Parallax

volume management for virtual machines

by

Dutch Thomassen Meyer

B.Sc. in Computer Science, University of Washington, 2002

B.A. in Mathematics, University of Washington, 2002

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

Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

October 2008

Abstract

Parallax is a distributed storage system that uses virtualization to provide storage facilities specifically for virtual environments. The system employs a novel architecture in which storage features that have traditionally been implemented directly on high-end storage arrays and switches are relocated into a federation of *storage VMs*, sharing the same physical hosts as the VMs that they serve. This architecture retains the single administrative domain and OS agnosticism achieved by array- and switch-based approaches, while lowering the bar on hardware requirements and facilitating the development of new features. Parallax offers a comprehensive set of storage features including frequent, low-overhead snapshot of virtual disks, the “gold-mastering” of template images, and the ability to use local disks as a persistent cache to dampen burst demand on networked storage.

Table of Contents

List of Tables

List of Figures

Acknowledgments

My supervisors Andrew Warfield and Norm Hutchinson each contributed immeasurably to my efforts here. Their advice, help, support, and enthusiasm has made the work what it is today, and has made the process fun as a bonus. My partner Hannah continues to be my inspiration, and deserves full credit for my embarking on this adventure in the first place. Thanks to my parents for their unerring support.

Geoffrey Lefebvre and Brendan Cully both made significant contributions to Parallax and the Eurosys paper.

I feel very fortunate to have been guided by so many excellent teachers. Steve Wolfman, Rachel Pottinger, and Nathan Kutz most notably.

Thanks to my many friends who provided non-contributory but otherwise entertaining conversation during the many hours it took to create this work. Thank you also to my two cats, who have shown uncharacteristic tolerance for my attention to silly computers.

Figure 3.3 was created by Andrew Warfield, and appears in his thesis as well as the original Parallax paper at HotOS.

Chapter 1

Introduction

In current deployments of hardware virtualization, storage facilities severely limit the flexibility and freedom of virtual machines.

Perhaps the most important aspect of the resurgence of virtualization is that it allows complex modern software – the operating system and applications that run on a computer – to be completely encapsulated in a virtual machine. The encapsulation afforded by the VM abstraction is without parallel: it allows whole systems to easily be quickly provisioned, duplicated, rewound, and migrated across physical hosts without otherwise disrupting execution. The benefits of this encapsulation have been demonstrated by numerous interesting research projects that allow VMs to travel through space [2, 15, 31], time [5, 14, 39], and to be otherwise manipulated [37].

Unfortunately, storage has not experienced a rapid evolution in support of virtualization, as we have seen in both system software and platform hardware such as CPUs and chipsets. While "storage virtualization" is widely available, the term is something of a misnomer in that it is largely used to describe the aggregation and repartitioning of disks at very coarse time scales for use by physical machines. VM deployments are limited by modern storage systems because the storage primitives available for use by VMs are not nearly as nimble as the VMs that consume them. Operations such as remapping volumes across hosts and checkpointing disks are frequently clumsy and esoteric on high-end storage systems, and are simply unavailable on lower-end commodity storage hardware.

This thesis investigates the nature of these storage limitations in the context of hardware virtualization. We contend that advanced storage features can be incorporated into virtualized environments, and done so with minimal overheads. To validate this position, we describe the design and implementation of the Parallax storage system for the Xen virtual machine monitor.

Parallax is effectively a cluster volume manager for virtual disks: each physical host shares access to a single, globally visible block device, which is collaboratively managed to present individual virtual disk images (VDIs) to VMs. Parallax attempts to *use* virtualization in order to provide advanced storage services *for* virtual machines. Parallax takes advantage of the structure of a virtualized environment to move storage enhancements that are traditionally implemented on arrays or in storage switches out onto the consuming physical hosts. Each host in a Parallax-based cluster runs a *storage VM*, which is a virtual appliance [30] specifically for storage that serves virtual disks to the VMs that run alongside it. The encapsulation provided by virtualization allows these storage features to remain behind the block interface, agnostic to the OS that uses them, while moving their implementation into a context that facilitates improvement and innovation.

The system has been designed with considerations specific to the emerging uses of virtual machines, resulting in some particularly unusual directions. Most notably, we desire very frequent (i.e., every 10ms) snapshots. This capability allows the fine-grained rewinding of the disk to arbitrary points in its history, which makes virtual machine snapshots much more powerful. In addition, since our goal is to present virtual disks to VMs, we intentionally do not support sharing of VDIs. This eliminates the requirement for a distributed lock manager, and dramatically simplifies our design. The VM-based design also allows Parallax to be implemented in user-space, allowing for a very fast development cycle.

This work was published at the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems in 2008. [20]

Chapter 2

Related Work

2.1 Storage in Support of Virtualization

Despite the many storage-related challenges present in virtualized environments, we are aware of only two other storage systems that cater specifically to VM deployments: Ventana [24] and VMware's VMFS [34].

Ventana attempts to provide support for virtual machines at the file system level, effectively virtualizing the file system namespace and allowing individual VMs to share underlying file objects where possible. File system virtualization is a fundamentally different approach to the block-level virtualization provided by Parallax. Ventana provides an improved degree of "transparency" into the contents of virtual disks, but sacrifices generality in order to achieve it. Windows VMs, for instance, cannot be hosted off of the NFS interface that the Ventana server presents. We discuss these block-versus-file trade-offs at more length in Chapter 3.1.1. Ventana's authors do not evaluate its performance, but do mention that the system suffers as the number of branches (equivalent to snapshots in Parallax) increases, while parallax does not suffer from an analogous limitation.

VMFS is a commercial block-level storage virtualization system intended for use with VMware ESX. VMFS is certainly the most similar known system to Parallax; both approaches specifically address virtualized environments by providing distributed facilities to convert one large shared volume into a number of virtual disks for use by VMs. VMFS acts largely as a cluster file system, specifically tuned to host image files. Virtual disks themselves are stored within VMFS as VMDK [33] images. VMDK is a image format for virtual disks, similar to QCOW [19] and VHD [22], which provides sparseness and allows images to be "chained". The performance of chained images decays linearly as the number of snapshots increases in addition to imposing overheads for open file handles and in-memory caches for each open image. In addition to chaining capabilities provided by VMDK, VMFS employs a redo log-based checkpoint facility that has considerable performance limitations [35]. Specifically, gold mastering and branching volumes are not well supported by redo logs. Parallax provides fine-grained sharing and snapshots as core aspects of its design, such that performance can be maintained independent of the number or depth of snapshots.

Another approach that addresses issues similar to those of Parallax has been undertaken in recent work by the Emulab developers at the University of Utah [6]. In order to provide snapshots for Xen-based VMs, the researchers modified Linux LVM (Logical Volume Management) to provide a branching facility. No details are currently available on this implementation.

2.2 Snapshots

Beyond VM-specific approaches, many other systems provide virtual volumes in block-level storage, most notably FAB [8] and its predecessor Petal [16]. Both systems, particularly FAB, aim to provide a SAN-like feature set at a low total system cost. Both systems also support snapshots; the ability to snapshot in FAB is best manifest in Olive [1, 12].

Parallax differs from these prior block-level virtual disk systems in three ways. First, Parallax assumes the availability of a single shared block device, such as an iSCSI or FiberChannel LUN, NFS-based file, or Petal-like virtual disk, while FAB and similar systems compose a shared volume from a federation of storage devices. Whereas other systems must focus on coordination among distributed storage nodes, Parallax focuses on coordinating distributed clients sharing a network attached disk. By relying on virtualized storage in this manner, we address fundamentally different challenges. Second, because we provide the abstraction of a local disk to virtualized guest operating systems, we can make a reasonable assumption that disk images will be single-writer. This simplifies our system and enables aggressive performance optimization. Third, Parallax's design and virtualized infrastructure enables us to rethink the traditional boundaries of a network storage system. In addition, among block-level virtualization systems, only Olive [1] has a snapshot of comparable performance to ours. Olive's snapshots have more complicated failure semantics than those of Parallax and the system imposes delays on write operations issued during a snapshot.

WAFL [10] has very similar goals to those of Parallax, and as a consequence results in a very similar approach to block address virtualization. WAFL is concerned with maintaining historical versions of the files in a network-attached storage system. It uses tree-based mapping structures to represent divergences between snapshots and to allow data to be written to arbitrary locations on the underlying disk. Parallax applies similar techniques at a finer granularity allowing snapshots of individual virtual disks, effectively the analogue of a single file in a WAFL environment. Moreover, Parallax has been designed to support arbitrary numbers of snapshots, as opposed to the hard limit of 255 snapshots available from current WAFL-based systems.

2.3 Data Protection

Since Parallax is principally designed as a volume management (as opposed to data protection) system, we do not consider backup or archival storage at great depth. Still, because Parallax can feasibly supplant many of the features of these systems, we consider a select few.

Continuous Data Protection is a fine-grained solution to data protection where every modification to a file system or disk is recorded. Its commercial success is in part due to the need for strong corporate data retention policies. Many companies offer systems with protection at or near this level, including Microsoft [21] and IBM [11]. Parallax's snapshot system is better described as *near* continuous, even though it can be configured to provide protection at the finest granularity. We evaluate this potential briefly in Chapter 5.1.2.

Venti [26] is an archival storage system for use with Plan 9 [25] and forms the basis of Fossil, Plan 9's file system. Venti is a content addressed storage system at the block level designed to provide permanent and transparent backups with performance comparable to that of a traditional file system. Snapshots, compression, and variable sized blocks are also supported. Venti does not support delete operations, so use of the system necessitates adopting write-once semantics. The on-disk structure shares elements with Parallax's extents, and snapshots use a similar tree structure. Venti also stores significant meta-data along with each block on disk including an on-disk directory structure with each group of blocks.

Elephant [29] explored protecting files from accidental modifications and deletion. In contrast with Parallax, user driven data retention policies are employed. This is possible in Elephant because of the file-system, as opposed to block level, interface. However, policy could be added to Parallax at the volume level. While course-grained, this would enable applying different policies to, for example, home and system directories, which could capture most common policy differences. By tracking the differences between multiple versions of files, Elephant is also able to determine likely versions to be deleted. Even though there is significantly less semantic information at the block level, Parallax may be able to employ similar techniques.

2.4 Other Approaches

Many other systems have provided snapshots as a storage system feature, ranging from file system-level support in ZFS [28] to block-level volume management systems like LVM2 [27]. In every case these systems suffer from either a limited range of supported environments, severely limited snapshot functionality, or both. These limitations make them ill-suited for general deployment in virtualized storage infrastructures.

Chapter 3

System Architecture

3.1 Clustered Storage Appliances

Figure 3.1 presents a high-level view of the structure of a Parallax-based cluster. 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. From an architectural perspective, this structure makes Parallax unique: 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.

In this chapter, we describe the set of specific design considerations that have guided our implementation, and then present an overview of the system's structure.

