Extensible Networked-Storage Virtualization with Metadata Management at the Block Level

by

Michail D. Flouris

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Extensible Networked-Storage Virtualization
with Metadata Management at the Block Level

Michail D. Flouris
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2009

Increased scaling costs and lack of desired features is leading to the evolution of high-performance storage systems from centralized architectures and specialized hardware to decentralized, commodity storage clusters. Existing systems try to address storage cost and management issues at the filesystem level. Besides dictating the use of a specific filesystem, however, this approach leads to increased complexity and load imbalance towards the file-server side, which in turn increase costs to scale.

In this thesis, we examine these problems at the block-level. This approach has several advantages, such as transparency, cost-efficiency, better resource utilization, simplicity and easier management.

First of all, we explore the mechanisms, the merits, and the overheads associated with advanced metadata-intensive functionality at the block level, by providing versioning at the block level. We find that block-level versioning has low overhead and offers transparency and simplicity advantages over filesystem-based approaches.

Secondly, we study the problem of providing extensibility required by diverse and changing application needs that may use a single storage system. We provide support for (i) adding desired functions as block-level extensions, and (ii) flexibly combining them to create modular I/O hierarchies. In this direction, we design, implement and evaluate an extensible *block-level storage virtualization framework*, Violin, with support for metadata-intensive functions. Extending Violin we build *Orchestra*, an extensible framework for cluster storage virtualization and scalable storage sharing at the block-level. We show that Orchestra's enhanced block interface can substantially simplify the design of higher-level storage services, such as cluster filesystems, while being scalable.

Finally, we consider the problem of consistency and availability in decentralized commodity clus-

ters. We propose *RIBD*, a novel storage system that provides support for handling both data and metadata consistency issues at the block layer. RIBD uses the notion of *consistency intervals (CIs)* to provide fine-grain consistency semantics on sequences of block level operations by means of a lightweight transactional mechanism. RIBD relies on Orchestra's virtualization mechanisms and uses a roll-back recovery mechanism based on low-overhead block-level versioning. We evaluate RIBD on a cluster of 24 nodes, and find that it performs comparably to two popular cluster filesystems, PVFS and GFS, while offering stronger consistency guarantees.

To my wife, Stavroti, for her support and patience to see the end of it.

To my children, Kleria and Dimitris, for making my days (and nights) worth it.

Acknowledgements

I have been extremely fortunate to have Professor Angelos Bilas as my supervisor. His constant support, insightful comments and patient guidance have aided me greatly throughout my research endeavors. I am deeply grateful to him for all his support and all I have learned under his supervision.

I would also like to thank the members of my committee, Professors Angela Demke Brown, H.-Arno Jacobsen, Cristiana Amza and my external appraiser Remzi Arpaci-Dusseau, for their thoughtful comments that significantly improved this thesis.

Special thanks go to all colleagues and paper co-authors, whose help and insight has been invaluable to my research efforts, more specifically Renaud Lachaize, Manolis Marazakis, Kostas Magoutis, Jesus Luna, Maciej Brzezniak, Zsolt Németh, Stergios Anastasiadis, Evangelos Markatos, Dionisios Pnevmatikatos, Sotiris Ioannidis, Dimitris Xinidis, Rosalia Christodoulopoulou, Reza Azimi, and Periklis Papakonstantinou.

A great thanks to all the people of the legendary Grspam list and all my friends in Toronto for making my life there warmer and more fun. I am especially grateful to Yannis Velegrakis, Tasos Kementsientsidis, Stavros Vassos, Giorgos Giakkoupis and Vasso Bartzoka for their hospitality.

I would like to thank also all the present and past members of the CARV lab and FORTH-ICS for all the conversations, the fun and for creating an enjoyable and stimulating research environment. Many special thanks go to Stavros Passas, Michalis Ligerakis, Sven Karlsson, Markos Foundoulakis and Yannis Klonatos for their help with the hardware, the cluster administration, and debugging.

A big “thank you” goes to Linda Chow for her help with the administrative tasks.

Thanks to all the friends in Greece for the moral support.

Finally, I would like to thank the people that really made all this possible; my wife Stavroti Liodaki, my children Kleria and Dimitris, my parents Maria and Dimitris, sister Irini, brother Andreas, and my family in-law. Thank you all for everything you have done for me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The beginning of knowledge is the discovery
of something we do not understand.
—Frank Herbert (1920 - 1986)

1.1 Motivation

Data are the lifeblood of computing and an invaluable asset of any organization, which makes data storage systems a critical component of current computing infrastructures. The significance and the value of this massive, ever-increasing volume of stored data [IDC, 2008; Lyman and Varian, 2003], imposes the need for improving the overall quality of data storage systems [Wilkes, 2001; Veitch et al., 2001].

Increasing requirements for improving storage efficiency and cost-effectiveness introduce the need for scalable storage systems in a data-center, that consolidate system resources into a single system image. Consolidation enables easier storage management, continuous system operation, uninterrupted from device faults, and ultimately lower cost of purchase and maintenance. The need for consolidation has largely led, over the past decade, to the adoption of storage area networks (SANs) as a popular solution to improve storage efficiency and manageability [Barker and Massiglia, 2002]. SANs are high-performance interconnects, enabling many application servers to share a set of disk arrays, as shown in Figure 1.1(left), and are usually considered synonymous to either Fibre Channel [Jurgens, 1995; Fibre Channel] or iSCSI [iSCSI] (Ethernet) networks.

However, there are two main issues with SAN-based storage architectures today [Anderson, 1995;

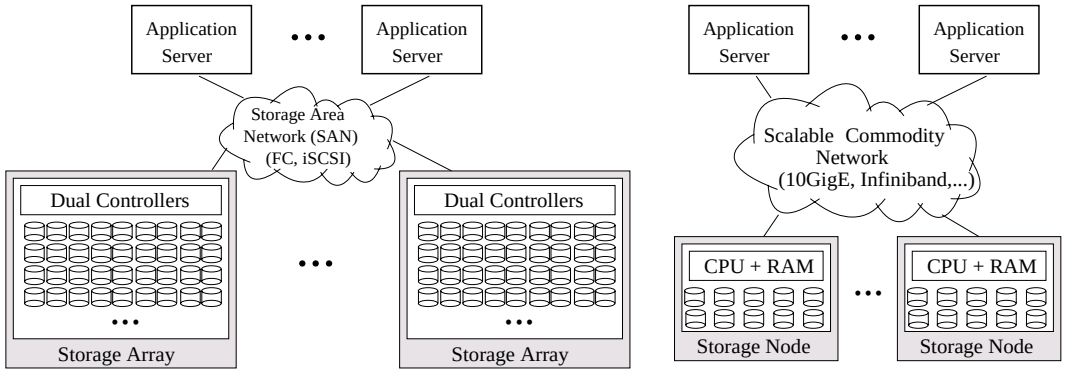


Figure 1.1: High-end SAN-based vs. cluster-based storage architectures.

Gartner Group, 2000; SNIA]: (i) they are not cost-effective to scale either in capacity or performance, and (ii) they lack desired functions and advanced features, such as data compression, deduplication or encryption. The reasons for cost-ineffectiveness are:

1. *The centralized architecture of the storage arrays.* Storage arrays are based on one or more controller backplanes (two is typical for highly-available configurations), which connect up to a few hundred disks, to allow scaling in capacity. In order to provide the highest level of performance, the array controllers are made from high-end customized hardware with high-speed buses and memory, and run specialized software. As a result they are expensive. However, no controller today is able to harness the full bandwidth of the maximum number of supported disks, if they were operating at full speed (e.g. doing sequential reads). The operation of the storage arrays depends on the fact that in reality, most I/O workloads have a high-degree of randomness which lead to disk seeks and low disk throughput.

   Scaling in capacity to more than a few hundred disks is not feasible simply by adding more storage arrays and controllers, because storage volumes cannot span storage arrays. In this case, capacity scaling requires software running on the application server side, either a distributed filesystem, a software RAID or a volume manager, which aggregates storage volumes across arrays.

2. *The cost of scaling custom interconnects used in high-end systems (i.e. Fibre Channel).* Cost per switch port rises super-linearly in large SAN switches, because of the specialized nature of the interconnect, and the small number of available vendors.

Due to the lack of cost-efficient scalability, high-performance storage systems are evolving from

centralized architectures and specialized hardware and interconnects, towards commodity-based scalable "decentralized storage clusters" (often referred to also as "storage brick architectures" or "storage grids") [HYDRAstor, 2008; Hoffman, 2007]. These systems, shown in Figure 1.1(right), are based on clusters of many commodity storage nodes with CPU, memory and I/O resources connected via a scalable, high-throughput, low-latency network, without a central controller [Gray, 2002; IBM Corp., 2008a; Saito et al., 2004].

The new storage architecture has strong potential for scaling both storage capacity and performance at reduced cost. Current commodity computers and networks have significant computing power and network performance, at a fraction of the cost of custom storage-specific hardware (controllers or networks) [Gibson et al., 1998]. Using such components, one can build low-cost storage clusters with enough capacity and performance to meet practically any storage needs, and scale it economically. This concept has been proposed by several researchers [Gray, 2002; Gibson et al., 1998; Anderson et al., 1996] and the necessary commodity technologies are maturing now with the emergence of new standards, protocols and interconnects, such as Serial ATA [SATA], 10-Gbit Ethernet, Infiniband [IBTA], Myrinet [Myrinet], PCI-X/Express/AS [Zirin and Joe Bennett], ATA over Ethernet (AoE) [Hopkins and Coile] and iSCSI [ISCSI].

On the other hand, this new architecture poses new challenges, due to the lack of central controller(s). The elimination of all centralization points and increase of I/O request asynchrony is desirable for performance and scalability purposes; however, it introduces significant challenges that do not exist in traditional centralized architectures. These challenges impose a new approach and protocols towards offering availability and strong consistency guarantees for data and metadata [Saito et al., 2004].

The second drawback of traditional, centralized storage systems is *the lack of desired functions and advanced features* [SNIA], such as deduplication, compression, encryption or tiered-storage support. The main reason for this, is that it is difficult and expensive to add advanced features to the array controllers or the SAN switches, and few storage vendors currently support them.

The ability to add new functions (*"extensibility"*), and to allow control of which available functions to use on each volume (*"flexibility"*), in order to tailor storage to application-specific requirements, is an important concern, because distinct application domains have diverse storage requirements [Leavitt, 2008; Bigelow, 2007]. Systems designed for the needs of scientific computations, data mining, e-mail serving, e-commerce, search engines, operating system (OS) image serving or data archival impose different trade-offs in terms of dimensions such as speed, reliability, capacity, high-availability, security,

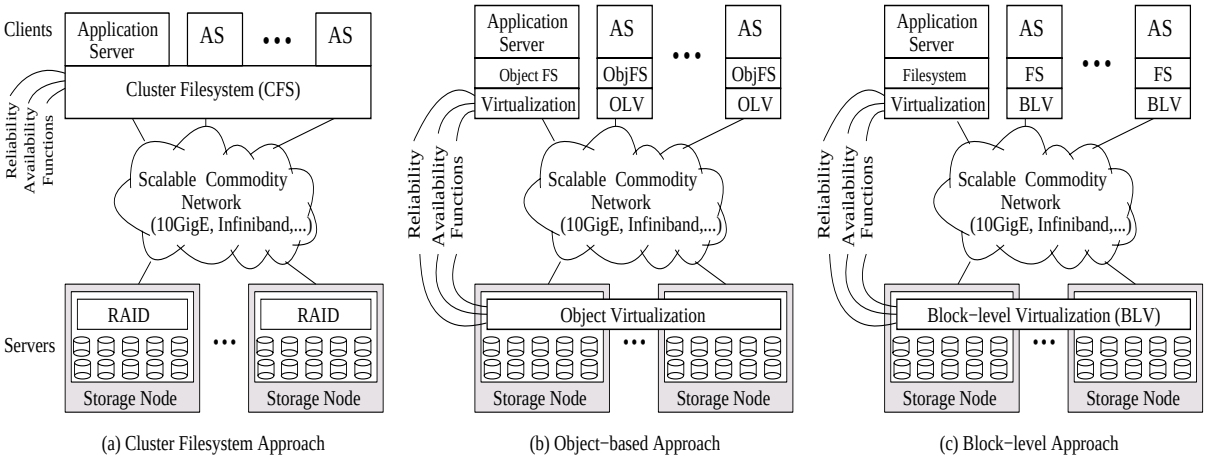


Figure 1.2: Reliability, Availability and Feature support in decentralized storage clusters.

data sharing and consistency. Tailoring storage to application needs, leads to reduced system cost and improved efficiency [Leavitt, 2008], yet current centralized systems can not be easily tailored to the needs of a particular application.

Support for reliability, availability and advanced functionality in decentralized storage clusters can be provided in either of three system layers: (a) at the filesystem, (b) at the object level, in the case of object-storage devices [Weber (Editor), 2004, 2007], and (c) at the block-level. These three cases and the corresponding system layers are illustrated in Figure 1.2.

In the first case, new functionality would be implemented within a cluster filesystem (e.g. GPFS [Schmuck and Haskin, 2002], Global File System (GFS) [Preslan et al., 1999]) that would run on the application servers. Functionality in such a filesystem would need to be implemented in a monolithic [Bonwick and Moore, 2008], or an extensible manner [Heidemann and Popek, 1994; Skinner and Wong, 1993; Zadok and Nieh, 2000]. In the block-level or object-level approaches, functions added would be implemented at the block or object level (e.g. Lustre [Sun Microsystems, 2007]) and executed either on the application servers or the storage nodes, with emphasis on the latter. Note also, that solutions at the object or block level would require changes to the filesystem, although one could argue that the changes in the block-level case would be less intrusive, involving mainly new control primitives, while preserving the traditional block-level API for data operations.

