere much with the normal functioning of the Parallax instances.

Bibliography

[1] Limitations of zfs. http://marc.info/?l=linux-fsdevel&m=113243953111393&w=2.

[2] Zfs administration guide. http://opensolaris.org/os/community/zfs/docs/zfsadmin.pdf.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles.*, October 2003.

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

[5] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS-1)*, Oct. 2004.

[6] D. h T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. hael J. Feeley, N. C. H. hinson, and A. Warfield. Parallax: virtual disks for virtual machines. *SIGOPS Oper. Syst. Rev.*, 42(4):41–54, 2008.

[7] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. http://www.netapp.com/library/tr/3002.pdf.

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

[9] C. B. M. III and D. Grunwald. Peabody: The time travelling disk. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS03)*, 2003.

[10] M. Ji. Instant snapshots in a federated array of bricks. In *Technical Report HPL-2005-15, HP Laboratories*, 2005.

[11] Joe Spurr. Deduping: an essential backup tool in the data center? http://searchdatacenter.techtarget.com/originalContent/0,289142,sid80_gci1192939,00.html.

[12] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *The Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[13] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI 2004)*, pages 17–30, 2004.

[14] B. Lewis. A-sis: Deduplication comes of age. http://www.netapp.com/news/techontap/dedupe.html.

[15] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for unix. In *ACM Transactions on Computer Systems, Vol. 2, No. 3*, pages 181–197, 1984.

[16] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: Virtual disks for virtual machines. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '08)*, April 2008.

[17] O. Rodeh and A. Teperman. zfs – a scalable distributed file system using object disks. In *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, page 207, Washington, DC, USA, 2003. IEEE Computer Society.

[18] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles and the February 1992 ACM Transactions on Computer Systems.*, February 1992.

[19] Y. Saito, S. Frlund, A. Veitch, A. Merchant, and S. Spence. Fab: Building distributed enterprise disk arrays from commodity components. http://hpl.hp.com/research/ssp/papers/2004-10-ASPLOS-FAB.pdf, 2004.

[20] A. Warfield. *Virtual Devices for Virtual Machines*. PhD thesis, University of Cambridge, 2006.

[21] Wiebren de Jonge and M. Frans Kaashoek and Wilson C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. http://www.cs.utah.edu/ wilson/papers/logical-disk.pdf.

[22] Wikipedia.org. Fragmentation Overview. http://en.wikipedia.org/wiki/Fragmentation_computer.

[23] Wikipedia.org. Garbage collection Overview. http://en.wikipedia.org/wiki/Garbage_collection_computer_science.

[24] Wikipedia.org. Logical Volume Manager (Linux). http://en.wikipedia.org/wiki/Lvm.

Chapter 8

Statement of co-authorship

All material in Chapter 2 is included only to provide the background necessary to understand the rest of the thesis. It is entirely composed of excerpts from [6].