3.1.1 Design Considerations

Parallax's design is based on four high-level themes:

Agnosticism and isolation. Parallax is implemented as a collaborative set of storage *appliances*; as shown in Figure 3.1, each physical host in a cluster contains a *storage VM* that 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 has been used previously to insulate running VMs from device driver crashes [7,

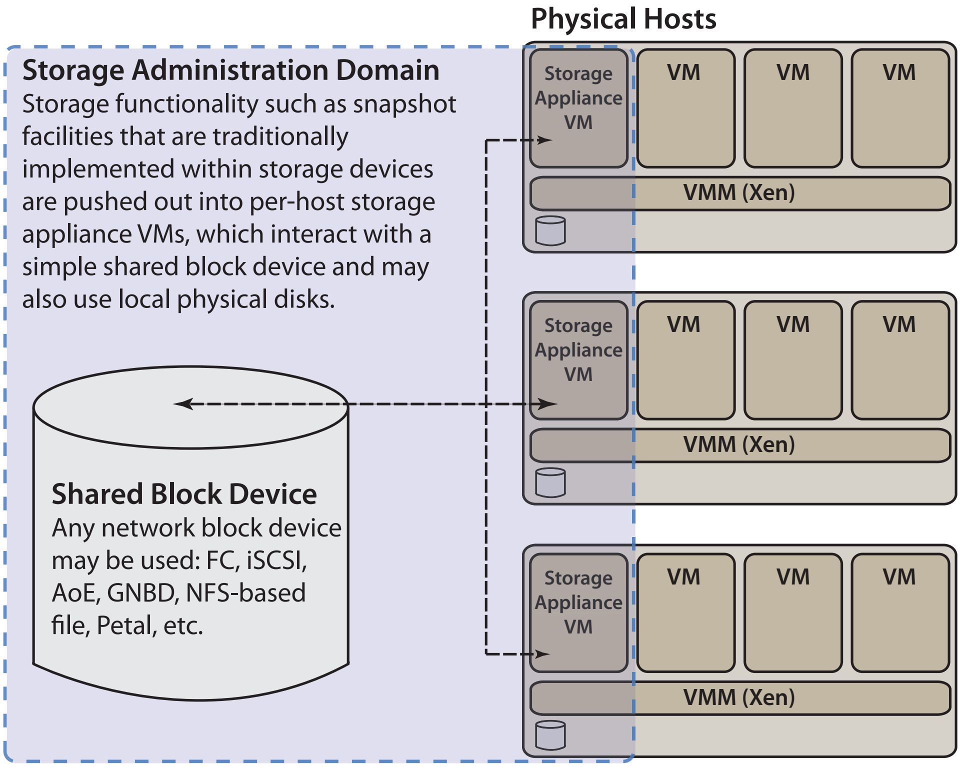
Physical Hosts
Storage Administration Domain
Storage functionality such as snapshot facilities that are traditionally implemented within storage devices are pushed out into per-host storage appliance VMs, which interact with a simple shared block device and may also use local physical disks.
Shared Block Device
Any network block device may be used: FC, iSCSI, AoE, GNBD, NFS-based file, Petal, etc.
Storage Appliance VM
VM
VM
VM
VMM (Xen)
Storage Appliance VM
VM
VM
VM
VMM (Xen)
Storage Appliance VM
VM
VM
VM
VMM (Xen)

Figure 3.1: 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.

17], 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 can manage 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 guest VMs, Parallax operates at the block, rather than file, level. Block-level virtualization provides a narrower interface and allows Parallax to present simple virtual disks to individual VMs. While virtualization at the block level yields an agnostic and simple 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, as discussed in Chapter 4.1.1.

Minimize lock management. Distributed storage has historically implied some degree of concurrency control. Write sharing of disk data, especially at the file system level, typically involves the introduction of some form of distributed lock manager. Lock management is a very complex service to provide in a distributed setting and is notorious for difficult failure cases and recovery mechanisms. Moreover, although write conflict resolution is a well-investigated area of systems research, it is one for which no general solutions exist.

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 ensures that each VDI 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 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. In existing storage systems, the ability to snapshot storage has typically been implemented as an afterthought, and for very limited use cases such as the support of backup services. Post-hoc implementations of snapshot facilities are typically complex, involve inefficient techniques such as redo logs [34], or impose hard limits on the maximum number of snapshots [10]. Our belief in constructing Parallax has been that the ability to take and preserve very frequent, low-overhead snapshots is an enabling storage feature for a wide variety of VM-related applications such as high-availability, debugging, and continuous data protection. As such, the system 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.

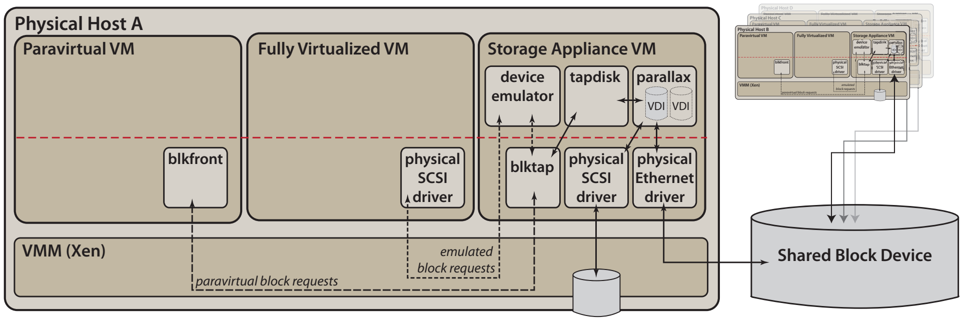

Figure 3.2: Overview of the Parallax system architecture.

3.1.2 System Structure

Figure 3.2 shows an overview of Parallax's architecture and allows a brief discussion of components that are presented in more detail throughout the remainder of the paper.

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, allowing functionality that is currently implemented within enterprise storage hardware to be pushed out and implemented on individual hosts. The result is that advanced storage features, such as snapshot facilities, may be implemented in software and delivered above commodity network storage targets.

Parallax itself runs as a user-level daemon in the Storage Appliance VM, and uses Xen's *block tap* driver [38] 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. We place no restrictions as to what this device need be, so long as it is sharable 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 this 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. Including physical block device drivers in the storage VM allows a storage administrator the ability to do live upgrades of block device drivers in an active cluster. This is an area of future exploration for us, but a very similar approach has been described previously [7].

3.2 Virtual Disk Images

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. Table 3.1 presents a summary of the administrative operations that may be performed on VDIs; these operations are available through the command line of the storage VM. There are three core operations, allowing VDIs to be created, deleted, and snapshotted. These are the only operations required to actively manage VDIs; once created, they may be attached to VMs as would any other block device. In addition to the three core operations, Parallax provides some convenience operations that allow an administrator to view catalogues of VDIs, snapshots associated with a particular VDI, and to "tag" particular snapshots with a human-readable alias, facilitating creation of new VDIs based on that snapshot in the future. An additional convenience function produces a simple visualization of the VDIs in the system as well as tagged snapshots.

create(*name, [snapshot]***)** → *VDI_id*	Create a new VDI, optionally based on an existing snapshot. The provided name is for administrative convenience, whereas the returned VDI identifier is globally unique.
delete(*VDI_id***)**	Mark the given VDI as deleted. When the garbage collector is run, the VDI and all snapshots are freed.
snapshot(*VDI_id***)** → *snap_id*	Request a snapshot of the given VDI.
list() → *VDI_list*	Return a list of VDIs in the system.
snap_list(*VDI_id***)** → *snap_list*	Return the log of snapshots associated with the specified VDI.
snap_label(*snap_id, name***)**	Label the given snapshot with a human-readable name.
tree() → *(tree view of VDIs)*	Generate a graph of the current system-wide VDI tree (see Figure 3.4 for an example.)

Table 3.1: VDI Administrative Interfaces.

3.2.1 VDIs as Block Address Spaces

In order to achieve the design goals that have been outlined regarding VDI functionality, in particular the ability to take fast and frequent snapshots, Parallax borrows heavily from techniques used to manage virtual memory. 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 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, and 4K data blocks yields $2^{9*3} * 2^{12} = 2^{39} = 512\text{GB}$). Adding a level to the radix tree extends this by a factor of 2^9 to 256TB 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 can be shared across descendant radix trees in order to implement snapshots.

3.3 Snapshots

A snapshot in Parallax is a read-only image of an entire disk at a particular point in time. Like many other systems, Parallax always ensures that snapshots are *crash consistent*, which means that snapshots will capture a file system state that could have resulted from a crash [1] [16] [23] [36] [24]. While this may necessitate running an application or file system level disk check such as fsck, it is unlikely that any block-level system can offer stronger guarantees about consistency without coordination with applications and file systems.

3.3.1 Coordination

Snapshots can be taken of a disk that is not in use, or they can be taken during normal operation. In this latter case, the snapshot semantics are strictly *asynchronous*; snapshots are issued directly into the stream of I/O requests in a manner similar to write barriers. It is often advantageous (thought not necessary) for such asynchronous snapshots to be coordinated with some external event. For example, in order to make a backup, a user may want to create a snapshot after a file is saved. Due to buffer caches, there is little guarantee that the saved file will actually be present in a snapshot. Two methods exist for establishing this coordination.

First, the disk may be quiesced prior to taking a snapshot. This can be further categorized by the level at which the coordination occurs. The application, file system, or block level can each be placed into a consistent state independently. Quiescing at a higher level (i.e., application) establishes the strongest consistency guarantees, but at the greatest performance impact. Parallax is designed to support consistency at the lowest possible level — its own metadata. This design allows for the highest possible performance. It also provides a framework upon which higher level consistency may be added, by making Parallax's capabilities visible at higher stages in the storage stack.

Alternatively, by capturing the memory and CPU state along with the disk image, the state of an entire system can be recorded. In this scenario, the system can be resumed at any checkpoint, and any requests that have not reached disk will be present in memory. This approach lends itself to virtual machines, because of the relative ease with which their internal state can be captured and restarted. In a prototype system for frequent and long running state capture, we have paired Parallax with a system called Remus [4]. Remus captures memory and CPU state at frequencies similar to Parallax, and flushes this data to a dedicated Parallax disk. We then issue coordinated snapshots of both the memory "disk" and the storage volume. This enables complete system state capture at very high frequency.

3.3.2 Implementation

To implement snapshots, we use the high-order bit of block addresses in the radix tree 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. In taking a snapshot, Parallax simply copies the root block of the radix tree and marks all of its references as read-only. The original root need not be modified as it is only referenced by a snapshot log that is implicitly read-only. The entire process usually requires just three block-write operations, two of which can be performed concurrently.