Although there are arguments for implementing features in each of the three layers, in this thesis we decide to follow the block-level approach. We believe that implementing advanced functionality at the block-level is appropriate for the following reasons:

- *Transparency* to applications and mildly-intrusive changes to filesystems. Users and organizations are usually unable or reluctant to change their applications and their filesystems, which are usually expensive and their features tightly controlled by vendors. On the other hand, block-level operation allows transparent evolution of the storage system.

- The storage hardware at the back-end has evolved significantly from simple disks and fixed controllers to powerful storage nodes [Gray, 2002; Gibson et al., 1998; Acharya et al., 1998] that offer block-level storage to multiple applications over a high-performance network. Block-level functions can *exploit* the processing capacity, hardware acceleration engines and memory *resources* of these storage nodes, whereas filesystem code (running mainly on the application servers and the clients) cannot.

- Off-loading the bulk of the processing to the block-level processors, and using the file server CPU only for coordination and I/O scheduling to individual storage nodes, allows the storage system to *scale* as new capacity is added. Furthermore, the *cost-effectiveness* of the system is increased by using more cheap commodity hardware instead of the expensive file-server hardware (processors, RAM, etc.).

- Storage management (e.g. migrating data, backup, redundancy) at the block level is *less complex to implement* and *easier to administer*, because of the unstructured form of data blocks.

- Operating at volume-level granularity is a requirement for many functions, such as compression, deduplication or encryption, in order to be efficient. Deduplication, for example, would find more duplicate data blocks when its search scope is an entire volume, than when it is restricted within a file or a file tree. In these cases of volume-wide optimization, knowledge of the file or object semantics is useless, and the block level is the right choice.

- Certain CPU-intensive functionality, such as compression, deduplication or encryption, may be *simpler* and/or more *efficient* to provide on unstructured fixed data blocks, rather than variable-sized files or objects, because (i) hardware accelerators can be more efficiently utilized for fixed-size blocks rather than variable-sized data, and (ii) blocks being more fine-grain than files or objects would incur less overhead in partial file reads or updates, where CPU-intensive operations (e.g. encryption) would be performed only on blocks read or written rather than the whole file each time.

- Current, high-performance disk controllers increasingly implement optimizations or features that alter the disk layout, such as block remapping [English and Alexander, 1992; Wang et al., 1999; Wilkes et al., 1996; Sivathanu et al., 2003] or logging [Wilkes et al., 1996; Stodolsky et al., 1994]. On these systems, many filesystem-level optimizations, such as log-structured writes or file defragmentation, besides doubling the implementation effort, are usually irrelevant or have adverse effects on performance. Furthermore, other features are being duplicated in the file and block layers, such as double caching in the buffer cache and the controller cache, also resulting in lower performance. These effects are caused by the well known issue of the "information gap", between the file and block layers [Denehy et al., 2002]. We believe that due to this gap, implementing functionality at the block level, and providing an *enhanced* block-level interface to the filesystem is advantageous.

The downside of working at the block level is that we have to restrict our system to the traditional block-level interface and lose the semantics of the file abstraction, and block liveness information [Sivathanu et al., 2003, 2004]. A higher-level API, such as objects, could be better in this case, and we are considering it for future work. However, our aim in this thesis is to explore the limits of the block-level API, before dealing with higher-level abstractions [Flouris et al., 2005].

1.2 Our Approach

An effective way of dealing with the scalability, reliability, flexibility and extensibility issues of a decentralized storage cluster is through *block-level virtualization*. The term "virtualization" has been used to describe mainly two concepts in storage systems: *indirection* and *sharing* [Flouris et al., 2005]. The first notion refers to the addition of an indirection layer between the physical resources and the logical abstractions accessed by applications. Such a layer allows administrators to create, and applications to access, various types of virtual volumes that are mapped to a set of physical devices on the storage system, while offering higher-level semantics through multiple layers of mappings. This kind of virtualization has several advantages:

i. It adds *flexibility* to the system configuration, since physical storage space can be arbitrarily mapped to virtual volumes. For example a virtual volume can be defined as any combination of basic operators, such as aggregation, partitioning, striping or mirroring. Thus storage can be

tailored to the requirements of a particular application, allowing trade-offs between dimensions such as speed, reliability, capacity, high-availability, security, data sharing and consistency [Leavitt, 2008; Bigelow, 2007].

ii. It enables numerous useful management and reconfiguration operations, such as non-disruptive data reorganization and migration during the system operation.

iii. It allows *off-loading* of important functionality from higher layers, such as fault-tolerance with RAID levels, backup with volume versions or encryption with encryption data filters.

iv. It allows *extensibility*, that is the addition of new functionality in the system. This is achieved by implementing new virtualization modules with the desired functions and incrementally adding them to the system.

Currently, the "indirection" virtualization mechanisms of existing storage systems and products mostly support a set of predefined functions and leave freedom to the administrator for combining them [Lee and Thekkath, 1996; MacCormick et al., 2004; Menon et al., 2003]. That is, they provide *configuration flexibility*, but little *extensibility*. For example predefined virtualization semantics include virtual volumes mapped to an aggregation of disks or RAID levels. Both research prototypes of volume managers [De Icaza et al., 1997; EVMS; GEOM; Lehey, 1999; Teigland and Mauelshagen, 2001] as well as commercial products [EMC Enginuity; EMC Symmetrix; HP OpenView; Veritas, b,c] belong in this category. In all these cases the storage administrator can switch on or off various features at the volume level. However, there is no support for extending the I/O protocol stack with new advanced functionality, such as versioning, encryption, compression, deduplication, virus-scanning, application-specific data placement or content hash computation, and it is not possible to combine such new functions in a flexible manner. The main reason behind this, is that such extensibility with advanced functions requires support for novel mechanisms, such as dynamic metadata management, end-to-end block-mapping and advanced control messaging.

Dynamic metadata management is essential in order to support advanced functionality at the virtualized block layer. The "dynamic" term denotes that such metadata, referring to a set of storage blocks, need to be processed and potentially updated during every block I/O operation. In contrast, some types of virtualization software, such as software RAID and logical volume managers, require small amounts of "static" metadata that store configuration information and change only when the system is reconfigured. A good example of a function that requires dynamic metadata support is online block-level

versioning. In this thesis we use block-level versioning as a guide for determining the merits, the mechanisms and the overheads associated with metadata-intensive functions at the block level.

The second notion of virtualization refers to the ability of *sharing* the storage resources across many hosts and multiple applications without compromising the performance of each application in any significant way. A networked filesystem (e.g. NFS) that allows many nodes to share access to the same files is an example of such a notion. Another example at the block level is a disk volume exported to multiple hosts via iSCSI [iSCSI] or ATA over Ethernet (AoE) [Hopkins and Coile]. In this case, there is currently no standard support for sharing volumes across nodes at the block level. Generally, even though some systems claim to support such sharing, either at the file or block level, efficient virtualization remains an elusive target in large-scale systems, because of the software complexity. Providing scalable resource sharing and maintaining consistency is complex and usually introduces significant overheads in the system.

We should note that the two notions of virtualization are orthogonal to each other. That is, an indirection layer can be used to map physical resources into logical ones, while the sharing layer is used on top to allow shared access of the logical storage resources to many nodes and applications. This organization maximizes the flexibility and the accessibility of the system.

1.3 Problems and Contributions

Our hypothesis is that *storage virtualization at the block-level with support for metadata management can be used to build scalable, reliable, available and customizable storage systems with advanced features*. To prove the hypothesis we address the following problems in a decentralized storage cluster:

1. Providing transparent, low-overhead versioning at the block level.

2. Generic support for advanced, metadata-intensive functionality at the block-level in distributed storage architectures.

3. Providing mechanisms for concurrent, scalable sharing of block-level storage.

4. Providing availability, replica consistency and recovery after failures.

Our contributions in this thesis at addressing the above problems are:

- We explore the mechanisms, the merits, and the overheads associated with metadata-intensive functions at the block level, with our work on *block-level versioning*. We find that block-level versioning has low overhead and offers advantages over filesystem-based approaches, such as transparency and simplicity.

- We design, implement and evaluate a *block-level storage virtualization framework* with support for metadata-intensive functions. We show that our framework, *Violin*: (i) significantly reduces the effort to introduce new functionality in the block I/O stack of a commodity storage node, and (ii) provides the ability to combine simple virtualization functions into hierarchies with semantics that can satisfy diverse application needs.

  We extend our single-node virtualization infrastructure, Violin, to Orchestra, an extended framework that allows almost arbitrary distribution of virtual hierarchies to application and storage nodes. providing a lot of flexibility in the mapping of system functionality to available resources.

- We design and implement locking and allocation mechanisms for concurrent, scalable sharing of block-level storage. Our framework supports shared dynamic metadata at the storage node side, but not at the application servers.

- We address consistency issues at the block level, providing simple abstractions and necessary protocols. Our approach, RIBD, uses *consistency intervals (CIs)*, a lightweight transactional mechanism, agreement, block-level versioning, and explicit locking, to address consistency of both replicas and metadata. We implement RIBD on a real prototype, in the Linux kernel. In the same area, we provide a taxonomy of alternative approaches and mechanisms for dealing with replica and metadata consistency. Finally, we evaluate RIBD in comparison with two popular filesystems, showing that RIBD performs comparably to systems with weaker consistency guarantees.

Next, in Section 1.4, we provide an overview of our work and contributions. Our contributions are presented in detail in chapters 3 to 6.

1.4 Overview

In this section we provide an overview of the work presented in the following chapters of the thesis.

1.4.1 Block-level versioning

Block-level versioning is a useful virtualization layer that provides transparent versioned volumes that can be used with multiple, third party, filesystems without the need for modifications. Data versions of the whole volume can be taken on demand and previous versions can be accessed online simultaneously with the current version, making recovery instant and easy. Moreover, it reduces complexity in higher layers of storage systems, namely the filesystem and storage management applications [Hutchinson et al., 1999]. Finally, it inherits all the block-level advantages mentioned previously in Section 1.2, such as exploiting the increased processing capabilities and memory sizes of active storage nodes and off-loading expensive host-processing overheads to the disk subsystem, thus increasing the scalability of a storage archival system [Hutchinson et al., 1999].

Block-level versioning poses several challenges as well:

1. Memory and disk space overhead: Because we only have access to blocks of information, depending on application data access patterns, there is increased danger for higher space overhead in storing previous versions of data and the related metadata.

2. I/O path performance overhead: It is not clear at what cost versioning functionality can be provided at the block level.

3. Consistency of the versioned data when the versioned volume is used in conjunction with a filesystem on top.

4. Versioning granularity: Since versioning occurs at a lower system layer, information about the content of data is not available, as is, for instance, the case when versioning is implemented in the filesystem or the application level. Thus, we only have access to versions of whole volumes as opposed to individual files.

In Chapter 3 we present *Clotho* a system that provides online versioning at the block level and addresses all above issues, demonstrating that this can be done at minimal space and performance overheads.

1.4.2 Storage resource sharing and management

Scalable storage systems provide a means of consolidating all storage in a single system and increasing storage efficiency. However, storage consolidation leads to increased requirements for “flexibility” that

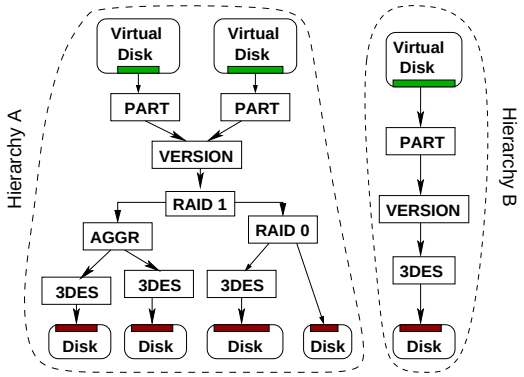


Figure 1.3: A virtual device graph in Violin.

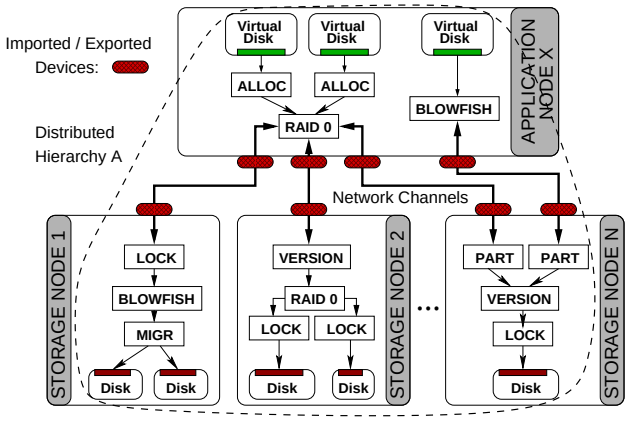


Figure 1.4: A distributed Orchestra hierarchy.

will be able to serve multiple applications and their diverse needs. This flexibility refers to both storage management and application access issues and is usually provided through virtualization techniques: administrators and applications see various types of virtual volumes that are mapped to physical devices but offer higher-level semantics through virtualization mechanisms.

In Chapter 4 of this thesis we address this problem by providing a kernel-level framework for (i) building and (ii) combining virtualization functions. We design, implement, and evaluate Violin (Virtual I/O Layer INtegrator), a virtual I/O framework for storage nodes that replaces the current block-level I/O stack with an improved I/O hierarchy that allows for (i) easy extension of the storage hierarchy with new mechanisms and (ii) flexibly combining these mechanisms to create modular hierarchies with rich semantics.

Although our approach shares similarities with work in modular and extensible filesystems [Heidemann and Popek, 1994; Schermerhorn et al., 2001; Skinner and Wong, 1993; Zadok and Nieh, 2000] and network protocol stacks [Kohler et al., 2000; Mosberger and Peterson, 1996; O'Malley and Peterson, 1992; Van Renesse et al., 1995], existing techniques from these areas are not directly applicable to *block-level* storage virtualization. A fundamental difference from network stacks is that the latter are essentially stateless (except for configuration information) and packets are ephemeral, whereas storage blocks and their associated metadata need to persist. Compared to extensible filesystems, block-level storage systems operate at a different granularity, with no information about the relationships and lifecycle of blocks. Thus, metadata need to be maintained at the block level resulting potentially in large memory overhead. Moreover, block I/O operations cannot be associated precisely with each other, limiting possible optimizations.