The result of a snapshot is illustrated in Figure 3.3. The figure shows a simplified radix tree mapping six-bit block addresses with two address bits per radix page. In the figure, a VDI has had a snapshot taken, and subsequently had a block

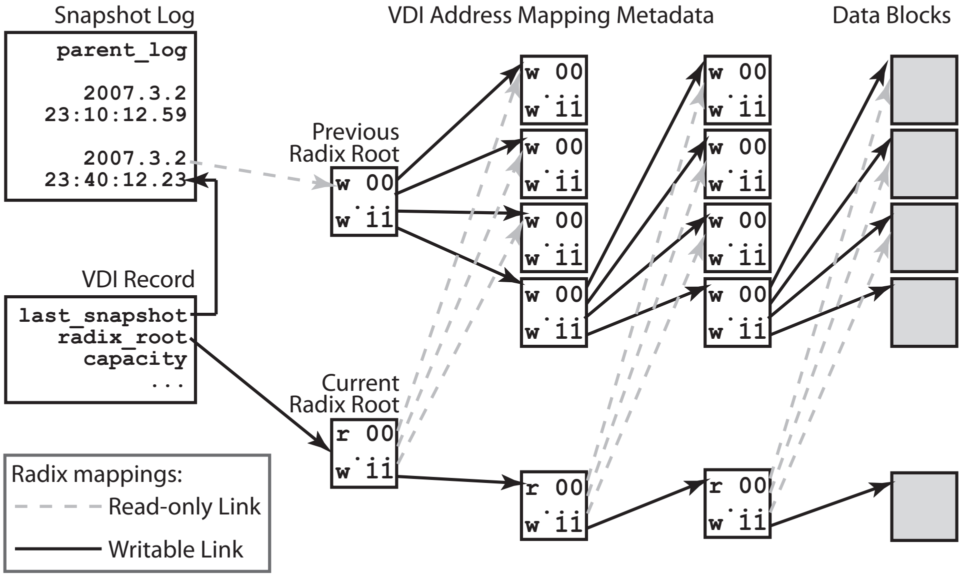

Figure 3.3: Parallax radix tree (simplified with short addresses) and COW behaviour.

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 no 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.

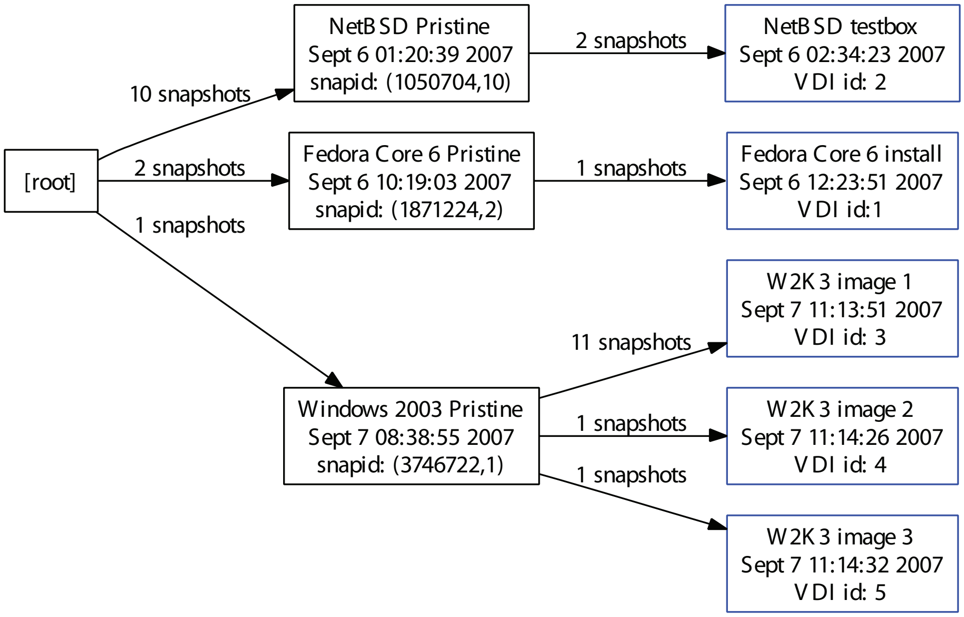

Figure 3.4: VDI Tree View—Visualizing the Snapshot Log.

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 the invariant that radix roots in 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 writable 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 disk. When a new VDI is created from an existing snapshot, its snapshot log is made to link back to 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 3.4. 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 snapshot logs are largely sufficient for Parallax to operate. However, these structures present several interesting challenges that are addressed in the following chapters. Chapter 3.4 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. Parallax's radix trees, described above, are very fine grained, and risk the introduction of a great deal of per-request latency. The system takes considerable effort, described in Chapter 4.1, to manage the request stream to eliminate these overheads.

3.3.3 Fault Cases

Providing crash consistency for snapshots in a distributed system can be difficult. Many of the coordination challenges faced by other systems [1] are obviated by Parallax's client-oriented design. However, since Parallax's snapshots are highly asynchronous, care must be taken with outstanding requests to the block layer. We discuss these challenges and their solution is Chapter 4.2;

3.4 The Shared Blockstore

Traditionally, distributed storage systems rely on distributed lock management to handle concurrent access to shared data structures within the cluster. In designing Parallax, we have attempted to avoid distributed locking wherever 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 our management of the shared blockstore in determining how data is laid out on disk and where locking is required.

3.4.1 Extent-Based Access

The physical blockstore is divided, at start of day, into fixed-size extents. These extents are large (2GB in our current implementation) and represent a lockable

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

single-allocator region. “Allocators” at the this 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. In addition, it helps preserve linearity in the placement of data blocks, by preventing metadata from becoming intermingled with data. All 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 room for additional allocation beyond the capacity of active extents. We will likely optimize this further in the future by arranging for connected Parallax instances to each lock a share of the unallocated extents, further reducing the already very limited need for allocation-related locking.

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. The second of these properties is a consequence of the fact that a given (data or metadata) block is only ever marked writable within a *single* radix tree.

[2] This is a white lie — there is a very coarse-grained lock on the allocation bitmaps used with the garbage collector, see Chapter 3.4.3.

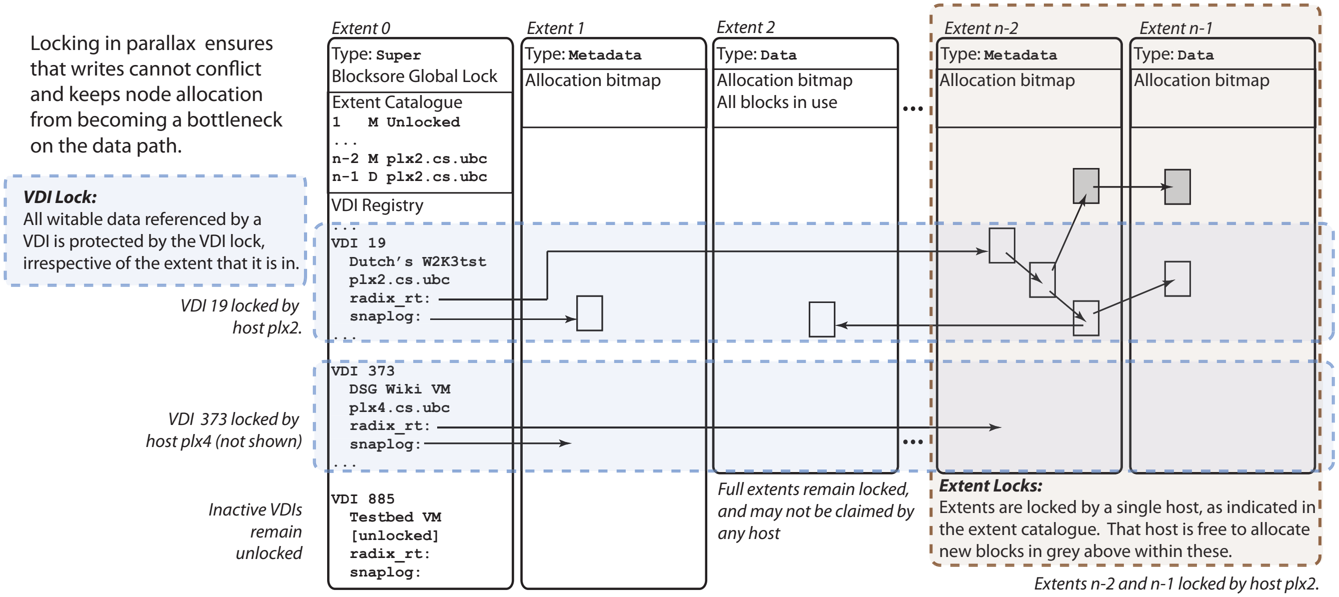
Locking in parallax ensures that writes cannot conflict and keeps node allocation from becoming a bottleneck on the data path.
VDI Lock:
All witable data referenced by a VDI is protected by the VDI lock, irrespective of the extent that it is in.
VDI 19 locked by host plx2.
VDI 373 locked by host plx4 (not shown)
Inactive VDIs remain unlocked
Extent 0
Type: Super
Blocksore Global Lock
Extent Catalogue
1 M Unlocked
...
n-2 M plx2.cs.ubc
n-1 D plx2.cs.ubc
VDI Registry
...
VDI 19
Dutch's W2K3tst
plx2.cs.ubc
radix_rt:
snaplog:
...
VDI 373
DSG Wiki VM
plx4.cs.ubc
radix_rt:
snaplog:
...
VDI 885
Testbed VM
[unlocked]
radix_rt:
snaplog:
Extent 1
Type: Metadata
Allocation bitmap
Extent 2
Type: Data
Allocation bitmap
All blocks in use
...
Full extents remain locked, and may not be claimed by any host
Extent n-2
Type: Metadata
Allocation bitmap
Extent n-1
Type: Data
Allocation bitmap
Extent Locks:
Extents are locked by a single host, as indicated in the extent catalogue. That host is free to allocate new blocks in grey above within these.
Extents n-2 and n-1 locked by host plx2.

Figure 3.5: Blockstore Layout.

Figure 3.5 illustrates the structure of Parallax's blockstore, and demonstrates how extent locks allow a host to act as the single writer for new allocations within a given extent, while VDI locks allow a host write access to allocated VDI blocks across all extents on the blockstore.

3.4.2 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 from a different extent.

The independence that this policy affords to each Parallax instance improves the scalability and reliability of the entire cluster. The scalability benefits are clear: with no lock manager acting as a bottleneck, the only limiting factor for throughput is the shared storage medium. Reliability is improved because Parallax instances can continue running in the absence of a lock manager as long as they have free space in the extents they have already claimed. Nodes that anticipate heavy block allocation can simply lock extra extents in advance.

In the case that a Parallax instance has exhausted its free space or cannot access the shared block device, the local disk cache described in Chapter 5.1.2 could be used for temporary storage until connectivity is restored.

Because it is unnecessary for data access, the lock manager can be very simple.

In our implementation we designate a single node to be the lock manager. When the manager 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 lock manager with lock requests for extents or VDIs. Failure recovery is not currently automated, but the system's tolerance for lock manager failure makes manual recovery feasible.

3.4.3 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 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.

Parallax's garbage collector is described as Algorithm 1. It is similar to a mark-and-sweep collector, except that it has a fixed, static set of passes. This is possible because we know that the maximum length of any chain of references is the height of the radix trees. 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. The only blocks that the collector considers as candidates for deallocation are those that were marked as allocated in the checkpoint

Algorithm 1 The Parallax Garbage Collector

```
1. Checkpoint Block Allocation Maps (BMaps) of extents.
2. Initialize the Reachability Map (RMap) to zero.
3. For each VDI in the VDI registry:
   If VDI is not marked as deleted:
       Mark its 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 (in the 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 the RMap.
7. For each extent:
   Lock the BMap.
   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.
   Unlock the BMap.
```

taken in Step 1 (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.

We discuss the performance of our garbage collector during our system evaluation in Chapter 5.1.2.