The main contributions of Violin are: (i) it significantly reduces the effort to introduce new functionality in the block I/O stack of a storage node and (ii) it provides the ability to combine simple virtualization functions into hierarchies with semantics that can satisfy diverse application needs. Violin provides virtual devices with full access to both the request and completion paths of I/Os allowing for easy implementation of synchronous and asynchronous I/O. Supporting asynchronous I/O is important for performance reasons, but also raises significant challenges when implemented in real systems. Also, Violin deals with metadata persistence for the full storage hierarchy, off-loading the related complexity from individual virtual devices. To achieve flexibility, Violin allows storage administrators to create arbitrary, directed acyclic graphs of virtual devices, each adding to the functionality of the successor devices in the graph. In each hierarchy, the blocks of each virtual device can be mapped in arbitrary ways to the successor devices, enabling advanced storage functions, such as dynamic block remapping. An example of a virtual device graph in Violin is shown in Figure 1.3.

In Chapter 4, we evaluate the effectiveness of Violin in three areas: ease of module development, configuration flexibility, and performance. In the first area we are able to quickly prototype modules for RAID levels (0, 1 & 5), versioning, partitioning, aggregation, MD5 hashing, migration and encryption. Further examples of useful functionality that can be implemented as modules include all kinds of encryption algorithms, deduplication, compression, storage virus-scanning modules, online migration and transparent disk layout algorithms (e.g. log-structured allocation or cylinder group placement). In many cases, writing a new module is just a matter of recompiling existing user-level library code. Overall, using Violin encourages the development of simple virtual modules that can later be combined to more complex hierarchies. Regarding configuration flexibility, we are able to easily configure I/O hierarchies that combine the functionality of multiple layers and provide complex high-level semantics that are difficult to achieve otherwise. Finally, we use Postmark [Katcher] and IOmeter [Iometer] benchmarks to examine the overhead that Violin introduces over traditional block-level I/O hierarchies. We find that overall, internal modules perform within 10% (in terms of throughput) of their native Linux block-driver counterparts.

Multi-storage-node Virtualization

Our approach towards a scalable virtualization platform is to enhance the Violin framework with the necessary primitives in order to allow multi-node (distributed) hierarchies. We find that two primitives are required to achieve this, locking of layer metadata, and control messaging through the distributed

I/O stack.

Orchestra, presented in Chapter 5, builds on Violin and is aimed to take advantage of decentralized storage architectures. Orchestra allows exporting of storage volumes to other storage nodes through block-level network protocols, such as iSCSI, NBD or other protocols. The nodes can further build I/O stacks on top of the remote storage volumes, creating higher-level volumes as needed. Orchestra manages I/O requests through these distributed hierarchies by the layers distributed to different nodes. Reliability issues regarding node or device failures are effectively handled by fault-tolerance layers (e.g. RAID levels) that map storage volumes between nodes or devices. The system can thus tolerate storage node failures as a RAID system tolerates device failures.

To sum up, our main contribution in this area is that we provide a novel approach for building cluster storage systems which consolidates system complexity in a single point, the Orchestra hierarchies. In Chapter 5, we examine how extensible, block-level I/O paths can be supported over distributed storage systems. Our proposed infrastructure deals with metadata persistence and control messaging, and allows for hierarchies to be distributed almost arbitrarily over application and storage nodes, providing a lot of flexibility in the mapping of system functionality to available resources. An example of a distributed Orchestra hierarchy is shown in Figure 1.4.

1.4.3 Scalable storage distribution and sharing

In this area we examine how sharing can be provided at the block level, that is how multiple hosts and applications can share storage volumes. Our approach is to design block-level locking and allocation facilities that can be inserted in Orchestra's distributed I/O hierarchies where required. A novel feature of this approach is that allocation and locking are both provided as *in-band mechanisms*, i.e. they are part of the I/O stack and are not provided as external services, such as distributed lock managers [Preslan et al., 2000; Schmuck and Haskin, 2002]. We believe that our approach simplifies the overall design of a storage system and leaves more flexibility for determining the most appropriate hardware/software boundary.

Orchestra, presented in Chapter 5, not only supports distributed virtualization hierarchies, but also implements sharing mechanisms for the virtualized resources on brick-based architectures. Our main goal for providing such support is to ease the development, deployment, and adaptation of shared storage services for various environments and application needs. We show that providing flexibility does not introduce significant overheads and does not limit scalability.

In the same chapter we examine how higher system layers can benefit from an enhanced block interface. Orchestra's support for distributed block locking, allocation, and metadata persistence can significantly simplify distributed filesystem design. We design and implement a *stateless*, *pass-through* filesystem, called `0FS`, that builds on Orchestra's advanced features and distributed virtualization mechanisms. In particular, this filesystem provides basic file and directory semantics and I/O operations, while it (i) does *not* maintain any internal distributed state, and (ii) does not require explicit communication among its instances running on different nodes.

We find that our approach significantly reduces system complexity by (i) moving many functions currently performed in higher system layers to the block-level store and (ii) providing an extensible framework that supports adding simple modules in Orchestra hierarchies that can span storage and application nodes. Performance-wise, we find that Orchestra introduces little overhead beyond existing, monolithic implementations that provide the same functionality. Moreover, scalability results show that our approach has the potential for scaling to large system sizes, because of its block-based, asynchronous operation.

1.4.4 Reliability and Availability

A decentralized storage architecture is appealing because the elimination of all centralization points increases I/O request asynchrony and allows the system to scale in capacity and performance. In a decentralized architecture, however, dealing with failures and always maintaining a consistent state of the system becomes a real challenge. This is exacerbated by reliance on commodity (and thus less reliable) equipment. Availability and survivability under system failures in this case is complicated and may result in significant performance penalties. In our view, there are two main issues in achieving availability, fault-tolerance and recoverability in a decentralized storage architecture:

- *Consistent redundancy scheme:* In centralized or single-node storage architectures, consistency of redundant data used for availability (i.e. replicas, parity or erasure codes) is ensured through a hardware or software RAID layer that manages redundancy through reliable hardware channels. A decentralized storage system, however, must ensure the consistency of redundant data (e.g. data replicas) across storage nodes. Logging or distributed agreement protocols (e.g. voting or two-phase commit) are used for this purpose in a variety of storage architectures [Saito et al., 2004; Schmuck and Haskin, 2002; Hartman and Ousterhout, 1992; Liskov et al., 1991].

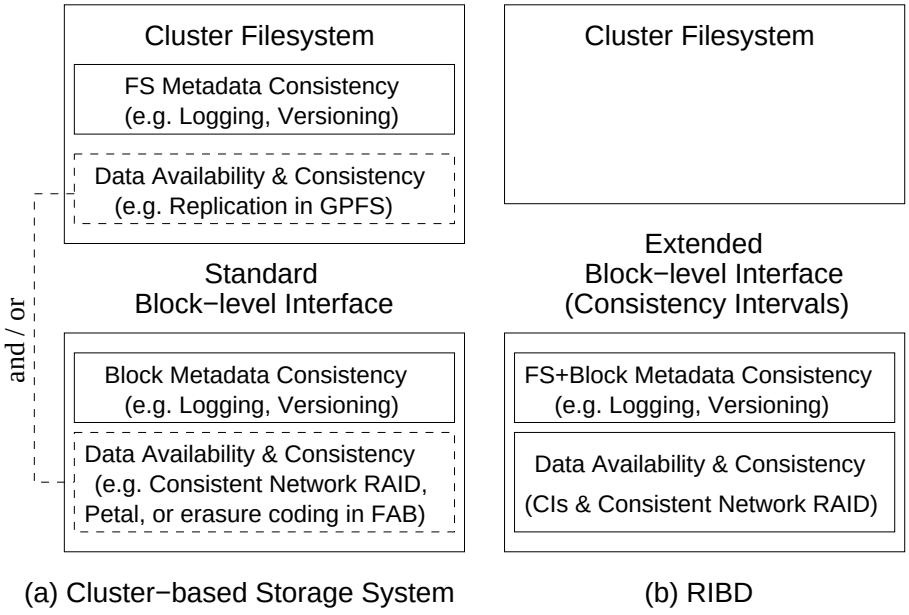


Figure 1.5: Overview of typical cluster-based scalable storage architectures and RIBD.

- *Recovery of data and metadata:* A number of component failures should be tolerated and the system should remain available, depending on the redundancy scheme being used. However, in the event of a global (non-tolerated) failure, data and system metadata must be recoverable to a consistent state so that normal system operation can either continue or be restarted without the need to reconstruct extensive state, e.g. by performing extensive consistency checks at the file level with `fsck`. Typically, this problem is addressed by using journaling techniques [Hagmann, 1987; Seltzer et al., 2000; Tweedie, 2000].

Cluster-based storage architectures in use today resemble the structure shown in Figure 1.5-(a) where a high-level layer such as a filesystem uses the services of an underlying *block-level storage system* through a standard block-level interface such as SCSI. Concerns such as data availability and metadata recovery after failures are commonly addressed through data redundancy (e.g., consistent replication) at the file or block (or both) layers, and by metadata journaling techniques at the file layer. This structure, where both redundancy and metadata consistency is built into the filesystem, leads to filesystems that are complex, hard to scale, debug and tune for specific application domains [Prabhakaran et al., 2005a; Yang et al., 2004; Sivathanu et al., 2005a; Prabhakaran et al., 2005b].

Modern high-end storage systems incorporate increasingly advanced features such as thin provisioning [Compellent, 2008; IBM Corp., 2008c], snapshots [IBM Corp., 2008d], volume management

[EVMS; GEOM; Teigland and Mauelshagen, 2001], data deduplication [Quinlan and Dorward, 2002], block remapping [English and Alexander, 1992; Wang et al., 1999; Wilkes et al., 1996; Sivathanu et al., 2003], and logging [Wilkes et al., 1996; Stodolsky et al., 1994]. The implementation of all such advanced features requires the use of significant block-level metadata, which are kept consistent using techniques similar to those used in filesystems (Figure 1.5-(a)). Filesystems running over such advanced systems often duplicate effort and complexity by focusing on file-level optimizations such as log-structured writes, or file defragmentation. Besides doubling the implementation and debugging effort, these file-level optimizations are usually irrelevant or adversely impact performance [Denehy et al., 2002].

In Chapter 6 we propose *Recoverable Independent Block Devices (RIBD)*, an alternative storage system structure (depicted in Figure 1.5-(b)) that moves the necessary support for handling both data and metadata consistency issues to the block layer in decentralized commodity platforms. RIBD extends Orchestra presented in Chapter 5, introducing the notion of *consistency intervals* (CIs) to provide fine-grain consistency semantics on sequences of block level operations by means of a lightweight transaction mechanism. Our model of CIs is related to the notion of *atomic recovery units* (ARUs), which were defined in the context of a local, centralized system [Grimm et al., 1996]. A key distinction between CIs and ARUs is that CIs handle both atomicity and consistency over multiple distributed copies of data and/or metadata, whereas ARUs do not face consistency issues within a single system. RIBD extends the traditional block I/O interface with commands to delineate CIs, offering a simple yet powerful interface to the file layer.

One distinctive feature of RIBD is its roll-back recovery mechanism based on low-cost versioning, that allows recovery and access not only to the latest system state (as journaling does), but to *a set of previous states*. This allows recovery from malicious attacks and human errors, as discussed in detail in Section 3.1. To the best of our knowledge, RIBD is the first system to propose and demonstrate the combination of versioning and lightweight transactions for recovery purposes at the block storage level.

RIBD has been implemented as a kernel module in the Linux kernel, and evaluated on a cluster of 24 nodes. To understand the overheads in our approach, we compare RIBD with two popular filesystems, PVFS [Carns et al., 2000] and GFS [Preslan et al., 1999], which however offer weaker consistency guarantees. We find that RIBD offers a simple and clean interface to higher system layers and performs comparably to existing solutions with weaker guarantees.

1.5 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 surveys previous and related work for this thesis.

Chapter 3 presents our work in the block-level versioning area with Clotho, demonstrating the applications, limits and overheads of metadata-intensive tasks at the block level. The content of this chapter roughly corresponds to publication [Flouris and Bilas, 2004].

Chapter 4 presents the foundation of our work on single-host block-level virtualization with Violin, a virtualization framework that allows (i) building and (ii) combining virtualization functions to create extensible virtual hierarchies. The content of Chapter 4 corresponds roughly to publication [Flouris and Bilas, 2005].

Chapter 5 presents our work on Orchestra, which extends Violin, allowing distributed placement of virtualization functions and supporting mechanisms for sharing virtualized storage to many hosts and applications. The content of this chapter roughly corresponds to publication [Flouris et al., 2008].

Chapter 6 presents our work on Recoverable Independent Block Devices (RIBD), which have been designed and implemented by extending Orchestra. The content of this chapter, with an early version of the protocol design and without the implementation and evaluation has been presented in [Flouris et al., 2006].

Finally, Chapter 7 concludes this thesis and presents directions for future research.

Chapter 2

Related Work

True knowledge exists in knowing that you know nothing.

—Socrates (469 BC - 399 BC)

The idea of moving application-specific or OS-specific code closer to the disks has been proposed by three different research groups at approximately the same period. “Network-attached secure disks” (or NASD) [Gibson et al., 1998], IDisks [Keeton et al., 1998] and “active disks” [Acharya et al., 1998] are based on similar concepts. NASD goes one step further in proposing an *object-based* storage API instead of the traditional block-level interface. Several publications on active disks and object storage devices (OSD) have followed these first projects [Wang et al., 1999; Uysal et al., 2000; Sivathanu et al., 2002; Yokota, 2000; Akinlar and Mukherjee, 2000]. Although these systems have provided motivation for our work, there are differences between these approaches and our work, as discussed next.

The Active Disk and IDisks projects propose the use of the on-disk controller specifically for database and decision support applications. However, the IDisk project has not explored the issues of what functions would be appropriate for running on the disk controller, what would be their runtime, and has not evaluated a real or simulated prototypes [Keeton et al., 1998]. On the other hand, the Active Disk project [Acharya et al., 1998; Uysal et al., 2000] has proposed and evaluated a runtime system with functions, called disklets, running on the disk controllers. The disklets, however, have very different characteristics from the functions we use, and although they run at the block-level, close to the disk, they cannot be called *virtualization* functions, for the following reasons: (i) Disklets are executed in a stream-based, sandboxed runtime, and their code is intended for data-filtering. As such, it is allowed to deal only with data stream processing, not with metadata. (ii) Disklets are not allowed to allocate

or free memory, to initiate I/O on their own (e.g. for writing metadata), or to change the I/O addresses of the input or output streams (e.g. to remap blocks as we do in versioning). As a result, the disklet model would not be appropriate even for writing a simple RAID-0 virtualization function. In contrast, the virtualization framework we build supports all these operations as well as metadata-management for I/O blocks.

Object-based storage devices (OSD), although not widely available yet [Mesnier et al., 2003], have been recently ratified as a new storage standard [Weber (Editor), 2004, 2007]. OSDs provide built-in support for metadata store and retrieval operations for objects, as well as support for collections of objects. However, the use of metadata in object storage has been explored mostly for storing file attributes in objects and providing security for objects, as in NASD [Gibson et al., 1998]. To our knowledge, no system based on OSDs has explored the use of object metadata for moving advanced metadata-intensive functionality, such as versioning, deduplication or compression, from the filesystem to the object store. Moreover, we have no reference of an extensible framework for object-based storage. However, if such a system is built, we expect that in order to provide extensibility and flexibility transparently, it will use the mechanisms we use in our block-level framework, such as an extended object command API for control, logical-to-physical object mappings, and a hierarchy of virtual object-level modules on OSDs. Also it would have to use similar availability and consistency protocols as the ones we propose. Finally, it should be mentioned that using our block-level virtualization framework, we can build an OSD layer with features similar to the ones described in the OSD standards [Weber (Editor), 2004, 2007]. As a result, we consider our work more generic than an object-level virtualization framework.

Implementing functionality below the block-level API has also been proposed in semantically-smart disks [Arpaci-Dusseau et al., 2006; Sivathanu et al., 2003]. This approach explores the capability of inferring, within the block-level subsystem, essential information missing from the higher layers. Such block-level metadata include block liveness, file system structure and block relationships [Sivathanu et al., 2004, 2005b]. Knowledge of such information at the block level would enable off-loading of many useful tasks, such as secure block deletion, data placement optimizations and garbage collection. The gray-box approach employed by semantically-smart disks is interesting, however, in many cases it incurs a risk of false inference (e.g. about block liveness in the file system) that may lead to data loss. In our approach, the goal is not to infer block-level metadata from higher layers, but to provide advanced functionality at the block level through an enhanced block API.

A block-level virtualization system, and perhaps the closest system to our work regarding the virtu-

alization aspects, is Petal [Lee and Thekkath, 1996] and its distributed filesystem Frangipani [Thekkath et al., 1997]. Petal's main focus was making storage management easier and for this reason it provides some advanced functionality features, such as dynamically configuring and reconfiguring virtual disk volumes over a pool of physical disk storage, automatically adding to or removing physical disks from the pool, providing high-availability with chained declustering and providing block-level versioning using copy-on-write and storing metadata for blocks.

Regarding block-level versioning, our scheme is close to Petal's, however our algorithm has lower overhead, because it does not perform copy-on-write, but rather remap-on-write, using at the same time sub-extent bitmaps showing the valid sub-extents. We thus avoid the copy on each write, trading disk capacity with write performance. On the other hand the remap-on-write approach alters data placement and may affect read performance. Moreover, in contrast to Petal, we propose, implement and evaluate differential compression for blocks belonging to subsequent versions in order to increase disk space efficiency.

In terms of features, we can classify Petal as being rich in features and supporting dynamic metadata for blocks, however its feature set is fixed, that is, it is not extensible. Instead, we support modular hierarchies, flexible to configure and easy to add new functionality, such as deduplication, encryption, or tiered-storage. Thus, while we can create volumes with application-specific functionality, Petal cannot. Also, metadata in our system depend on the stack of functions used in a volume and are not of a fixed size, potentially consuming resources, when the application does not use all features.

In Orchestra, presented in Chapter 5, we offer block-level sharing mechanisms for virtual volumes. Petal-Frangipani does not offer such block-level mechanisms, but uses filesystem-based locking, through a lock service. An advantage of our in-band, block-level locking at the storage nodes, is that it can scale as more nodes are added. Moreover, Petal-Frangipani implements the distributed block allocation at the filesystem level, while we provide it through an in-band mechanism at the block level.

Petal uses write-ahead logging for determining replica consistency after failures. We, on the other hand, propose and evaluate a lightweight transactional API coupled with low-overhead block-level versioning. We believe that our approach offers the same strong consistency guarantees with scalable performance, since versioning is a particularly promising approach for building reliable storage systems as it matches current disk capacity/cost tradeoffs.

In the rest of this chapter we examine previous and related work for each of the Chapters 3 to 6 of this thesis. Section 2.1 presents previous work related to block-level versioning, such as WAFL [Hitz

et al., 1994], SnapMirror [Patterson et al., 2002], Petal [Lee and Thekkath, 1996], Venti [Quinlan and Dorward, 2002] and CVFS [Soules et al., 2003]. Work related to (a) extensible filesystems (e.g. Ficus [Heidemann and Popek, 1994], Vnodes [Skinner and Wong, 1993], FiST [Zadok and Nieh, 2000]), (b) extensible network protocols (e.g. Click Router [Kohler et al., 2000], X-kernel [O'Malley and Peterson, 1992], Horus [Van Renesse et al., 1995]), and (c) block-level storage virtualization (e.g. EVMS [EVMS], GEOM [GEOM]) is presented in Section 2.2. Previous work on shared cluster storage systems (e.g. Petal-Frangipani [Lee and Thekkath, 1996; Thekkath et al., 1997], Boxwood [MacCormick et al., 2004], Swarm [Hartman et al., 1999]) is discussed in Section 2.3. Finally, work related to consistency and availability issues in cluster storage systems and a detailed taxonomy of existing solutions is presented in Section 2.4.

2.1 Clotho: Block-level Versioning

A number of projects have highlighted the importance and issues in storage management [Gray, 1999; Kubiatowicz et al., 2000; Patterson, 2000; Wilkes, 2001]. Our goal in Clotho, presented in Chapter 3, is to define innovative functionality that can be used in future storage protocols and APIs to reduce management overheads.

Block-level versioning was recently discussed and used in WAFL [Hitz et al., 1994], a filesystem designed for Network Appliance's NFS appliance. WAFL works in the block-level of the filesystem and can create up to 20 snapshots of a volume and keep them available online through NFS. However, since WAFL is a filesystem and works in an NFS appliance, this approach depends on the filesystem. In Chapter 3 we demonstrate that Clotho is filesystem agnostic by presenting experimental data with two production-level filesystems. Moreover, WAFL can manage a limited number of versions (up to 20), whereas Clotho can manage a practically unlimited number. Hitz et al. [Hitz et al., 1994] mention that WAFL's performance cannot be compared to other general purpose filesystems, since it runs on a specialized NFS appliance and much of its performance comes from its NFS-specific tuning. Hutchinson et al. [Hutchinson et al., 1999] use WAFL to compare the performance of filesystem- and block-level-based snapshots (within WAFL). They advocate the use of block-level backup, due to cost and performance reasons. However, they do not provide any evidence on the performance overhead of block-level versioned disks compared to regular, non-versioned block devices. In Chapter 3 we thoroughly evaluate this with both micro-benchmarks as well as standard workloads. SnapMirror [Patterson et al., 2002]

is an extension of WAFL, which introduces management of remote replicas in WAFL's snapshots to optimize data transfer and ensure consistency.

Venti [Quinlan and Dorward, 2002] is a block-level network storage service, intended as a repository for backup data. Venti follows a write-once storage model and uses content based addressing by means of hash functions to identify blocks with identical content. Instead, Clotho uses differential compression concepts. Furthermore, Venti does not support versioning features. Clotho and Venti are designed to perform complementary tasks, the former to version data and the latter as a repository to store safely the archived data blocks over the network.

Distributed block-level versioning support was included in Petal [Lee and Thekkath, 1996]. Although Petal's versioning concepts are similar to Clotho, Petal uses a copy-on-write algorithm, while we propose remap-on-write and subextent addressing to minimize both write overhead and memory footprint. Moreover, in contrast to Petal, we propose, implement and evaluate differential compression for blocks belonging to subsequent versions in order to increase disk space efficiency.

Since backup and archival of data is an important problem, there are many products available that try to address the related issues. However, specific information about these systems and their performance with commodity hardware, filesystems, or well-known benchmarks are scarce. LiveBackup [Storactive, 2002] captures changes at the file level on client machines and sends modifications to a back-end server that archives previous file versions. EMC's SnapView [SnapView] runs on the CLARiiON storage servers at the block level and uses a "copy-on-first-write" algorithm. However, it can capture only up to 8 snapshots and its copy algorithm does not use logging block allocation to speed up writes. Instead, it copies the old block data to hidden storage space on every first write, overwriting another block. Veritas's FlashSnap [Veritas, a] software works inside the Veritas File System, and thus, unlike Clotho, is not filesystem agnostic. Furthermore it supports only up to 32 snapshots of volumes. Sun's Instant Image [Sun Microsystems, 2008] works also at the block-level in the Sun StorEdge storage servers. Its operation appears similar to Clotho. However, it is used through drivers and programs in the Sun's StorEdge architecture, which runs only through the Solaris architecture and is also filesystem aware.

Each of the above systems, especially the commercial ones, uses proprietary customized hardware and system software, which makes comparisons with commodity hardware and general purpose operating systems difficult. Moreover, these systems are intended as standalone services within centralized storage appliances, whereas Clotho is designed as a transparent autonomous block-lever layer for active storage devices and appropriate for pushing functionality closer to the physical disk. In this direction,

Clotho categorizes the challenges of implementing block-level versioning and addresses the related problems.

De Jonge et al. [De Jonge et al., 1993] examine the possibility of introducing an additional layer in the I/O device stack to provide certain functionality at lower system layers, which also affect the functionality that is provided by the filesystem. Other efforts in this direction mostly include work in logical volume management and storage virtualization that try to create a higher level abstraction on-top of simple block devices. Teigland et al. [Teigland and Mauelshagen, 2001] present a survey of such systems for Linux. Such systems usually provide the abstraction of a block-level volume that can be partitioned, aggregated, expanded, or shrunk on demand. Other such efforts [De Icaza et al., 1997] add RAID capabilities to arbitrary block devices. Our work is complementary to these efforts and proposes adding versioning capabilities to the block-device level.

Other previous work in versioning data has mostly been performed either at the filesystem layer or at higher layers. Santry et al. [Santry et al., 1999] propose versioning of data at the file level, discussing how the filesystem can transparently maintain file versions as well as how these can be cleaned up. Moran et al. [Moran et al., 1993] try to achieve similar functionality by providing mount points to previous versions of directories and files. They propose a solution that does not require kernel-level modification but relies on a set of user processes to capture user requests to files and to communicate with a back-end storage server that archives previous file versions. Other, similar efforts [Olson, 1993; Pike et al., 1990; Roome, 1992; Soules et al., 2003; Strunk et al., 2000] approach the problem at the filesystem level as well and either provide the ability for checkpointing of data or explicitly manage time as an additional file attribute.

Self-securing storage [Strunk et al., 2000] and its filesystem, CVFS [Soules et al., 2003] target secure storage systems and operate at the filesystem level. Some of the versioning concepts in self-securing storage and CVFS are similar to Clotho, but there are numerous differences as well. The most significant one is that self-securing storage policies are not intended for data archiving and thus, retain versions of data for a short period of time called *detection window*. No versions are guaranteed to exist outside this window of time and no version management control is provided for specifying higher-level policies. CVFS introduces certain interesting concepts for reducing metadata space, which however, are also geared towards security and are not intended for archival purposes. Since certain concepts in [Soules et al., 2003; Strunk et al., 2000] are similar to Clotho, we believe that a block-level self-secure storage system could be based on Clotho, separating the orthogonal versioning and security functionalities in

different subsystems.

Systems more recent than Clotho, such as the VDisk [Wires and Feeley, 2007], have focused on "secure" versioning st the block level, where the versioning layer is protected within a virtual machine. The advantage of the VDisk is that storage data and metadata are additionally protected from operating system bugs that may cause corruption. On the other hand, VDisk uses expensive metadata management techniques which result to 50% performance penalty on block writes. We believe that similar protection from operating system faults with a much lower overhead, can be provided by encapsulating Clotho in a virtual machine or by embedding it in a hardware disk controller.

2.2 Violin: Extensible Block-level Storage

In this section we comment on previous and related work on (a) extensible filesystems, (b) extensible network protocols, and (c) block-level storage virtualization.

2.2.1 Extensible filesystems

Storage extensions can be implemented at various levels in a storage system. The highest level is within the filesystem, where combined software layers implement the desired functionality. The concept of layered filesystems has been explored in previous research.

Ficus [Heidemann and Popek, 1994], one of the earliest approaches, proposes a filesystem with a stackable layer architecture, where desired functionality is achieved by loading appropriate modules. A later approach [Skinner and Wong, 1993] proposes layered filesystem implementation by extending the vnode/VFS interface to allow "stacking" vnodes. A similar concept at the user-level, was proposed in the recent Mona FS [Schermerhorn et al., 2001], which allows application extensions to files through streaming abstractions. Another recent approach, FiST [Zadok and Nieh, 2000], uses a high-level specification language and a compiler to produce a filesystem with the desired features. FiST filesystems are built as extensions on top of *baseFS*, a native low-level filesystem. A similar concept was used in the Exokernel [Kaashoek et al., 1997], an extensible OS that comes with XN, a low-level in-kernel storage system. XN allows users to build library filesystems with their desired features. However, developing library filesystems for the Exokernel requires significant effort and can be as difficult as developing a native monolithic FS for an operating system.