3.4.4 Radix Node Cache

Parallax relies on the caching of radix node blocks to mitigate the overheads associated with radix tree traversal. We cache only the intermediate radix nodes (i.e., Parallax's metadata) not the actual data stored on disk. Our cache policy is effectively write-through; however, the cache also ensures the correct ordering of IO requests during write operations through a generic dependency tracking mechanism. We discuss this aspect of the design at more length in Chapter 4.2.1.

The cache is responsible for mitigating the performance impact of reads of the radix tree. There are two aspects of Parallax's design that makes this possible. First, single-writer semantics of virtual disk images remove the need for any cache coherency mechanisms. Second, the ratio of data to metadata is approximately 512:1, which makes caching a large proportion of the radix node blocks for any virtual disk feasible. With our current default cache size of just 64MB we can fully accommodate a working set of nearly 32GB of data. We expect that a production-grade Parallax system will be able to dedicate a larger portion of its RAM to caching radix nodes. To maintain good performance, our cache must be scaled linearly with the working set of data.

The cache replacement algorithm is a simple numerical hash based on block address. Since this has the possibility of thrashing or evicting a valuable root node in favor of a low-level radix node, we have plan to implement and evaluate a more sophisticated page replacement algorithm in the future. For performance and consistency reasons the root node is always kept in-cache.

3.4.5 Local Disk Cache

Our local disk cache leverages our client-oriented approach in order to allow persistent data to be written by a Parallax host without contacting the primary shared storage. The current implementation is in a prototype phase. We envision several eventual applications for this approach. In this thesis, we discuss an approach to mitigate the effects of degraded network operation by temporarily using the disk as a cache. We evaluate this technique in Chapter 5.1.2. In the future we plan to use this mechanism to support fully disconnected operation of a physical host.

The local disk cache is designed as a log-based ring of write requests that would have otherwise been sent to the primary storage system. The write records are stored in a file or raw partition on the local disk. In addition to its normal processing, Parallax consumes write records from the front of the log and sends them to the primary storage system. By maintaining the same write ordering we ensure that the consistency of the remote storage system is maintained. When the log is full, records must be flushed to primary storage before request processing can continue. In the event of a physical host crash, all virtual disks (which remain locked) must be quiesced before the virtual disk can be remounted.

A drawback to this approach is that it incorporates the physical host's local disk into the failure model of the storage system. Users must be willing to accept the minimum of the reliability of the local disk and that of the storage system. For many users, this will mean that a single disk is unacceptable as a persistent cache, and that the cache must be stored redundantly to multiple disks. For example, dual hard drives with RAID 1 on each physical machine may suffice for some users.

Chapter 4

Pipeline Operation

In addition to the basic architecture discussed in Chapter 3, Parallax employs several interesting optimizations to ensure correct and efficient operation. In each of the following cases, we consider Parallax to act on a pipeline of block I/O requests. At each step we make every attempt to push the largest possible number of requests through the pipeline, while balancing against the need to ensure correct operation in the face of error.

4.1 Requests in the Pipeline

While Parallax's fine-grained address mapping trees provide efficient snapshots and sharing of block data, they risk imposing a high performance cost on block requests. At worst, accessing a block on disk can incur three dependent metadata reads that precede the actual data access. Given the high cost of access to block devices, it is critical to reduce this overhead. However, since Parallax is presenting virtual block devices to the VMs that use it; we must be careful to provide the semantics that OSes expect from their disks. We now discuss how Parallax aggressively optimizes the block request stream while ensuring the correct handling of block data.

4.1.1 Consistency and Durability

VDIs appear as SCSI-like block devices that respond to I/O requests with the same semantics as a local disk. Only 64 requests are exposed to Parallax at a time and these in-flight requests may complete in any order. Parallax does not support a tag or barrier operation, although this is an area of interest for future work. Currently guest OSes must allow the request queue to drain in order to ensure that all issued writes have hit the disk. We expect that the addition of barriers will improve our performance by better saturating the request pipeline.

While in-flight requests may complete out of order, Parallax must manage considerable internal ordering complexity. Consider that each *logical* block request, issued by a guest, will result in a number of *component* block requests to read, and potentially update metadata and finally data on disk. Parallax must ensure that these component requests are carefully ordered to provide both the consistency and durability expected by the VM. These expectations may be satisfied through the following two invariants:

1. Durability is the guest expectation that acknowledged write requests indicate that data has been written to disk. [1] To provide durability, Parallax cannot notify the guest operating system that a logical I/O request has completed until all component I/O requests have committed to physical storage.

2. Consistency is the guest expectation that its individual block requests are atomic—that while system crashes may lose in-flight logical requests, Parallax will not leave its own metadata in an invalid state.

In satisfying both of these properties, Parallax uses what are effectively soft updates [18]. All dependent data and metadata are written to disk before updates are made that reference this data from the radix tree. This ordering falls out of the copy-on-write structure of the mapping trees, described in the previous chapter. For any VDI, all address lookups must start at the radix root. When a write is being made, either all references from the top of the tree down to the data block being written are writable, in which case the write may be made in-place, or there is an intermediate reference that is read-only or sparse. In cases where such a

[1] Or has at least been acknowledged as being written by the physical block device.

reference exists, Parallax is careful to write all tree data below that reference to disk *before* updating the reference on disk. Thus, to satisfy consistency for each logical request, Parallax must not modify nodes in the on-disk tree until all component requests affecting lower levels of the tree have been committed to disk.

We refer to the block that contains this sparse or read-only reference as a *commit node*, as updates to it will atomically add all of the new blocks written below it to the lookup tree. In the case of a crash, some nodes may have been written to disk without their commit nodes. This is acceptable, because without being linked into a tree, they will never be accessed, and the corresponding write will have failed. The orphaned nodes can be returned to the blockstore through garbage collection.

4.1.2 Intra-request Dependencies

Logical requests that are otherwise independent can share commit nodes in the tree. During writes, this can lead to nodes upon which multiple logical requests are dependent. In the case of a shared commit node, we must respect the second invariant for both nodes independently. In practice this is a very common occurrence.

This presents a problem in scheduling the write of the shared commit node. In Figure 4.1, we provide an example of this behaviour. The illustration shows a commit node and its associated data at four monotonically increasing times. At each time, nodes and data blocks that are flushed to disk and synchronized in memory appear darker in color, and are bordered with solid lines. Those blocks that appear lighter and are bordered with dashed lines have been modified in memory but those modifications have not yet reached disk.

The illustration depicts the progress of n logical write requests, a_0 through a_n, all of which are sequential and share a commit node. For simplicity, this example will consider what is effectively a radix tree with a single radix node; the Parallax pipeline behaves analogously when a full tree is present. At time t_0, assume for the purpose of illustration that we have a node, in memory and synchronized to disk, that contains no references to data blocks. At this time we receive the n requests in a single batch, we begin processing the requests issuing the data blocks to the disk, and updating the root structure in memory. At time t_1 we have made all updates to the root block in memory, and a write of one of the data blocks

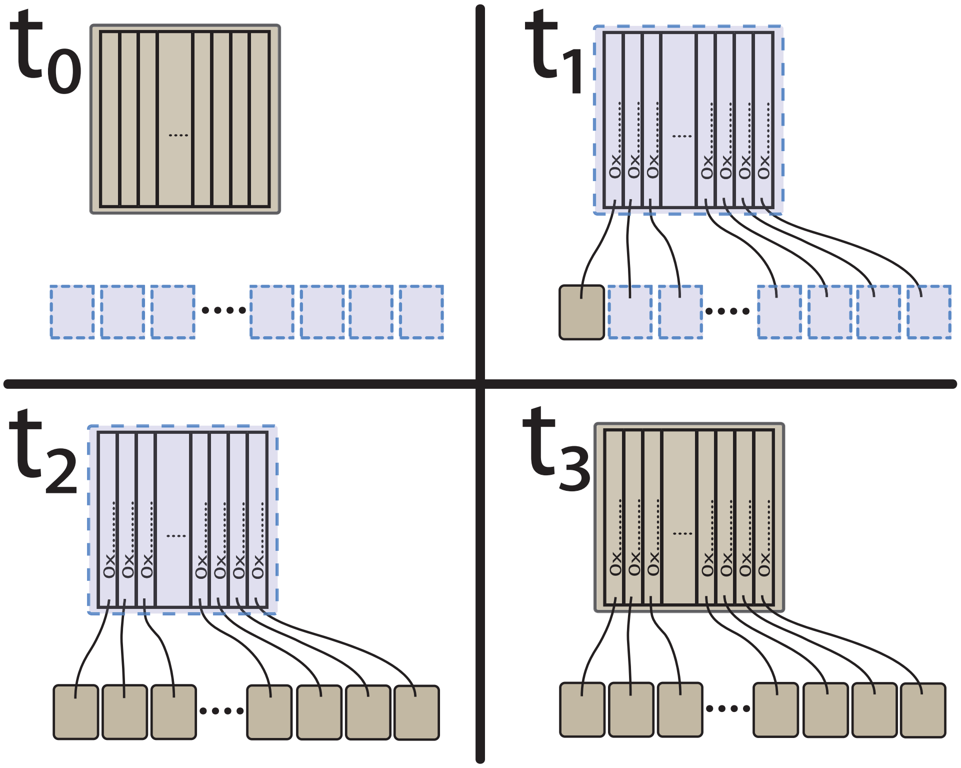

Figure 4.1: Example of a shared write dependency.

has been acknowledged by the storage system. We would like to complete the logical request a_0 as quickly as possible but we cannot flush the commit node in its given form because it still contains references to data blocks that have not been committed to disk. In this example, we wait. At time t_2, all data blocks have successfully been committed to disk; this is the earliest time that we can finally proceed to flush the commit node. Once that request completes at time t_3, we can notify the guest operating system that the associated I/O operations have each completed successfully.

The latency for completing request a_0 is thus the sum of the time required to write the data for the subsequent $n-1$ requests, plus the time required to flush the commit node. The performance impact can be further compounded by the dependency requirements imposed by a guest file system. These dependencies are only visible to Parallax in that the guest file system may stop issuing requests to Parallax due to the increased latency on some previously issued operation. [2]

4.1.3 Commit Node Inheritance

As an additional optimization, we introduce the notion of commit node inheritance. Consider a write operation that involves modifying intermediate nodes in the radix tree N_A and N_B. Suppose further that N_A is higher in the tree than N_B, and is thus a commit node for N_B. As discussed, writes to N_B can proceed immediately, as can writes to any data blocks or intermediate nodes below N_B, while writing N_A must be delayed. If at this point a second write request is made that modifies N_B but *not* N_A then, following the algorithm above, we would declare N_B a commit node on the second write operation. This would necessitate waiting to write N_B until all component requests on the second write had completed. However, in this case, we can do better. Since N_A is a commit node for N_B we know that N_B is a newly written node and not linked into the original tree. Since this is the case, we can flush the re-modified N_B immediately, safe in the knowledge that it is still protected by N_A as a commit node. In effect, the commit node for the first outstanding request is inherited onto the second. In addition to providing a performance boost, this optimization has the effect of simplifying our I/O processing by introducing the invariant that chains of dependency don't grow beyond length two: a single commit nodes, and many dependant nodes.

4.1.4 Dependency Tracking

Our API for dependency tracking consists of a single function provided by the cache. A dependency can be added between any two nodes with outstanding write requests. This is done before writes are flushed to disk, but after each node is written. Since Parallax only writes at whole block granularity we don't support more complicated tracking, as is done in [9]. While this is a simple mechanism, it has proven very powerful, and is used in establishing consistency for all of Parallax's current operations.