Our work on Violin, presented in Chapter 4, shares similarity with these approaches to the extent that

we are also dealing with extensible storage stacks. However, the fact that we operate at the block-level in the storage system requires that our framework, provides different APIs to modules, uses different I/O abstractions for its services, and different kernel facilities in its implementation. Thus, although the high level concept is similar to extensible filesystems, the challenges faced and solutions provided are different.

Implementing storage extensions at either the filesystem level or the storage system level is desirable, each for different reasons and each approach is not exclusive of the other. The basic pros and cons of each approach stem from the API and metadata each level is aware of. One fundamental characteristic of filesystems, for example, is that they have file metadata. Thus, they can associate blocks that belong to the same file and are able to provide policies at the file level. On the other hand, storage systems operate at the block-level and they have no information about the relationships of blocks. Thus, metadata need to be maintained at the block level resulting potentially in large memory overhead. Moreover, block I/O operations cannot be associated precisely with each other, limiting possible optimizations. On the positive side, block-level extensions are transparent to any filesystem and volume-level policies can be provided. Moreover, many block-level functions, e.g. encryption or RAID-levels, can operate faster than at the filesystem level, since they operate on raw fixed-size blocks and not on the structured variable-sized data of the filesystem.

2.2.2 Extensible network protocols

Our work on storage virtualization shares similarities with frameworks for layered network stacks. The main efforts in this direction are the Click Router [Kohler et al., 2000], X-kernel [O'Malley and Peterson, 1992], and Horus [Van Renesse et al., 1995]. All three approaches aim at building a framework for synthesizing network protocols from simpler elements. They use a graph representation for layering protocols, they envision simple elements in each protocol, and provide mechanisms for combining these simple elements in hierarchies with rich semantics and low performance overhead. Scout [Mosberger and Peterson, 1996] is a communication-centric OS, also supporting stackable protocols. The main abstraction of Scout is the I/O paths between data sources and sinks, an idea applicable both to network and storage stacks.

In Violin we are interested in providing a similar framework for block-level storage systems. We share the same observation that there is an increased need to extend the functionality of block-level storage (network protocol stacks) and that doing so is a challenging task with many implications on

storage (network) infrastructure design. However, the underlying issues in network and storage stacks are different:

- Network stacks distinguish flows of self-contained packets, while storage stacks cannot distinguish flows, but map data blocks from one device to another.

- Network and storage stacks exhibit fundamentally different requirements for *state persistence.* Network stacks do not need to remember where they scheduled every packet in order to recover it at a later time, and thus, do not require extensive metadata. On the other hand, storage stacks must be able to reconstruct the data path for each I/O request passing through the stack, requiring often times large amounts of persistent metadata.

- Send and receive paths in network protocol stacks are fairly independent, whereas in a storage hierarchy there is strong coupling between the request and completion paths for each read and write request. Moreover, an issue not present in network protocol stacks is the need for asynchronous handling of I/O requests and completions, which introduces additional complexity in the system design and implementation.

2.2.3 Block-level storage virtualization

The most popular virtualization software is volume managers. The two most advanced open-source volume managers currently are EVMS and GEOM. EVMS [EVMS], is a user-level distributed volume manager for Linux. It uses the MD [De Icaza et al., 1997] and device-mapper kernel modules to support user-level plug-ins called *features.* However, it does not offer persistent metadata or block remapping primitives to these plug-ins. Moreover, EVMS focuses on configuration flexibility with predefined storage semantics (e.g. RAID levels) and does not easily allow generic extensions (e.g. versioning). GEOM [GEOM] is a stackable BIO subsystem under development for FreeBSD. The concepts behind GEOM are, to our knowledge, the closest to Violin. However, GEOM does not support persistent metadata which, combined with dynamic block mapping, are necessary for advanced modules such as versioning discussed in Chapter 3. LVM [Teigland and Mauelshagen, 2001] and Vinum [Lehey, 1999] are simpler versions of EVMS and GEOM. Violin has all the configuration and flexibility features of a volume manager coupled with the ability to write extension modules with arbitrary virtualization semantics.

Besides open-source software, there exist numerous virtualization solutions in the industry. HP OpenView Storage Node Manager [HP OpenView] helps administrators control, plan, and manage direct-attached and networked storage, acting as a central management console. EMC Enginuity [EMC Enginuity], a storage operating environment for high-end storage clusters, employs various techniques to deliver optimized performance, availability and data integrity. Veritas Volume Manager [Veritas, c] and Veritas File System aim at assisting with online storage management. Similar to other volume managers, physical disks can be grouped into logical volumes to improve disk utilization and eliminate storage-related downtime. Moreover, administrators have the ability to move data between different storage arrays, balance I/O across multiple paths to improve performance, and replicate data to remote sites for higher availability. However, in all cases, the offered virtualization functions are predefined and they do not seem to support extensibility of the I/O stack with new features. For instance, EMC Enginuity currently supports only the following predefined data protection options: RAID-1, Parity RAID (1 parity disk per 3 or 7 data disks), and RAID-5 [EMC Symmetrix].

RAIDframe [CourtrightII et al., 1996] enables rapid prototyping and evaluation of RAID architectures, which is similar to, but narrower than, our goals. RAIDframe uses a DAG representation for specifying RAID architectures, similar to Violin. However, its goal is to evaluate certain architectural parameters (encoding, mapping, caching) and not to provide extensible I/O hierarchies with arbitrary virtualization functions.

The latest versions of the Windows OS support an integrated model for building device drivers, the Windows Driver Model (WDM) [Oney]. This model specifies the runtime support available to writers of device drivers. However, unlike Violin for block-level devices, it only provides generic kernel support and does not include functionality specific to storage.

Research in virtual machines, such as [Barham et al., 2003; Sugerman et al., 2001] aims at virtualizing hardware resources so that multiple instances of the same I/O resource are shared in a safe manner by different instances of the same or different operating systems. Our work is orthogonal and aims at providing additional functionality on top of disk storage resources and within a single operating system.

2.3 Orchestra: Extensible Networked-storage Virtualization

In this section we present previous and related work on building storage systems for high performance clusters of servers. Next, we discuss the limitations of current, conventional solutions and we review

prototypes that share similar goals with Orchestra, contrasting our approach to other contributions.

2.3.1 Conventional cluster storage systems

Currently, building scalable storage systems that provide storage sharing for multiple applications relies on layering a distributed filesystem on top of a pool of block-level storage. This approach is dictated by the fact that block-level storage has limited semantics that do not allow for performing advanced storage functions and especially they are not able to support transparent sharing without application support.

Efforts in this direction include distributed *cluster file systems* often based on VAXclusters [Kronenberg et al., 1986] concepts that allow for efficient sharing of data among a set of storage servers with strong consistency semantics and fail-over capabilities. Such systems typically operate on top of a pool of physically shared devices through a SAN. However, they do not provide much control over the system's operation.

Modern cluster filesystems such as the Global File System (GFS) [Preslan et al., 1999] and the General Parallel File System (GPFS) [Schmuck and Haskin, 2002] are used extensively today in medium and large scale storage systems for clusters. However, their complexity makes them hard to develop and maintain, prohibits any practical extension to the underlying storage system, and forces all applications to use a single, almost fixed, view of the available data. In this chapter we examine how such issues can be addressed in future storage systems.

The complexity of cluster filesystems is often mitigated by means of a logical volume manager which virtualizes the storage space and allows transparent changes to the mappings between data and devices while the system is on-line. The two most advanced open-source volume managers currently are EVMS and GEOM. EVMS [EVMS] is a user-level distributed volume manager for Linux. It uses the MD [De Icaza et al., 1997] and device-mapper kernel modules to support user-level plug-ins. Recent versions offer persistent metadata and block re-mapping primitives to these plug-ins. However, EVMS does not support generic extensions (such as versioning or advanced reliability features), unlike Orchestra. GEOM [GEOM] is a single-node stackable block I/O subsystem under development for FreeBSD. The basic I/O stack concepts of GEOM are similar to Orchestra, however, GEOM does not support distributed hierarchies, volume sharing or persistent metadata which, combined with dynamic block mapping are necessary for advanced modules such as versioning, presented in Chapter 3.

Beyond open-source software, there exist numerous commercial virtualization solutions as well, such as HP OpenView Storage Node Manager [HP OpenView], EMC Enginuity [EMC Enginuity] and

Veritas Volume Manager [Veritas, c]. However, in all cases, the offered virtualization functions are predefined and there is no support for extending the I/O stack with new features.

In contrast, Orchestra, presented in Chapter 5, has all the configuration and flexibility features of a volume manager coupled with the ability to write extension modules with arbitrary virtualization semantics.

2.3.2 Flexible support for distributed storage

To overcome the aforementioned limitations of the traditional tools, a number of research projects have aimed at building scalable storage systems by defining more modular architectures and pushing functionality towards the block-level.

A popular approach is based on the concept of a *shared virtual disk* [Shillner and Felten, 1996; Lee and Thekkath, 1996; MacCormick et al., 2004], which handles most of the critical concerns in distributed storage systems, including fault-tolerance, dynamic addition or removal of physical resources, and sometimes (consistent) caching. In this way, the design of the filesystem can be significantly simplified. However, most of the complexity is pushed to the level of the virtual disk (assisted sometimes by a set of external services, e.g. for locking or consensus), whose design remains monolithic.

Our work on Orchestra bears similarity with this approach, and in particular with Petal-Frangipani [Lee and Thekkath, 1996; Thekkath et al., 1997] in that all filesystem communication happens through a distributed volume layer, simplifying filesystem design and implementation. However, contrary to Frangipani which uses an out-of-band lock server and allocates blocks through the FS, Orchestra performs locking as well as block allocation through the block layer. We believe that this in-band management of all the core functions is an important feature, which leaves more freedom to system designers regarding the hardware/software boundary of a storage system. Thus, different trade-offs can be explored more quickly because all the functionalities exported by the virtual volume are built from a stack of modules. Moreover, Orchestra allows storage systems to provide varying functionality through virtual hierarchies and also increases flexibility and control in distributing many storage layers to a number of (possibly cascaded) storage nodes.

Device-served locks have been envisioned in the past, notably in the context of GFS, which led to the specifications of the DMEP/DLOCK SCSI commands [Preslan et al., 2000]. However, DMEP/DLOCK capabilities never really became a reality due to the reluctance of the storage vendors to evolve the interface of their products. Unlike these past efforts, our propositions do not require any hardware

modification and enables a broader range of constructs (such as hierarchical locking) so as to allow more scalability.

Recent research has investigated more deeply the modular approach for building cluster storage systems, in particular with the Swarm [Hartman et al., 1999] and Abacus [Amiri et al., 2000] projects. However, Swarm does not enable data sharing among multiple (client) nodes. Besides, due to its log-based interface to storage devices, Swarm leaves little freedom regarding the placement of most modules: much of the functionality is pushed on the client side. Abacus has mostly focused on optimal, dynamic function placement through automatic migration of components. However, to the best of our knowledge, it has not addressed issues, such as functional extensibility and simplified metadata management.

Furthermore, a number of frameworks for extensible files systems have been proposed [Heidemann and Popek, 1994; Schermerhorn et al., 2001; Skinner and Wong, 1993; Zadok and Nieh, 2000]. However, these proposals have been limited to schemes relying on a central server, which are not well suited in data-centers, where scalability is a primary concern. While we strive to present a single system image to the clients, our work targets large storage systems based on a distributed (and more scalable) topology.

2.3.3 Support for cluster-based storage

A number of research projects have explored the potential of decentralized storage systems made of a large number of nodes with increased processing capabilities.

One of the pioneering efforts in this regard is based on Object-based Storage Devices (OSD). The OSD approach defines an object structure that is understood by the storage devices, and which may be implemented at various systems components, e.g. the storage controller or the disk itself. Our approach does not specify fixed groupings of blocks, i.e. objects. Instead, it allows virtual modules to use metadata and define block groupings dynamically, based on the module semantics. These groupings and associations may occur at any layer in a virtual storage hierarchy. For instance, a versioning virtual device (e.g. Clotho presented in Chapter 3) may be inserted either at the application server or storage node side and specifies through its metadata which blocks form each version of a specific device. In addition, our propositions do not necessarily require modifications of the current interface of storage devices.

In both approaches, allocation occurs closer to the block-level. In OSD, objects are allocated by the storage device, whereas in Orchestra by a virtual module that is inserted in the storage hierarchy.

In terms of locking, and to the best of our knowledge, the current OSD specification [Weber (Editor), 2004] does not state explicitly how mutual exclusion should be provided, however, object attributes may be used to implement mutual exclusion mechanisms on top of it. In Orchestra, similarly to allocation, locking is provided by a new virtual module that is inserted to the storage hierarchy on demand. OSD specifies that object devices perform protection checking, whereas in Orchestra we envision that, beyond traditional permissions in the filesystem, protection will also be provided by custom virtual modules at fine granularity.

Overall, Orchestra shares many goals with OSD in enriching the semantics of the block-level. However, our approach allows for more flexibility on how storage devices are "composed" from distributed physical disks and, as we explore in this chapter, facilitates simpler cluster filesystem design.

Ursa Minor [Abd-El-Malek et al., 2005], a system for object-based storage bricks coupled with a central manager, provides flexibility with respect to the data layout and the fault-model (both for client and storage nodes). These parameters can be adjusted dynamically on a per data item basis, according to the needs of a given environment. Such fine grain customization yields noticeable performance improvements. However, the system imposes a fixed architecture based on a central metadata server that could limit scalability and robustness, and is not extensible, i.e. the set of available functions and reconfigurations is pre-determined.

The Federated Array of Bricks (FAB) [Saito et al., 2004] discusses how storage systems may be built out of commodity storage nodes and interconnects and yet compete (in terms of reliability and performance) with custom, high-end solutions for enterprise environments. Overall, we share similar objectives. However, FAB has mostly investigated optimized protocols for consistent updates to replicas and quick background recovery using a non-extensible, out-of-band approach, while our main focus is to examine how future block-level systems can be tailored to support changing application needs without compromising transparency and scale.

Previous work has also investigated a number of issues raised by the lack of a central controller and the distributed nature of cluster-based storage systems, e.g. consistency for erasure-coded redundancy schemes [Amiri et al., 2000] and efficient request scheduling [Lumb et al., 2004]. We consider these concerns to be orthogonal to our work but we note that the existing solutions in these domains could be implemented as modules within our framework.

Finally, our design of the block allocator, described in Section 5.2.2, bears similarity with Hoard [Berger et al., 2000], a scalable memory allocator for multi-threaded applications. Both systems par-

tition the address space in regions, called “supernodes” by Hoard and “allocation zones” in Orchestra and can scale to many processes or nodes. However, since Hoard was designed for allocating memory, its operation is designed according to the memory hierarchy characteristics, such as cache misses. On the other hand, Orchestra’s allocator considers networked storage characteristics, such as the latency of locking regions, and data placement because of the performance difference between sequential and random access to disks.

2.3.4 Summary

Overall, current approaches have important limitations. First, block-level virtualization systems tend to provide simple semantics, and rely on higher-level layers to deal with consistent and structured data sharing. Moreover, cluster filesystems are complex and monolithic, which makes it very challenging to tune them for specific application needs. Recent work on cluster-based storage has mostly investigated issues related to reliability, security, and performance optimizations. However, to the best of our knowledge, techniques for building highly configurable storage systems from (distributed) block devices have received far less attention.

2.4 RIBD: Taxonomy and Related Work

In this section we present a taxonomy of existing and possible solutions for handling data replica and metadata consistency issues in distributed storage systems. We are interested in solutions that (1) assume no centralization point in the data path from clients to data and (2) deal with data replication for availability purposes. Then we discuss related work on RIBD.

2.4.1 Taxonomy of existing solutions

First, we categorize solutions based on two high-level parameters used for recovering the whole system to a consistent state after a failure: (a) recovering data replicas to a consistent state and (b) ensuring consistency of metadata and possibly data as well.

Data replica consistency: This problem can be addressed either by extending the block or the filesystem. In either case, it is necessary to ensure that read operations to different disk servers, will always return consistent block contents after a failure. This requires the file or block level to know after a failure

that may have resulted in updating only a subset of the replicas, which replica is the right one to use for subsequent requests. The first approach, file-level replication (FLR), requires the filesystem on the node performing the I/O to be aware of replica placement and management, eliminating the possibility of providing replication by various RAID options at the block level. The later, block-level replication (BLR) complicates replica updates, as it introduces dependencies among updates in different disk servers.

When replication happens at the block level, then it can apply to all blocks of a volume, both data and metadata. File-level replication can either refer to metadata only (FR-M) or both data and metadata (FR-MD), as in practice it does not make sense to only replicate data blocks.

Metadata consistency: This issue can be addressed either by logging operations and rolling forward (RF) after a failure by replaying logged operations or by maintaining multiple versions of data and rolling backwards (RB) to a previously consistent state of the system after a failure. Either approach can be implement at the file level (FC) or the block level (BC), leading to four options: FC-RF, FC-RB, BC-RF, BC-RB.

Finally, in this taxonomy, we do not distinguish between systems that log only metadata or both metadata and data. Both types of system would be able to recover to a consistent state after a failure.

Table 2.1 categorizes some existing systems based on this taxonomy. There are various observations we can make:

First, there is a number of systems [PVFS2; Nieuwejaar and Kotz, 1996; White et al., 2001; Menon et al., 2003] that focus on scalability and performance, leaving reliability and availability issues to other layers in the system. Typically, these systems assume that underlying storage devices do not fail or deal with failures themselves and that metadata consistency is either maintained by avoiding associated problems (e.g. with centralized, synchronous operations) or is provided by other system layers [Menon et al., 2003].

Second, most systems that deal with failures themselves provide metadata consistency by using file-level roll-forward techniques, either based on logs or shadow buffers. These systems vary mostly in their assumptions about replication of storage and fall in all possible categories: Systems that do not replicate storage assuming that the underlying devices are reliable [Sun Microsystems, 2007; Devarakonda et al., 1996], systems that replicate only file metadata [Preslan et al., 1999; Mitchell and Dion, 1982; Brown et al., 1985], systems that replicate both metadata and data either at the file level [Schmuck and Haskin, 2002; Hartman and Ousterhout, 1992; Liskov et al., 1991], or at the block level [Thekkath et al., 1997;

	NONE	FC-RF	FC-RB	BC-RF/RB
NONE	PVFS2 [PVFS2] Galley [Nieuwejaar and Kotz, 1996] ST [Menon et al., 2003] Legion [White et al., 2001]	GFS [Preslan et al., 1999] Lustre [Sun Microsystems, 2007] Calypso [Devarakonda et al., 1996]		
FR-M		XDFS [Mitchell and Dion, 1982] Alpine [Brown et al., 1985] CFS [Mitchell and Dion, 1982]		
FR-MD		GPFS [Schmuck and Haskin, 2002] Zebra [Hartman and Ousterhout, 1992] Harp [Liskov et al., 1991]	ZFS [Bonwick and Moore, 2008]	
BR	Kybos [Wong et al., 2005] FAB [Saito et al., 2004] Petal [Lee and Thekkath, 1996]	Frangipani [Thekkath et al., 1997] xFS [Anderson et al., 1996] Ceph [Weil et al., 2006]		Echo [Hisgen et al., 1993] BSTs [Amiri et al., 2000] RIBD

Table 2.1: Categorization of existing distributed file systems and storage systems depending on their replica consistency and metadata consistency mechanisms. Notation: (i) Vertical Categories: file-level replication (FR), block-level replication (BR), file-level, metadata only replication (FR-M), data and metadata replication (FR-MD), (ii) Horizontal Categories: Consistency at file level (FC), or block level consistency (BC). Mechanisms: roll-forward logging (RF), or roll-back versioning (RB), lead to four combinations: FC-RF, FC-RB, BC-RF, BC-RB.

Anderson et al., 1996; Weil et al., 2006][1].

Third, there have been fewer systems in the rest of the categories. This can be explained as follows: File-level roll-back recovery (FC-RB column) can be implemented generally in two ways: either by logging old values of metadata and data, e.g. in a journal, or by using in place versioning on the storage devices. The first approach, is not practical as it requires updating data on the disk and thus, performing seeks and storing the old values in the journal for roll back purposes and thus, incurring all I/Os in the critical path. Instead, it makes more sense to just use roll forward, by only writing the new values directly in the journal and applying them to the actual data later on, during a cleanup procedure, off the critical path. Thus, roll back approaches make sense only for systems that have support for low-cost versioning, such as ZFS [Bonwick and Moore, 2008][2].

Block level recovery on the other hand, requires maintaining state at the block level, which has been difficult to provide in previous generation systems. Block level, roll forward recovery (BC-RF column) has been examined in the Echo system [Hisgen et al., 1993]. Echo uses a transactional API at the block level, implemented with a logging mechanism, to resolve inconsistencies both for replication purposes as well as for system metadata. Dealing with these issues at the block level allows Echo to use a single log for both purposes. The BST approach [Amiri et al., 2000] introduces base storage transactions (BSTs) at the block level. By restricting their scope, it is possible to reduce the amount of state required and to avoid maintaining a log.

Finally, the shaded classes do not seem to make sense for practical, real-life systems, whereas the FC-RB/BR class is not populated, but it appears to be a valid design point. This would correspond e.g., to a versioning filesystem implemented appropriately on top of a block level such as Petal [Lee and Thekkath, 1996], FAB [Saito et al., 2004], or Kybos [Wong et al., 2005].

Given these observations, there are two main questions to consider:

1. Does it make sense to deal with consistency issues at the file or block level?

2. Is roll forward or roll back recovery more appropriate?

In Chapter 6 we argue that dealing with consistency issues at the block level by using roll back recovery is a promising approach for building storage systems.

[1] Ceph [Weil et al., 2006] is a filesystem based on object-based storage, nevertheless, relies on the object storage devices for replication of data and guarantees metadata consistency at the file level.

[2] Although ZFS is not a cluster filesystem we mention it here because it provides an interesting design point.

2.4.2 Related Work

Besides the taxonomy presented above, our work on RIBD, presented in Chapter 6, bears similarity with various storage systems in three respects:

First, in providing a transactional API at the block level, Echo [Hisgen et al., 1993] provides arbitrary transactions but uses roll forward recovery. BSTs [Amiri et al., 2000] allow the system to maintain consistency at all block operations, without the need for recovery after failures. Instead our API uses the notion of *consistency intervals* (CIs) that deal with atomicity, consistency, and durability, relying for isolation to the use of explicit locks within CIs. CIs can include arbitrary block operations. To maintain consistency, unlike Echo, we use roll back recovery based on versioning. Finally, unlike both Echo and BSTs we evaluate the impact on scalability using a real prototype. CIs are similar to *atomic recovery units* (ARUs) [Grimm et al., 1996]. However, ARUs do not deal with replica consistency, as they target recovery in local storage systems.

Second, with systems that aim at low-overhead versioning. Such systems that use versioning at the file level are ZFS [Bonwick and Moore, 2008], Elephant [Santry et al., 1999], 3DFS [Roome, 1992], the Inversion filesystem [Olson, 1993], Plan9 [Pike et al., 1995], Spiralog [Johnson and Laing, 1996], and self-securing storage [Strunk et al., 2000]. Also, Cedar [Hagmann, 1987] and Venti [Quinlan and Dorward, 2002] use a similar concept of immutable files. Finally, Soules et al. [Soules et al., 2003] examine issues related to metadata in versioning filesystems.

Systems that use versioning at the block level are Clotho (presented in Chapter 3) that is extended in Chapter 6 to support distributed globally-consistent versions, Petal [Lee and Thekkath, 1996], WAFL [Hitz et al., 1994], SnapMirror [Patterson et al., 2002], HP Olive [Aguilera et al., 2006] and Timeline [Hue Moh and Liskov, 2004].

WAFL and SnapMirror are single-node versioning systems. Petal offers crash-consistent snapshots using a copy-on-write mechanism but requires disruption of user access to stored data when capturing or accessing versions. Olive builds upon the FAB [Saito et al., 2004] clustered storage system, adding the ability to create writable storage branches incurring minimal overhead . However, Olive's snapshots are crash-consistent (rather than consistent at the higher level software level as in RIBD) and require a recovery mechanism at the application level (e.g. fsck), combined with write support, in order to be consistent. Finally, Timeline has a different scope supporting versioning for a persistent object store distributed over the wide area.

Third, our implementation of RIBD relies on Orchestra, an extensible storage system prototype presented in Chapter 5. Other systems that have used similar approaches for extending the functionality of the I/O stack are the Hurricane filesystem [Krieger and Stumm, 1997] and FiST [Zadok and Nieh, 2000]. Unlike Orchestra, that replaces the block level in the Linux kernel, these systems operate at the file level.

Besides *storage* systems, there has been a lot of work on issues related to failures and recovery of distributed systems. In particular, Sinfonia [Aguilera et al., 2007] provides mini-transactions among groups of nodes and has been used to build a cluster filesystem, SinfoniaFS, for evaluation purposes. Like Sinfonia, RIBD targets data centers, does not require interaction between file servers, and offers configurable durability semantics. There are nonetheless significant differences. First, Sinfonia proposes a relatively general-purpose transactional abstraction while our model is simpler and focuses on storage systems. In addition, Sinfonia follows a roll-forward recovery model relying on logging while our system implements a roll-backwards recovery scheme based on versioning. In addition SinfoniaFS explores a different design space by supporting coherent, client-side caching, with a write-through read-validation strategy. Our approach would also impose a write-through scheme for (optional) client-side caching at the end of a transaction. However, the use of explicit locking would allow client caches to avoid read validation.

The Federated Array of Bricks (FAB) [Saito et al., 2004] is also related to RIBD and is interesting in the respect that it uses a voting protocol to deal with all types of failures in replica consistency. Unlike CIs, FAB allows updates to replicas to complete when quorum is guaranteed. During reads, a voting process takes place and decides the value of the block. This approach incurs fairly different tradeoffs compared to our protocols (writes are faster and reads are slower). Supporting a voting mechanism for replica consistency of non critical data in our system would only require modifications to a small subset of modules. However, we choose to use a single, transactional mechanism for all purposes, as we believe it is more appropriate for primary storage applications, as opposed to archival-type applications.

Finally, RIBD does not address Byzantine failures and silent data errors, which have been the topic of systems such as PASIS [Goodson et al., 2003], as well as off-site disaster recovery [Patterson et al., 2002].

Chapter 3

Clotho: Transparent Data Versioning at the Block I/O Level

Those who cannot remember the past are condemned to repeat it.

—George Santayana (1863 - 1952), The Life of Reason

The content of this chapter roughly corresponds to publication [Flouris and Bilas, 2004].

3.1 Introduction

Storage is currently emerging as one of the major problems in building tomorrow's computing infrastructure. Future systems will provide tremendous storage, CPU processing, and network transfer capacity in a cost-efficient manner and they will be able to process and store ever increasing amounts of data. The cost of managing these large amounts of stored data becomes the dominant complexity and cost factor for building, using, and operating modern storage systems. Recent studies [Gartner Group, 2000] show that storage expenditures represent more than 50% of the typical server purchase price for applications such as OLTP (On-Line Transaction Processing) or ERP (Enterprise Resource Planning) and these percentages will keep growing. Furthermore, the cost of storage administration is estimated at several times the purchase price of the storage hardware [Anderson et al., 2002; Cox et al., 2002; Lee and Thekkath, 1996; Gibson and Wilkes, 1996; Thekkath et al., 1997; Veitch et al., 2001; Wilkes, 2001]. Thus, building self-managed storage devices that reduce management-related overheads and complexity is of paramount importance.