[2] With support for write barriers, this artifact would not be a concern.

4.1.5 Further Improvements

Commit nodes are the fundamental “dial” for trading off batching versus latency in the request pipeline. In the case of sequential writes, where all outstanding writes (of which there are a finite number) share a common commit node, it is possible that every in-flight request must complete before notifications may be passed back to the guest, resulting in bubbles while we wait for the guest to refill the request pipeline. We intend to address this by limiting the number of outstanding logical requests that are dependent on a given commit node, and forcing the node to be written once this number exceeds a threshold, likely half of the maximum in-flight requests. Issuing intermediate versions of the commit node will trade off a small number of additional writes for better interleaving of notifications to the guest. This technique was employed in [9]. As a point of comparison, we have disabled the dependency tracking between nodes, allowing them to be flushed immediately. Such an approach yields a 5% increase in sequential write performance, thought it is obviously unsafe for normal operation. With correct flushing of intermediate results we may be able to close this performance gap.

4.2 Snapshots in the Pipeline

In systems where distributed writes to shared data must be managed, a linearizability of I/O requests around snapshots must be established, otherwise there can be no consensus about the correct state of a snapshot. Establishing this linearity can be expensive and complicated. In other systems, this requires pausing the I/O stream to some degree. A simple approach is to drain the I/O queue entirely [16], while a more complicated approach is to optimistically assume success and to retry I/O that conflicts with the snapshot [1].

In Parallax, we expect that high frequency snapshots will inevitably conflict with write operations. To avoid the performance penalties associated with the previously mentioned techniques, we instead treat snapshots as pipelined requests and allow the pipeline to continue to be processed without pauses or retries. Linearization of write requests in Parallax comes naturally because each VDI is being written to by at most one physical host. Without our single-writer assumptions and our client-oriented design such an approach would not be possible.

The point at which Parallax receives a snapshot request is when the snapshot is said to occur, and the snapshot is then placed in the pipeline. However, snapshots share the same ordering semantics as any other request in the pipeline; the ordering between any requests that are in the pipeline at the same time is ambiguous. Under normal conditions, all requests received prior to a snapshot will be contained in the snapshot, while any later requests will not; however, this is not guaranteed.

By design, both requests arriving before and after the snapshot can continue to be processed in parallel with the snapshot itself. Therefore, Parallax establishes an ordering of requests with respect to the snapshot, but does not need to write blocks to disk in that order. In fact, we can allow both pre-snapshot and post-snapshot operations to complete on their respective views of the disk simultaneously, and even after the record of the snapshot itself is written to disk. Our generalized dependency tracking to provides a simple mechanism to ensure that the disk remains consistent and that writes are correctly ordered.

4.2.1 Handling Failures

As long as Parallax terminates normally, allowing outstanding requests to complete after the snapshot has been recorded does not pose a serious problem; a simple short-lived lock on the snapshot image until it becomes truly read-only suffices. [3] However, when considering the possibility that Parallax may crash, we must address several potential problems. Each has been directly validated by injected failures in a running system.

Incomplete snapshots: Creating a snapshot requires that at least three blocks are written to disk. It is possible that some of these writes reach disk just before a crash, while others do not. The resulting incomplete snapshot is detected with a simple validation check every time the system boots. Incomplete snapshots are discarded at this time. This is considerably less complicated and costly than similar methods for comparable systems [1].

Complete snapshot, but incomplete post-snapshot requests: It is possible that a snapshot is correctly placed on disk, but some requests that follow the snap-

[3] We do not measure or discuss this delay any further, because it effects only the rate at which new Parallax volumes can be created. We have no reason to believe that this is an important metric at this time.

shot are still in the pipeline when a crash occurs. This case is effectively no different from those when a snapshot does not occur. Consistency is guaranteed by the dependency tracking system.

Complete snapshot, incomplete pre-snapshot requests: It is also possible that a snapshot is correctly placed on disk, but some requests that precede the snapshot are still in the pipeline when a crash occurs. In this case, such requests are necessarily lost. Since the ordering of these requests relative to the snapshot is ambiguous and the requests have not been confirmed as completed to the guest OS, the integrity of the snapshot is unchanged. The lost requests can be safely discarded as if they had arrived after the snapshot. If the snapshot is made in conjunction with a full system checkpoint, the guest OS will re-issue these requests. Consistency of the disk is guaranteed by the dependency tracking system.

Chapter 5

Evaluation

5.1 Evaluation

We now consider Parallax's performance. As discussed in previous chapters, the design of our system includes a number of factors that we expect to impose considerable overheads on performance. Block address virtualization is provided by the Parallax daemon, which runs in user space in an isolated VM and therefore incurs context-switching on every batch of block requests. Additionally, our address mapping metadata involves 3-level radix trees, which risks a dramatic increase in the latency of disk accesses due to seeks on uncached metadata blocks.

There are two questions that this performance analysis attempts to answer. First, what are the overheads that Parallax imposes on the processing of I/O requests? Second, what are the performance implications of the virtual machine specific features that Parallax provides? We address these questions in turn, using sequential read and write [3] (in Chapter 5.1.1) and PostMark [13] (in Chapter 5.1.1) to answer the first and using a combination of micro and macro-benchmarks to address the second.

In all tests, we use IBM eServer x306 machines, each node with a 3.2 GHz Pentium-4 processor, 1 GByte of RAM, and an Intel e1000 GbE network interface. Storage is provided by a NetApp FAS3070 [1] exporting an iSCSI LUN over gigabit

[1] We chose to benchmark against the FAS 3070 because it is simply the fastest iSCSI target available to us. This is the UBC CS department filer, and so has required very late-night benchmarking

links. We access the filer in all cases using the Linux open-iSCSI software initiator (v2.0.730, and kernel module v1.1-646) running in domain 0. We have been developing against Xen 3.1.0 as a base. One notable modification that we have made to Xen has been to double the maximum number of block requests, from 32 to 64, that a guest may issue at any given time, by allocating an additional shared ring page in the split block (blkback) driver. The standard 32-slot rings were shown to be a bottleneck when connecting to iSCSI over a high capacity network.

5.1.1 Overall Performance

It is worth providing a small amount of additional detail on each of the test configurations that we compare. Our analysis compares access to the block device from Xen's domain 0 (dom0 in the graphs), to the block device directly connected to a guest VM using the block back driver (blkback), and to Parallax. Parallax virtualizes block access through blktap [38], which facilitates the development of user-mode storage drivers.

Accessing block devices from dom0 has the least overhead, in that there is no extra processing required on block requests and dom0 has direct access to the network interface. This configuration is effectively the same as unvirtualized Linux with respect to block performance. In addition, in dom0 tests, the full system RAM and both hyperthreads are available to dom0. In the following cases, the memory and hyperthreads are equally divided between dom0 (which acts as the Storage VM[2]) and a guest VM.

In the "Direct" case, we access the block device from a guest VM over Xen's blkback driver. In this case, the guest runs a block driver that forwards requests over a shared memory ring to a driver (blkback) in dom0, where they are issued to the iSCSI stack. Dom0 receives direct access to the relevant guest pages, so there is no copy overhead, but this case does incur a world switch between the client VM and dom0 for each batch of requests.

Finally, in the case of Parallax, the configuration is similar to the direct case, but

efforts. The FAS provides a considerable amount of NVRAM on the write path, which explains the asymmetric performance between read and write in many of our benchmark results.

[2] We intend to explore a completely isolated Storage VM configuration as part of future work on live storage system upgrades.

when requests arrive at the dom0 kernel module (blktap instead of blkback), they are passed on to the Parallax daemon running in user space. Parallax issues reads and writes to the Linux kernel using Linux's asynchronous I/O interface (libaio), which are then issued to the iSCSI stack.

Reported performance measures a best of 3 runs for each category. The alternate convention of averaging several runs results in slightly lower performance for dom0 and direct configurations relative to Parallax. Memory and CPU overheads were shown to be too small to warrant their inclusion here.

Sequential I/O

For each of the three possible configurations, we ran Bonnie++ twice in succession. The first run provided cold-cache data points, while the second allows Parallax to populate its radix node cache[3]. The strong write performance in the warm cache case demonstrates that Parallax is able to maintain write performance near the effective line speed of a 1Gbps connection. Our system performance is within 5% of dom0. At the same time, the 12% performance degradation in the cold cache case underscores the importance of caching in Parallax, as doing so limits the overheads involved in radix tree traversal. As we have focused our efforts to date on tuning the write path, we have not yet sought aggressive optimizations for read operations. This is apparent in the Bonnie++ test, as we can see read performance slipping to more than 14% lower than that of our non-virtualized dom0 configuration.

PostMark

Figure 5.2 shows the results of running PostMark on the Parallax and directly attached configurations. PostMark is designed to model a heavy load placed on many small files [13]. The performance of Parallax is comparable to and slightly lower than that of the directly connected configuration. In all cases we fall within 10% of a directly attached block device. File creation and deletion are performed during and after the transaction phase of the PostMark test, respectively. We have merged both phases, and illustrated the relative time spent in each.

[3] In the read path, this may also have some effect on our filer's caching; however, considering the small increase in read throughput and the fact that a sequential read is easily predictable, we conclude that these effects are minimal.

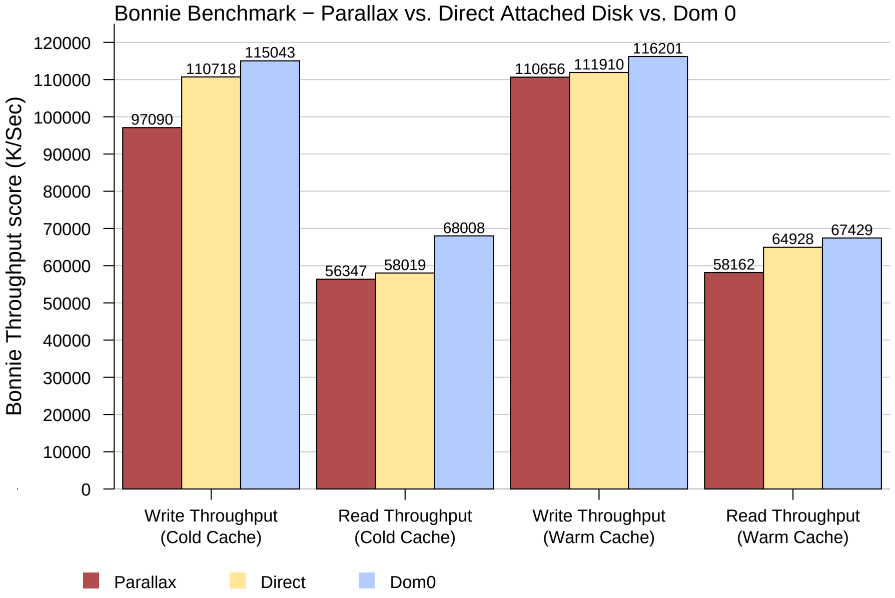

Figure 5.1: System throughput as reported by Bonnie++ during a first (cold) and second (warm) run.

Local Disk Performance

To demonstrate that a high-end storage array with NVRAM is not required to maintain Parallax's performance profile, we ran the same tests using a commodity disk as a target. Our disk was a Hitachi Deskstar 7K80, which is an 80GB, 7,200 RPM SATA drive with an 8MB cache. The results of Bonnie++ are shown in Figure 5.3. Again, the importance of caching intermediate radix nodes is clear. Once the system has been in use for a short time, the write overheads drop to 13%, while read overheads are shown to be less than 6%. In this case, Parallax's somewhat higher I/O requirements increase the degree to which the local disk acts as a bottleneck. The lack of tuning of read operations is not apparent at this lower throughput.

In Figure 5.4 we show the results of running the PostMark test with a local disk, as above. Similarly, the results show a only small performance penalty when Parallax is used without the advantages of striping disks or a large write cache.

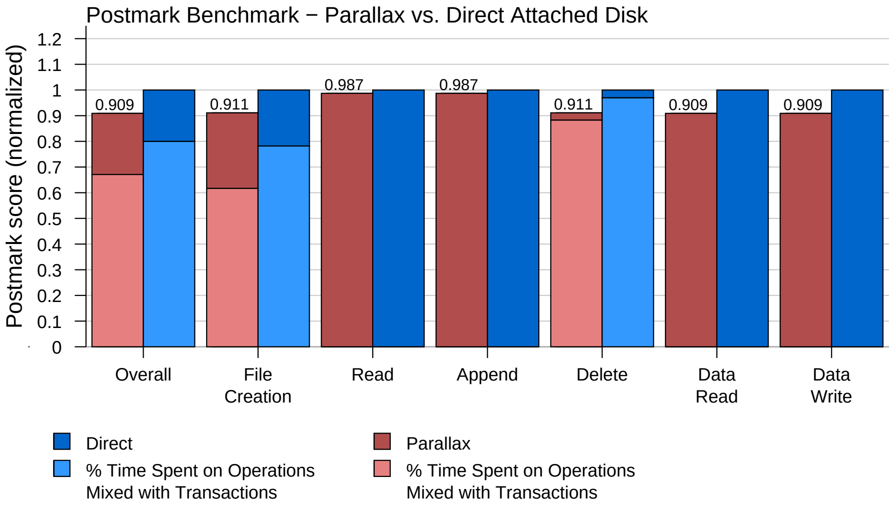

Figure 5.2: PostMark results running against network available filer (normalized).

5.1.2 Measuring Parallax's Features

Disk Fragmentation

While our approach to storage provides many beneficial features, it raises concerns over how performance will evolve as a blockstore ages. This is not unique to Parallax, rather it is the natural argument against any copy-on-write system — that block fragmentation will eventually prove detrimental to performance.

In Parallax, fragmentation occurs when the block addresses visible to the guest VM are sequentially placed, but the corresponding physical addresses are not. This can come as a result of several usage scenarios. First, when a snapshot is deleted, it can fragment the allocation bitmaps forcing future sequential writes to be placed non-linearly. Second, if a virtual disk is sparse, future writes may be placed far from other blocks that are adjacent in the block address space. Similarly, when snapshots are used, the CoW behavior can force written blocks to diverging locations on the physical medium. Third, the interleaving of writes to

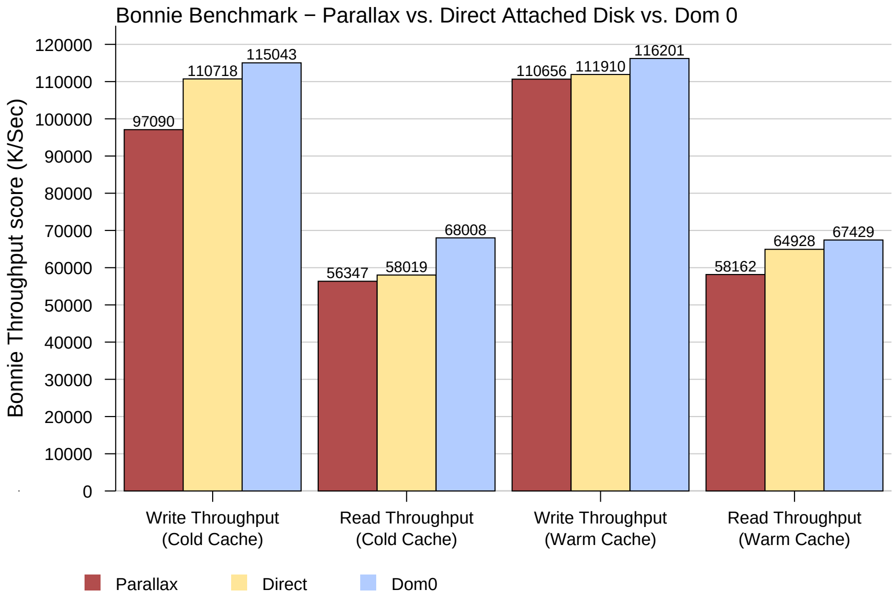

Figure 5.3: System throughput against a local disk as reported by Bonnie++ during a first (cold) and second (warm) run.

multiple VDIs will result in data for each virtual disk being placed together on the physical medium. Finally, VM migration will cause the associated Parallax virtual disks to be moved to new physical hosts, which will in turn allocate from different extents. Thus data allocations after migration will not be located near those that occurred before migration. Note however that fragmentation will not result from writing data to blocks that are not marked read-only, as this operation will be done in place. In addition, sequential writes that target a read-only or sparse region of a virtual disk will remain sequential when they are written to newly allocated regions. This is true even if the original write-protected blocks were not linear on disk, due to fragmentation.

Thus, as VDIs are created, deleted, and snapshotted, we intuitively expect that some fragmentation of the physical media will occur, potentially incurring seeks even when performing sequential accesses to the virtual disk. To explore this pos-

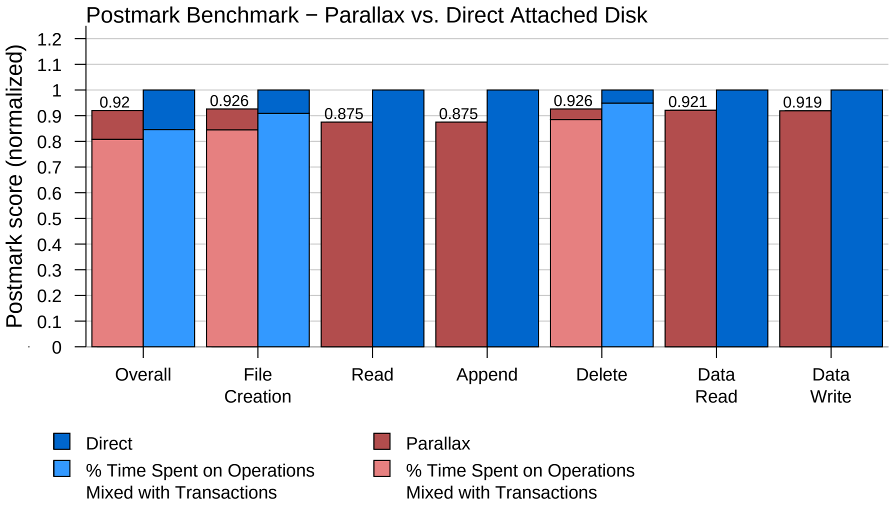

Figure 5.4: PostMark results running against a local disk (normalized).

sibility further, we modified our allocator to place new blocks randomly in the extent, simulating a worst-case allocation of data. We then benchmarked local disk and filer read performance against the resulting VDI, as shown in Figure 5.5.

Even though this test is contrived to place extreme stress on disk performance, the figure presents three interesting results. First, although it would be difficult to generate such a degenerate disk in the normal use of Parallax, in this worst case scenario, random block placement does incur a considerable performance penalty, especially on a commodity disk. In addition, the test confirms that the overheads for Bonnie++, which emphasizes sequential disk access, are higher than those for PostMark, which emphasizes smaller reads from a wider range of the disk. Interestingly, the third result is that when the workload is repeated, the filer is capable of regaining most of the lost performance, and even outperforms PostMark with sequential allocation. Although a conclusive analysis is complicated by the encapsulated nature of the filer, this result demonstrates that the increased reliance on disk striping, virtualized block addressing, and intelligent caching makes the fragmentation problem both difficult to characterize and compelling. It punctuates the

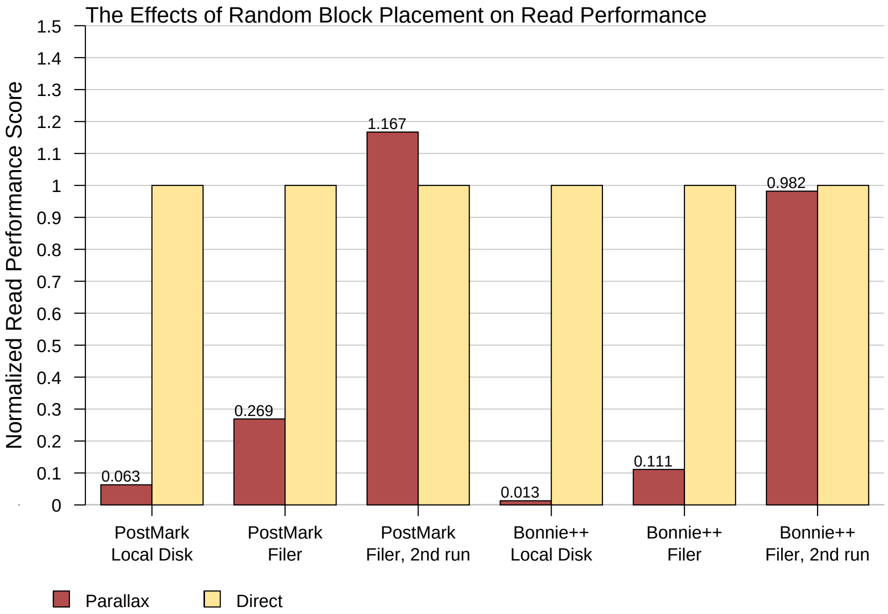

Figure 5.5: The effects of a worst case block allocation scheme on Parallax performance.

observation made by Stein et al [32], that storage stacks have become incredibly complex and that naive block placement does not necessarily translate to worse case performance — indeed it can prove beneficial.

As a block management system, Parallax is well positioned to tackle the fragmentation problem directly. We are currently enhancing the garbage collector to allow arbitrary block remapping. This facility will be used to defragment VDIs and data extents, and to allow the remapping of performance-sensitive regions of disk into large contiguous regions that may be directly referenced at higher levels in the metadata tree, much like the concept of superpages in virtual memory. These remapping operations are independent of the data path, similar in design to the garbage collector discussed in Chapter 3.4.3. Ultimately, detailed analysis of these features, combined with a better characterization of realistic workloads, will be necessary to evaluate this aspect of Parallax's performance.