One of the most cumbersome management tasks that requires human intervention is creating, maintaining, and recovering previous versions of data for archival, durability, and other reasons. The problem is exacerbated as the capacity and scale of storage systems increases. In large-scale storage systems, proper backup is of paramount importance, however current solutions are expensive and arduous. Today, backup is the main mechanism used to serve these needs. However, traditional backup systems are limited in the functionality they provide. Moreover they usually incur high access and restore overheads on magnetic tapes, they impose a very coarse granularity in the allowable archival periods, usually at least one day, and they result in significant management overheads [Cox et al., 2002; Quinlan and Dorward, 2002]. Automatic versioning, in conjunction with increasing disk capacities, has been proposed [Cox et al., 2002; Quinlan and Dorward, 2002] as a method to address these issues. In particular, magnetic disks are becoming cheaper and larger and it is projected that disk storage will soon be as competitive as tape storage [Cox et al., 2002; Esener et al., 1999]. With the advent of inexpensive high-capacity disks, we can perform continuous, real-time versioning and we can maintain online repositories of archived data. Moreover, *online storage versioning* offers a new range of possibilities compared to simply recovering users' files that are available today only in expensive, high-end storage systems:

- **Recovery from user mistakes.** Online versioning allows frequent storage snapshots that reduce the amount of data lost compared to traditional backup systems. The users themselves can recover accidentally deleted or modified data by rolling-back to a saved version.

- **Recovery from system corruption.** In the event of a malicious incident on a system, administrators can quickly identify corrupted data as well as recover to a previous, consistent system state [Soules et al., 2003; Strunk et al., 2000].

- **Historical analysis of data modifications.** When it is necessary to understand how a piece of data has reached a certain state, versioning proves a valuable tool.

Our goal in the work presented in this chapter is to provide online storage versioning capabilities in commodity storage systems, in a *transparent* and *cost-effective* manner. Storage versioning has been previously proposed and examined purely at the filesystem level [Pike et al., 1990; Santry et al., 1999] or at the block level [Hitz et al., 1994; Sun Microsystems, 2008] but being filesystem aware. These approaches to versioning were intended for large, centralized storage servers or appliances. We argue that to build self-managed storage systems, versioning functionality should be pushed to lower system layers, closer to the disk to offload higher system layers [Strunk et al., 2000]. This is made possible by

underlying technologies that drive storage systems. Disk storage, network bandwidth, processor speed, and main memory are reaching speeds and capacities that make it possible to build cost-effective storage systems with significant processing capabilities [Esener et al., 1999; Gibson et al., 1998; Gray, 1999; Patterson, 2000] that will be able to both store vast amounts of information [Gray, 1999; Lesk, 1997] and to provide advanced functionality.

Our approach of providing online storage versioning is to provide all related functionality at the block level. This approach has a number of advantages compared to other approaches that try to provide the same features either at the application or the filesystem level. First, it provides a higher level of transparency and in particular is completely filesystem agnostic. For instance, we have used our versioned volumes with multiple, third party, filesystems without the need for any modifications. Data snapshots can be taken on demand and previous versions can be accessed online simultaneously with the current version. Second, it reduces complexity in higher layers of storage systems, namely the filesystem and storage management applications [Hutchinson et al., 1999]. Third, it takes advantage of the increased processing capabilities and memory sizes of active storage nodes and offloads expensive host-processing overheads to the disk subsystem, thus, increasing the scalability of a storage archival system [Hutchinson et al., 1999].

However, block-level versioning poses certain challenges as well: (i) Memory and disk space overhead: Because we only have access to blocks of information, depending on application data access patterns, there is increased danger for higher space overhead in storing previous versions of data and the related metadata. (ii) I/O path performance overhead: It is not clear at what cost versioning functionality can be provided at the block-level. (iii) Consistency of the versioned data when the versioned volume is used in conjunction with a filesystem. (iv) Versioning granularity: Since versioning occurs at a lower system layer, information about the content of data is not available, as is, for instance, the case when versioning is implemented in the filesystem or the application level. Thus, we only have access to full volumes as opposed to individual files.

We design *C*lotho[1], a system that provides versioning at the block-level and addresses all above issues, demonstrating that this can be done at minimal space and performance overheads. First, Clotho has low memory space overhead and uses a novel method to avoid copy-on-write costs when the versioning extent size is larger than the block size. Furthermore, Clotho employs off-line *differential compres-*

[1]Clotho, one of the Fates in ancient Greek mythology, spins the thread of life for every mortal.

sion (or diffing) to reduce disk space overhead for archived versions. Second, using advanced disk management algorithms, Clotho's operation is reduced in all cases to simply manipulating pointers in in-memory data structures. Thus, Clotho's common-path overhead follows the rapidly increasing processor-memory curve and does not depend on the much lower disk speeds. Third, Clotho deals with version consistency by providing mechanisms that can be used by higher system layers to guarantee that either all data is consistent or to mark which data (files) are not. Finally, we believe that volumes are an appropriate granularity for versioning policies. Given the amounts of information that will need to be managed in the future, specifying volume-wide policies and placing files on volumes with the appropriate properties, will result in more efficient data management.

We implement Clotho as an additional layer (driver) in the I/O hierarchy of Linux. Our implementation approach allows Clotho the flexibility to be inserted in many different points in the block layer hierarchy in a single machine, a clustered I/O system, or a SAN. Clotho works over simple block devices such as a standard disk driver or more advanced device drivers such as volume managers or hardware/software RAIDs. Furthermore, our implementation provides to higher layers the abstraction of a standard block device and thus, can be used by other disk drivers, volume/storage managers, object stores or filesystems.

The main memory overhead of Clotho for metadata is about 500 KBytes per GByte of disk space using 32-KByte extents (i.e. 500MB per TByte of disk). This is a significant amount of memory, however, it can be reduced using larger extents (e.g. with 128-KByte extents Clotho needs 128MB per TByte of disk). Further reduction of memory overhead can be achieved by swapping unused metadata from memory to disk, as discussed in the next chapters.

We evaluate our implementation with both micro-benchmarks as well as real filesystems and the SPEC SFS 3.0 suite over NFS. We find that the performance overhead of Clotho for I/O operations is minimal, however, it may change the behavior of higher layers (including the filesystem), especially if they make implicit assumptions about the placement of data blocks on the underlying block device. Clotho uses remap-on-write with a logging policy, which optimizes write performance, but in the long run may have adverse effects on the performance of reads. To avoid such behavior, co-design or tuning of the two layers (file system and Clotho) may be necessary. Overall, we find that our approach is promising in offloading significant management overhead and complexity from higher system layers to the disk itself and is a concrete step towards building self-managed storage systems.

The rest of this chapter is organized as follows. Section 3.2 presents our design and discusses the

related challenges in building block-level versioning systems. Section 3.3 presents our implementation. Section 3.4 presents our experimental evaluation and results, while Section 3.5 presents limitations and future work. Finally, Section 3.6 draws our conclusions.

3.2 System Design

The design of Clotho is driven by the following high-level goals and challenges:

- Flexibility and transparency.
- Low metadata footprint and low disk space overhead.
- Low-overhead common I/O path operation.
- Consistent online snapshots.

Next we discuss how we address each of these challenges separately.

3.2.1 Flexibility and Transparency

Clotho provides versioned volumes to higher system layers. These volumes look similar to ordinary physical disks that can, however, be customized, based on user-defined policies to keep previous versions of the data they store. Essentially, Clotho provides a set of mechanisms that allow the user to add time as a dimension in managing data by creating and manipulating volume versions. Every piece of data passing through Clotho is indexed based not only on its location on the block device, but also on the time the block was written. When a new version is created, a subsequent write to a block will create a new block preserving the previous version. Multiple writes to the same data block between versions result in overwriting the same block. Using Clotho, device versions can be captured either on demand or automatically at specified intervals. The user can view and access all previous versions of the data online, as independent block devices along with the current version. The user can compact and/or delete previous volume versions. In this chapter we focus on the mechanisms Clotho provides and we only present simple policies we have implemented and tested ourselves. We expect that systems administrators will further define their own policies in the context of higher-level storage management tools.

Clotho provides a set of primitives (mechanisms) that higher-level policies can use for automatic version management:

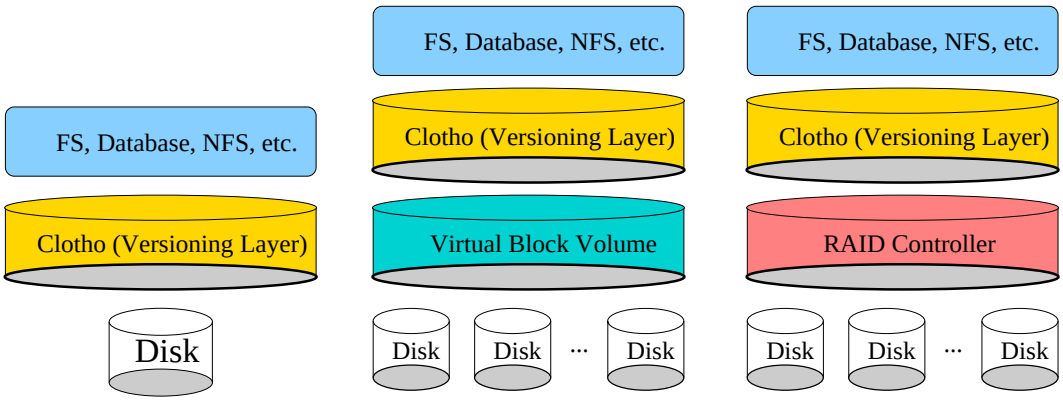


Figure 3.1: Clotho in the block device hierarchy.

- `CreateVersion()` provides a mechanism for capturing the lower-level block device's state into a version. This operation can be triggered either explicitly by OS services or user applications or optionally by Clotho based on internal events, such as on each write or every few writes.
- `DeleteVersion()` explicitly removes a previously archived version and reclaims the corresponding volume space.
- `ListVersions()` shows all saved version of a specific block device.
- `ViewVersion()` enables creating a virtual device that corresponds to a specific version of the volume and is accessible in read-only mode.
- `CompactVersion()` and `UncompactVersion()` provide the ability to compact and un-compact existing versions for reducing disk space overhead.

Versions of a volume have the following properties: Each version is identified by a unique *version number*, which is an integer counter starting from value 0 and increasing with each new version. Version numbers are associated with timestamps for presentation purposes. All blocks of the device that are accessible to higher layers during a period of time will be part of the version of the volume taken at that moment (if any) and will be identified by the same version number. Each of the archived versions exists solely in read-only state and will be presented to the higher levels of the block I/O hierarchy as a distinct, virtual, read-only block device. The latest version of a device is both readable and writable, exists through the entire lifetime of the Clotho's operation, and cannot be deleted.

One of the main challenges in Clotho is to provide all versioning mechanisms at the block level in a transparent and flexible manner. Clotho can be inserted arbitrarily in a system's layered block I/O

hierarchy. This stackable driver concept has been employed to design other block-level I/O abstractions, such as software RAID systems or volume managers, in a clean and flexible manner [Sun Microsystems, 2008]. The *input* (higher) layer can be any filesystem or other block-level abstraction or application, such as a RAID, volume manager, or another storage system. Clotho accepts block I/O requests (read, write, ioctl) from this layer. Similarly, the *output* (lower) layer can be any other block device or block-level abstraction. This design provides great flexibility in configuring a system's block device hierarchy. Figure 3.1 shows some possible configurations for Clotho. On the left part of Figure 3.1, Clotho operates on top of a physical disk device. In the middle, Clotho acts as a wrapper of a single virtual volume constructed by a volume manager, which abstracts multiple physical disks. In this configuration Clotho captures versions of the whole virtual volume. On the right side of Figure 3.1, Clotho is layered on top of a RAID controller which adds reliability to the system. The result is a storage volume that is both versioned and can tolerate disk failures.

Most higher level abstractions that are built on top of existing block devices (e.g. filesystems) assume a device of fixed size, with few rare exceptions such as resizable filesystems. However, the space occupied by previous versions of data in Clotho changes dynamically, depending on the volume of modified data and the frequency of data versioning. Clotho can provide both a fixed size block device abstraction to higher layers, as well as dynamically resizable devices, if the higher layers support it. At device initialization time Clotho reserves a configurable percentage of the available device space for keeping previous versions of the data. This essentially partitions (logically not physically) the capacity of the wrapped device into two logical segments as illustrated in Figure 3.2. The *Primary Data Segment* (PDS), which contains the data of the current (latest) version and the *Backup Data Segment* (BDS), which contains all the data of the archived versions. When BDS becomes full, Clotho simply returns an appropriate error code and the user has to reclaim parts of the BDS by deleting or compacting previous versions, or by moving them to some other device. These operations can also be performed automatically by a module that implements high-level data management policies. The latest version of the block device continues to be available and usable at all times. Clotho enforces this capacity segmentation by reporting as its total size to the input layer, only the size of the PDS. The space reserved for storing versions is hidden from the input layer and is accessed and managed only through the API provided by Clotho.

Finally, Clotho's metadata needs to be saved on the output device along with the actual data. Losing indexing metadata would render the data stored throughout the block I/O hierarchy unusable. Currently

Clotho lazily flushes metadata to the output device periodically in order to keep them consistent. Using this scheme, Clotho is able to recover after a crash to the latest saved volume version (i.e. roll-back) doing a full scan over the metadata which are stored sequentially on the disk. It is not, however, able to recover to the latest version of the data volume. Block-level metadata management and consistency in the case of crashes are important problems and hard to address without performance overheads. While Clotho does not currently implement sophisticated metadata consistency mechanisms in order to recover the latest data version, such issues and solutions are discussed in more detail in Section 4.2.3. The size of the metadata depends on the size of the encapsulated device and the extent size. In general, Clotho's metadata are much less than the metadata of a typical filesystem and they are stored sequentially on the disk. Thus, lazily synchronizing them to stable storage, or reading them to recover does not introduce large overheads.