Radix tree overheads

In order to provide insight into the servicing of individual block requests, we use a microbenchmark to measure the overheads. There are three distinct node types in a radix tree. A node may be writable, allowing in-place modification. It may be sparse, in that it is non-physical and zero-filled. Finally, it may be read-only, requiring that the contents be copied in order to process write requests. We instrumented Parallax to generate each of these types of nodes at the top level of the tree, to highlight their differences. When non-writable nodes are reached at lower levels in the tree, the performance impact will be smaller. Figure 5.6 shows the results. Unsurprisingly, when a single block is written, Parallax performs very similarly to the other configurations, because writing is done in place. When a sparse node is reached at the top of the radix tree, Parallax must perform writes on intermediate radix nodes, the radix root, and the actual data. Of these writes, the radix root can only complete after all other requests have finished, as was discussed in Chapter 4.1. The faulted case is similar in that it too requires a serialized write, but it also carries additional overheads in reading and copying intermediate tree nodes.

Garbage collection

As described in Chapter 3.4.3, the Parallax garbage collector works via sequential scans of all metadata extents. As a result, 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. We've run the garbage collector on full blockstores ranging in size from 10GB to 50GB, and we characterize its performance by the amount of data it can process (measured as the size of the blockstore) per unit time. Its performance is linear at a rate of 0.96GB/sec. This exceeds the line speed of the storage array because leaf nodes do not need to be read to determine if they can be collected.

The key to the good performance of the garbage collector is that the Reachability Map is stored in memory. In contrast to the Block Allocation Maps of each extent which are always scanned sequentially, the RMap is accessed in random order. This puts a constraint on the algorithm's scalability. Since the RMap contains

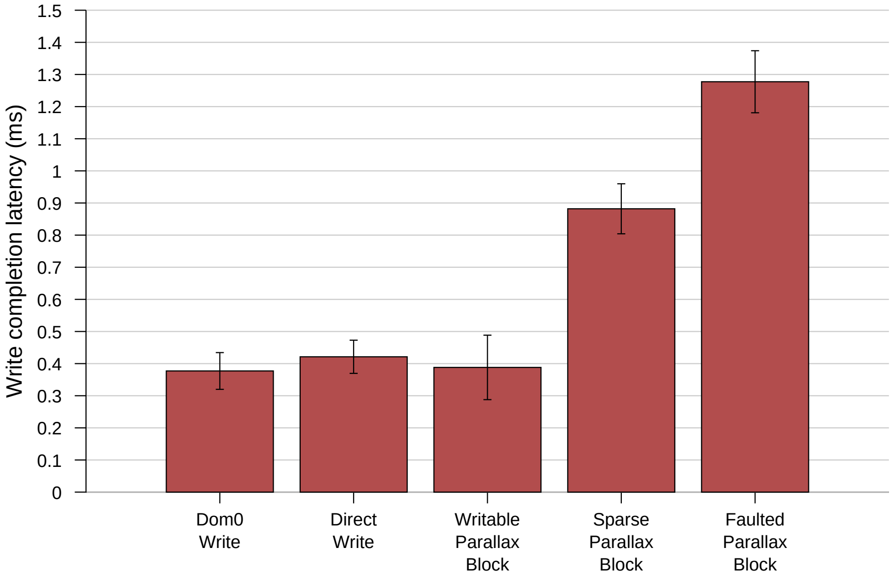

Figure 5.6: Single write request latency for dom0, direct attached disks, and three potential Parallax states. A 95% confidence interval is shown.

one bit per blockstore block, each 1GB of memory in the garbage collector allows it to manage 32TB of storage. To move beyond those constraints, RMap pages can be flushed to disk. We look forward to having to address this challenge in the future, should we be confronted with a sufficiently large Parallax installation.

Snapshots

To establish baseline performance, we first measured the general performance of checkpointing the storage of a running but idle VM. We completed 500 checkpoints in a tight loop with no delay. A histogram of the time required by each checkpoint is given in Figure 5.7. The maximum observed snapshot latency in this test was 3.25ms. This is because the 3 writes required for most snapshots can be issued with a high degree of concurrency and are often serviced by the physical disk's write cache. In this test, more than 90% of snapshots completed within a single millisecond; however, it is difficult to establish a strong bound on snapshot

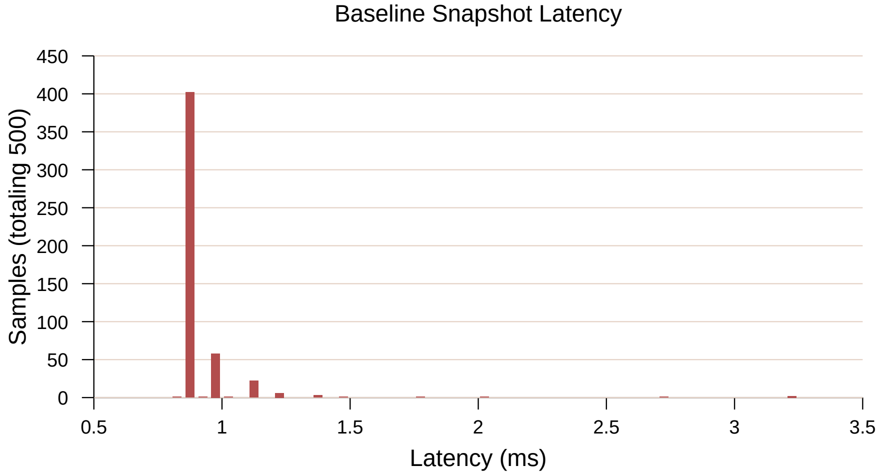

Figure 5.7: Snapshot latency of running VM during constant checkpointing.

latency. The rate at which snapshots may be taken depends on the performance of the underlying storage and the load on Parallax's I/O request pipeline. If the I/O pipeline is full, the snapshot request may be delayed as Parallax services other requests. Average snapshot latency is generally under 10ms, but under very heavy load we have observed average snapshot latency to be as high as 30ms.

Next we measured the effects of varying snapshot rates during the decompression and build of a Linux 2.6 kernel. In Figure 5.8 we provide results for various sub-second snapshot intervals. While this frequency may seem extreme, it explores a reasonable space for applications that require near-continuous state capture. Larger snapshot intervals were tested as well, but had little effect on performance. The snapshot interval is measured as the average time between successive snapshots and includes the actual time required to complete the snapshot. By increasing the snapshot rate from 1 per second to 100 per second we incur only a 4% performance overhead. Furthermore, the majority of this increase occurs as we move from a 20ms to 10ms interval.

Figure 5.9 depicts the results of the same test in terms of data and metadata creation. The data consumption is largely fixed over all tests because kernel com-

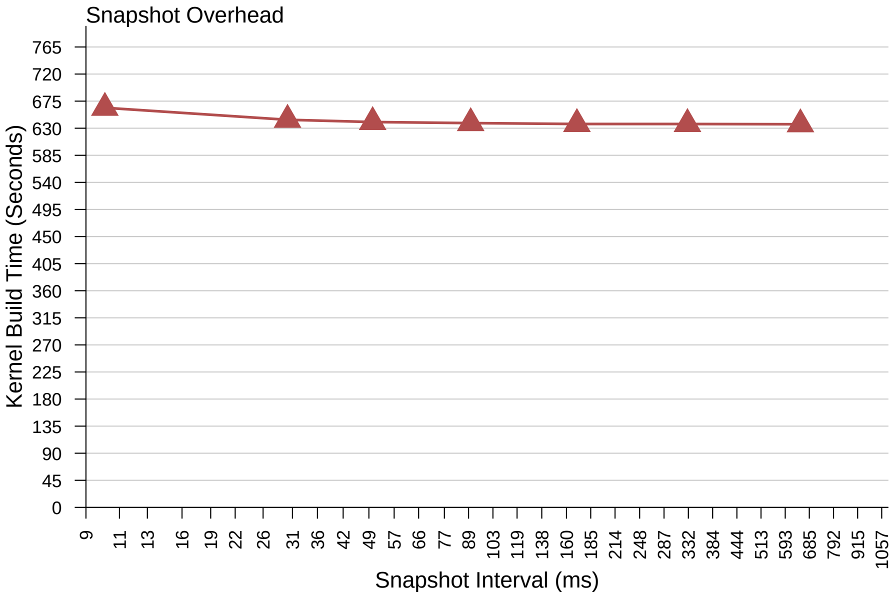

Figure 5.8: Measuring performance effects of various snapshot intervals on a Linux Kernel decompression and compilation.

pilation does not involve overwriting previously written data, thus the snapshots have little effect on the number of data blocks created. In the extreme, taking snapshots every 10ms, 65,852 snapshots were created, each consuming just 5.84KB of storage on average. This accounted for 375 MB of metadata, roughly equal in size to the 396 MB of data that was written.

To further explore the potential of snapshots, we created two alternate modes to investigate even more fine-grained state capture in Parallax. In the first case we perform a snapshot after every write request, and in the second we inject a snapshot after each batch of requests. Owing to the experimental nature of this code, our implementation is unoptimized. In particular, we were forced to delay the I/O stream to avoid issuing multiple snapshots in parallel. This restriction has since been removed.

The effects on performance can be categorized as follows. First, there is an in-

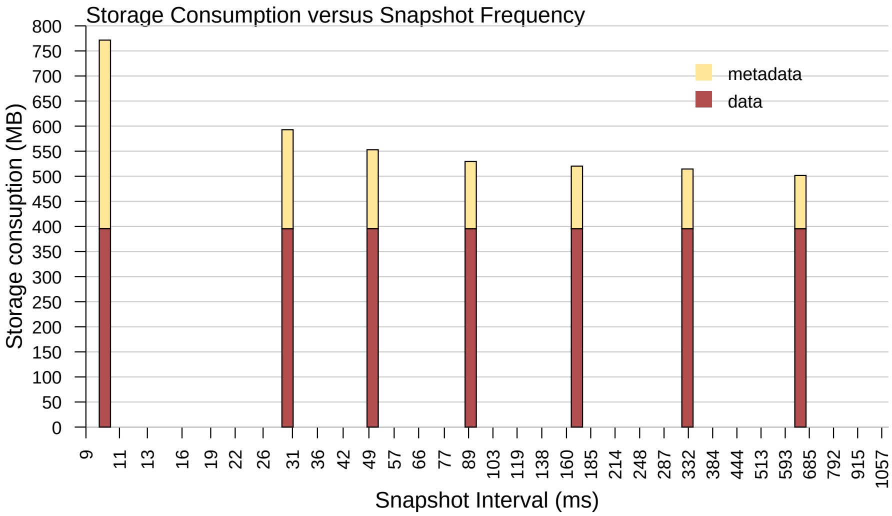

Figure 5.9: Measuring data consumption at various snapshot intervals on a Linux Kernel decompression and compilation.

crease in the amount of I/O being performed. Capturing every block modification multiplies the amount of I/O required by a factor of 3, while capturing each batch can yield an increase as low as 5%. Second, these snapshot modes increase the number of faulted writes, as a simple function of the number of snapshots taken. Finally, the delays associated with correct dependency tracking and the pipeline stalls we introduced add to overhead. It is our belief that the bulk of the performance decrease is actually in this third category.

Snapshot per Write	877.921 seconds	1188.59 MB
Snapshot per Batch	764.117 seconds	790.46 MB

Table 5.1: Alternate snapshot configurations.

The impact on the performance of the kernel compile is shown in Table 5.1. Metadata values are very much as we would expect, showing three metadata blocks for each data block in the snapshot per write case, and two in the snapshot per batch

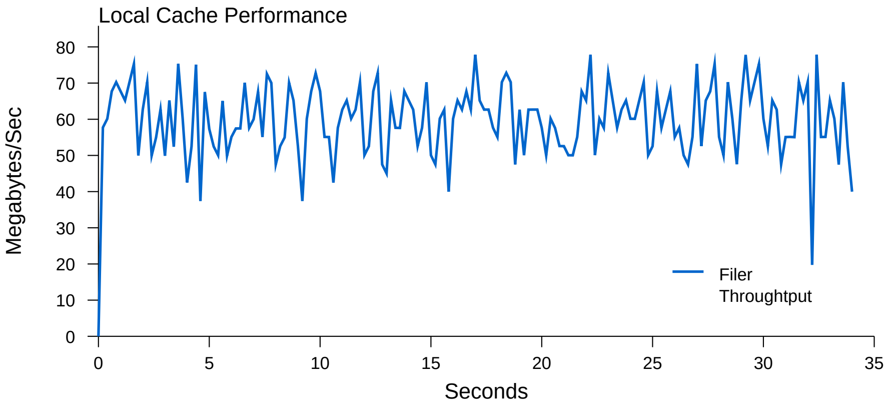

Figure 5.10: Performance of bursted write traffic.

case. The performance results increase by 16% to 33%, and we feel that there is considerable room for improvement, as discussed. We conclude from these results that extending Parallax into the area of continuous data protection represents a promising new direction.

Local Disk Cache

We evaluated our local disk cache to illustrate the advantage of shaping the traffic of storage clients accessing a centralized network storage device. We have not yet fully explored the performance of caching to local disk in all scenarios, as its implementation is still in an early phase. The following experiment is not meant to exhaustively explore the implications of this technique, merely to illustrate its use and current implementation. In addition, the local disk cache demonstrates the ease with which new features may be added to Parallax, owing to its clean isolation from both the physical storage system and the guest operating system. The local disk cache is currently implemented in less than 500 lines of code.

In Figure 5.10, we show the time required to process 500MB of write traffic by 4 clients simultaneously. This temporary saturation of the shared storage resource may come as a result of an unusual and temporary increase in load, such as occurs

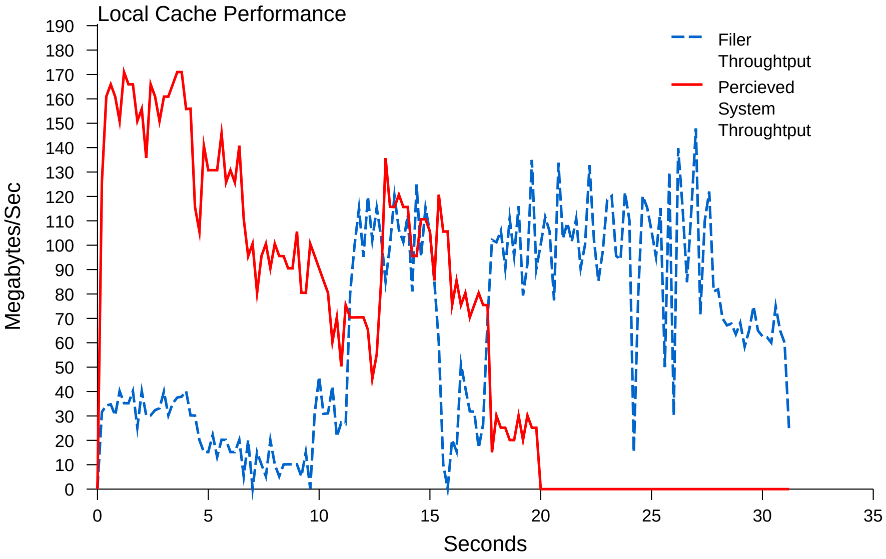

Figure 5.11: Performance of bursted write traffic with local disk caching.

when a system is initially brought online. This scenario results in a degradation of per-client performance, even as the overall throughput is high.

In Figure 5.11 we perform the same test with the help of our local disk cache. The Storage VMs each quickly recognize increased latency in their I/O requests to the filer and enable their local caches. As a result, clients perceive an aggregate increase in throughput, because each local disk can be accessed without interference from competing clients. In the background, writes that had been made to the local cache are flushed to network storage without putting too much strain on the shared resource. Clients process the workload in significantly less time (18-20 seconds). A short time after the job completes, the cache is fully drained, though this background process is transparent to users.

Metadata consumption

While there are some large metadata overheads, particularly in the initial extent, we expect that metadata consumption in Parallax will be dominated by the storage of radix nodes. Measuring this consumption is difficult, because it is parameterized by not only the image size, but also the sparseness of the images, the system-wide frequency and quality of snapshots, and the degree of sharing involved. To simplify this problem, we consider only the rate of radix nodes per data block on an idealized system.

In a full tree of height three with no sparseness we must create a radix node for every 512 blocks of data, an additional node for every 262,144 blocks of data, and finally a root block for the whole disk. With a standard 4KB block size, for 512GB of data, we must store just over 1GB of data in the form of radix nodes. Naturally for a non-full radix tree, this ratio could be larger. However, we believe that in a large system, the predominant concern is the waste created by duplication of highly redundant system images — a problem we explicitly address.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Parallax is a system that attempts to provide storage virtualization specifically for virtual machines. The system moves functionality, such as volume snapshots, that is commonly implemented on expensive storage hardware out into a software implementation running within a VM on the physical host that consumes the storage. This approach is a novel organization for a storage system, and allows a storage administrator access to a cluster-wide administration domain for storage.

Despite its use of several potentially high-overhead techniques, such as a user-level implementation and fine-grained block mappings through 3-level radix trees, Parallax achieves good performance against both a very fast shared storage target and a commodity local disk. We attribute our performance to our design which avoids locking on the data path, and makes use of caching, batching, and an efficient pipeline in order to mask increases in per-request latency. Our system also boasts an impressive snapshot capability, which can be used for high frequency state capture approaching that of continuous data protection.

6.2 Future Work

We are actively exploring a number of improvements to the system including the establishing of a dedicated storage VM, the use of block remapping to recreate the sharing of common data as VDIs diverge, the creation of superpage-style mappings to avoid the overhead of tree traversals for large contiguous extents, and exposing Parallax's snapshot and dependency tracking features as primitives to the guest file system. We also plan to investigate the fragmentation problem in great detail and explore the degree to which it can be solved by actively re-linearizing the blocks of a running system. As an alternative to using a single network available disk, we are designing a mode of operation in which Parallax itself will manage multiple physical volumes. This may prove a lower cost alternative to large sophisticated arrays.

Work is currently in progress to export the blktap interface to Linux block devices. While virtual machine based operation is still the motivation for Parallax, such a development will enable still wider use of the system.

We are also continually making performance improvements to Parallax. As part of these efforts we are also testing Parallax on a wider array of hardware. We plan to deploy Parallax as part of an experimental VM-based hosting environment later this year. This will enable us to refine our designs and collect more realistic data on Parallax's performance. An open-source release of Parallax, with current performance data, is available at: http://dsg.cs.ubc.ca/parallax/.

Bibliography

[1] M. K. Aguilera, S. Spence, and A. Veitch. Olive: distributed point-in-time branching storage for real systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design & Implementation (NSDI 2006)*, pages 367–380, Berkeley, CA, USA, May 2006.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 2005)*, May 2005.

[3] R. Coker. Bonnie++. http://www.coker.com.au/bonnie++.

[4] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association. ISBN 111-999-5555-22-1.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design & Implementation (OSDI 2002)*, December 2002. URL http://www.usenix.org/events/osdi02/tech/dunlap.html.

[6] E. Eide, L. Stoller, and J. Lepreau. An experimentation workbench for replayable networking research. In *Proceedings of the Fourth USENIX Symposium on Networked Systems Design & Implementation*, April 2007.

[7] 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)*, October 2004.

[8] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. C. Veitch. Fab: Enterprise storage systems on a shoestring. In *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems, Lihue (Kauai), Hawaii, USA*, pages 169–174, May 2003.

[9] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, pages 307–320, October 2007.

[10] 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, January 1994.

[11] IBM. Help protect your most valuable data and application assets continuously. ftp://ftp.software.ibm.com/software/tivoli/whitepapers/solution-cdptsm.pdf.

[12] M. Ji. Instant snapshots in a federated array of bricks., January 2005.

[13] J. Katcher. Postmark: a new file system benchmark, 1997.

[14] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005*, pages 1–15, Berkeley, CA, April 2005.

[15] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications, Calicoon, NY*, pages 40–46, June 2002.

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

[17] 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, December 2004. URL http://www.usenix.org/events/osdi04/tech/levasseur.html.

[18] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *FREENIX Track: 1999 USENIX Annual TC*, pages 1–18, Monterey, CA, June 1999.

[19] M. McLoughlin. The QCOW image format. http://www.gnome.org/˜markmc/qcow-image-format.html.

[20] 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 *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 41–54, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-013-5. doi:http://doi.acm.org/10.1145/1352592.1352598.

[21] Microsoft. Protecting virtualized environments with system center data protection manager 2007. http://www.microsoft.com/systemcenter/dataprotectionmanager/en/us/white-papers.aspx.

[22] Microsoft TechNet. Virtual hard disk image format specification. http://microsoft.com/technet/virtualserver/ downloads/vhdspec.mspx.

[23] Z. Peterson and R. Burns. Ext3cow: a time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005. ISSN 1553-3077. doi:http://doi.acm.org/10.1145/1063786.1063789.

[24] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design & Implementation (NSDI 2006)*, pages 353–366, Berkeley, CA, USA, May 2006.

[25] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from bell labs. In *In Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990.

[26] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX Conference on File and Storage Technologies*, Monterey,CA, 2002.

[27] Red Hat, Inc. LVM architectural overview. http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual /Cluster_Logical_Volume_Manager/LVM_definition.html.

[28] O. Rodeh and A. Teperman. zFS – A scalable distributed file system using object disks. In *MSS '03: Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 207–218, Washington, DC, USA, April 2003. ISBN 0-7695-1914-8.

[29] D. S. Santry, M. J. Feeley, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. pages 110–123, 1999.

[30] C. Sapuntzakis and M. Lam. Virtual appliances in the collective: A road to hassle-free computing. In *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems*, pages 55–60, May 2003.

[31] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design & Implementation (OSDI 2002)*, December 2002. URL http://www.usenix.org/events/osdi02/tech/sapuntzakis.html.

[32] L. Stein. Stupid file systems are better. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 5–5, Berkeley, CA, USA, 2005.

[33] VMWare, Inc. Virtual machine disk format. http://www.vmware.com/interfaces/vmdk.html.

[34] VMware, Inc. VMware VMFS product datasheet. http://www.vmware.com/pdf/vmfs_datasheet.pdf, .

[35] VMware, Inc. Performance Tuning Best Practices for ESX Server 3. http://www.vmware.com/pdf/vi_performance_tuning.pdf, .

[36] VMWare, Inc. Using vmware esx server system and vmware virtual infrastructure for backup, restoration, and disaster recovery. www.vmware.com/pdf/esx_backup_wp.pdf.

[37] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, fidelity and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 148–162, Brighton, UK, October 2005.

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

[39] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI 2004)*, pages 77–90, December 2004.