3.2.2 Reducing Metadata Footprint

The three main types of metadata in Clotho are the *Logical Extent Table* (LXT), the *Device Version List* (DVL), and the *Device Superblock* (DSB).

The *Logical Extent Table (LXT)* is a structure used for logical to physical block translation. Clotho presents to the input layer *logical* block numbers as opposed to the *physical* block numbers provided by the wrapped device. Note that these block numbers need not directly correspond to actual physical locations, if another block I/O abstraction, such as a volume manager (e.g. LVM [Teigland and Mauelshagen, 2001]) is used as the output layer. Clotho uses the LXT to translate logical block numbers to physical block numbers.

The *Device Version List (DVL)* is a list of all versions of the output device that are available to higher layers as separate block devices. For every existing version, it stores its version number, the virtual device it may be linked to, the version creation timestamp, and a number of flags.

The *Device Superblock (DSB)* is a small table containing important attributes of the output versioned device. It stores information about the capacity of the input and output device, the space partitioning, the size of the extents, the sector and block size, the current version counter, the number of existing versions and other usage counters.

The LXT is the most demanding type of metadata and is conceptually an array indexed by block numbers. The basic block size for most block devices varies between 512 Bytes (the size of a disk sector) and 8 KBytes. This results in large memory requirements. For instance, for 1 TByte of disk

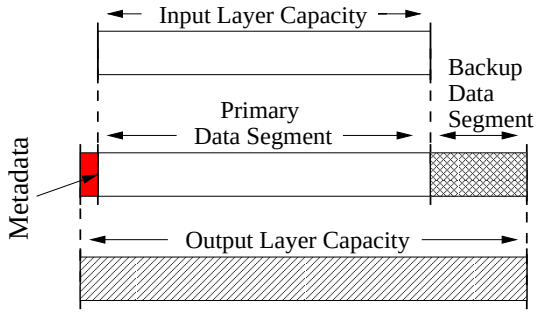


Figure 3.2: Logical space segments in Clotho.

storage with 4-KByte blocks, the LXT has 256M entries. In the current version of Clotho, every LXT entry is 128-bits (16 bytes). These include 32 bits for block addressing and 32 bits for versions that allow for a practically unlimited number of versions. Thus, such an LXT requires about 4 GBytes per TByte of disk storage. Note that a 32-bit address space, with 4 KByte blocks, can address 16 TBytes of storage.

To reduce the footprint of the LXT and at the same time increase the addressing range of LXT, we use *extents* as opposed to device blocks as our basic data unit. An extent is a set of consecutive (logical and physical) blocks. Extents can be thought of as Clotho's internal blocks, which one can configure to arbitrary sizes, up to several hundred KBytes or a few MBytes. Similarly to physical and logical blocks, we denote extents as logical (input) extents or physical (output) extents. We have implemented and tested extent sizes ranging from 1 KByte to 64 KBytes. With 32-KByte extents and subextent addressing, we need only 500 MBytes of memory per TByte of storage. Moreover with a 32-KByte extent size we can address 128 TBytes of storage.

However, using large extent sizes may result in significant performance overhead. When the extent size and the operating system block size for Clotho block devices are the same (e.g. 4KBytes), Clotho receives from the operating system the full extent for which it has to create a full version. When using extents larger than this maximum size, Clotho sees only a subset of the extent for which it needs to create a new version. Thus, it needs to copy the rest of the extent in the new version, even though only a small portion of it is written by the higher system layers. This copy-on-write operation can significantly decrease performance in the common I/O path, especially for large extent sizes. However, large extents are desirable for reducing metadata footprint. Given that operating systems support I/O blocks of up to a maximum size (e.g. 4K in Linux), this may result in severe performance overheads.

To address this problem we use *subextent addressing*, which enables remap-on-write operation on

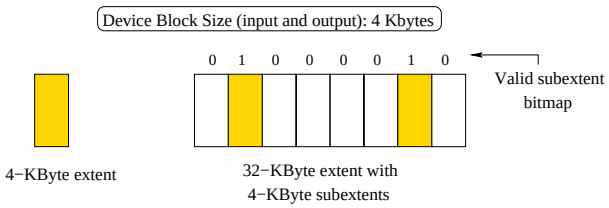


Figure 3.3: Subextent addressing in large extents.

every write. Remap-on-write does not copy the old data to a new extent, but allocates a new, free extent for the data and translates the logical extent there. To solve the problem of partial updates we use small bitmap in each LXT entry, marking with one bit the valid subextents, that is the ones containing valid data. The invalid subextents are considered to contain garbage.

Remap-on-write, enabled by subextent addressing, has much lower overhead than copy-on-write in write operations since it avoids data copying. However, subextent addressing has a significant implication: it changes the way read operations proceed. During a read, we need to search for the valid subextents in the LXT, both in the current but *also in older extents*, before translating the read operation to physical extents. Depending on the size of the read request this may lead to more than one reads to the disk. According to our experience and testing with real workloads and file systems this occurs rarely. However, if this is not acceptable for an application, Clotho can also operate using a copy-on-write algorithm which may exhibit better behavior in read-intensive workloads.

Another possible approach to reduce memory footprint is to store only part of the metadata in main memory and perform swapping of active metadata from stable storage. This solution is orthogonal to subextent addressing and can be combined with it. Active metadata swapping is necessary for large-scale storage systems where storage capacity is so large, that even with subextent addressing its footprint exceeds available memory. Support for such functionality in Clotho is left for future work.

3.2.3 Version Management Overhead

All version management operations can be performed at a negligible cost by manipulating in-memory data structures. Creating a new version in Clotho involves simply incrementing the current version counter and does not involve copying any data. When `CreateVersion()` is called, Clotho stalls all incoming I/O requests for the time required to flush all its outstanding writes to the output layer. When everything is synchronized on stable storage, Clotho increases the current version counter, appends a new entry to the device version list, and creates a new virtual block device that can be used to access

the captured version of the output device, as explained later. Since each version is linked to exactly one virtual device, the (OS-specific) device number that sends the I/O request can be used to retrieve the I/O request's version.

The fact that device versioning is a low-overhead operation makes it possible to create flexible versioning policies. Versions can be created by external processes periodically or based on system events. For instance, the user processes can specify that it requires a new version every 1 hour, or whenever all files to the device are closed or on every single write to the device. Some of the mechanisms to detect such events, e.g. if there are any open files on a device, may be (and currently are) implemented in Clotho but could also be provided by other system components.

In order to free backup disk space, Clotho provides a mechanism to delete volume versions. On a `DeleteVersion()` operation, Clotho first locates the stored volume version that immediately follows (time-wise) the delete candidate. Let us assume that the delete candidate is version numbered v, while the "next-in-time" version is v'. Clotho then traverses the primary LXT segment and follows the list of older versions for every extent. For every list entry that has a version number equal to v (i.e. the delete candidate), it checks whether there is also a list entry with the v' version number. If not, it changes the version number of the v entry to v', because version v' needs this extent and it cannot be deleted. If yes, it deletes the extent with version number v from the version list, marks the LXT entry as free, and frees the related physical extents. As with version creation, all above operations for version deletion are performed in-memory and may overlap with regular I/O.

`DeleteVersion()` is provided to the higher layers in order to implement *version cleaning policies*. Since storage space is finite, such policies are necessary in order to continue versioning without running out of backup storage. The `DeleteVersion()` operation is also necessary to proceed writing when both the primary (PDS) and backup data segments (BDS) are full. In this case, Clotho automatically deletes the oldest existing volume version, in order for to continue writing to the latest version of data.

3.2.4 Common I/O Path Overhead

We consider the common path for Clotho, as the I/O path to read and write to the latest (current) version of the output block device, while versioning occurs frequently. Accesses to older versions are of less importance since they are not expected to occur as frequently as current version usage. Accordingly, we divide read and write access to volume versions in two categories, accesses to the current version and accesses to any previous version. The main technique to reduce common path overhead is to divide

the LXT in two logical segments, corresponding to the primary and backup data segments of the output device as illustrated in Figure 3.2. The primary segment of the LXT (mentioned as PLX in figures) has an equal number of logical extents as the input layer to allow a direct, 1-1 mapping between the logical extents and the physical extents of the current version on the output device. By using a direct, 1-1 mapping, Clotho can locate a physical extent of the *current version* of a data block with a *single lookup* in the primary LXT segment, when translating I/O requests to the current version of the versioned device. If the input device needs to access previous versions of a versioned output device, then multiple accesses to the LXT maybe required to locate the appropriate version of the requested extent.

To find the physical extent that holds the specific version of the requested block, Clotho first references the primary LXT segment entry to locate the current version of the requested extent (a single table access). Then it uses the linked list that represents the version history of the extent to locate the appropriate version of the requested block. Depending on the type of each I/O request and the state of the requested block, I/O requests can be categorized as follows:
Write requests can only be performed on the current version of a device, since older versions are read-only. Thus, Clotho can locate the LXT entry of a current version extent with a *single LXT access*. Write requests can be one of three kinds as shown in Figure 3.4:

a. Writes to new, unallocated blocks. In this case, Clotho calls its extent allocator module, which returns an available physical extent of the output device, it updates the corresponding entry in the LXT, and forwards the write operation to the output device. The default *extent allocation policy* in our current implementation is a scan-type (or logging) policy, starting from the beginning of the PDS to its end. Free extents are ignored until we reach the end of the device, when we rewind the allocation pointer and start allocating the free extents. We have also implemented other allocation policies, such as first free block and logging with immediate reuse on free.

b. Writes to existing blocks that have been modified after the last snapshot was captured (i.e. their version number is equal to the current version number). In this case Clotho locates the corresponding entry in the primary LXT segment with a single lookup and translates the request's block address to the existing physical block number of the output device. Note that in this case the blocks are *updated in place*.

c. Writes to existing blocks that have not been modified since the last snapshot was captured (i.e. their version number is lower than the current version number). The data in the existing physical

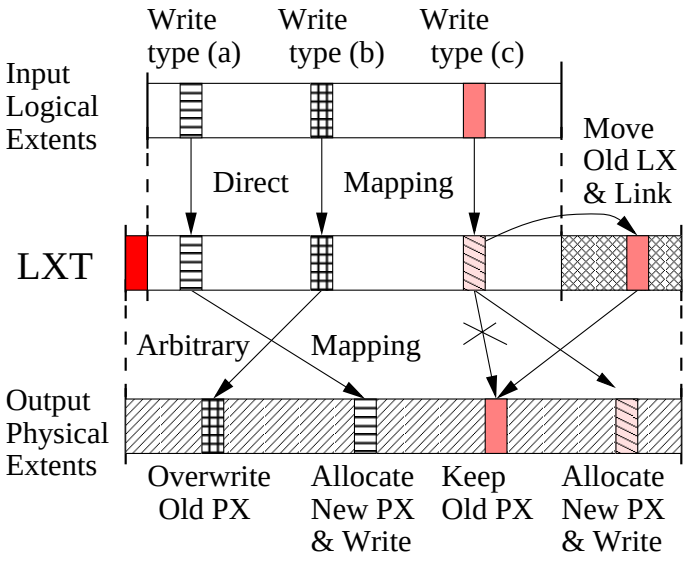


Figure 3.4: Translation path for write requests.

extent must not be overwritten, but instead the new data should be written in a different location and a new version of the extent must be created. Clotho allocates a new LXT entry *in the backup segment* and *swaps* the old and new LXT entries so that the old one is moved to the backup LXT segment. The block address is then translated to the new physical extent address, and the request is forwarded to the output layer. This "swapping" of LXT entries maintains the 1-1 mapping of current version logical extents in the LXT which optimizes common-path references to a single LXT lookup.

This write translation algorithm allows for independent, fine grain versioning at the extent level. Every extent in the LXT is versioned according to its updates from the input level. Extents that are updated more often have more versions than extents written less frequently.

Read request translation is illustrated in Figure 3.5. First Clotho determines the desired version of the device by the virtual device name and number in the request (e.g. /dev/clt1-01 corresponds to version 1 and /dev/clt1-02 to version 2). Then, Clotho traverses the version list on the LXT for the specific valid extent and subextent and locates the appropriate physical block. Note that, as mentioned in Section 3.2.2, due to remap-on-write and subextent addressing, it may be necessary to search for the valid subextents in the LXT, both in the current but *also in older extents*, before translating the read operation to physical extents. Depending on the size of the read request and the placement of valid subextents in the versioned extents, an application read may lead to more than one read operations to the disk. According to our

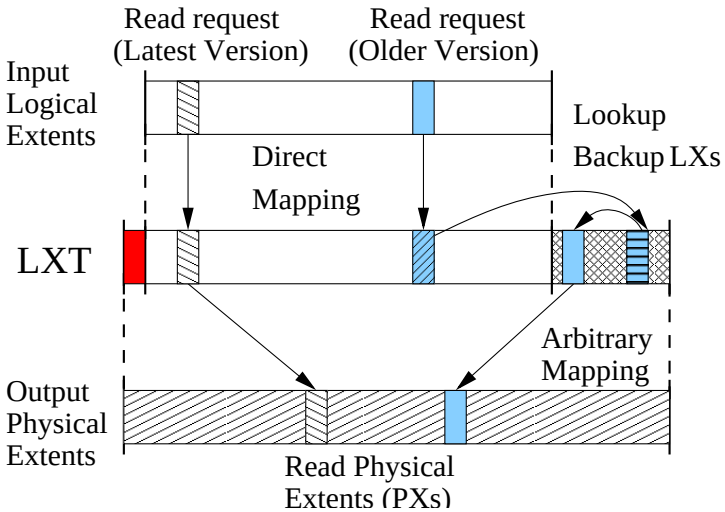


Figure 3.5: Translation path for read requests.

experiments, however, this occurs infrequently and we found that remap-on-write is a better choice that copy-on-write in the workloads we have measured.

Finally, previous versions of a Clotho device appear as different virtual block devices. Higher layers, e.g. filesystems, can use these devices to access old versions of the data. If the device ID of a read request is different from the normal input layer's device ID, the read request refers to an extent belonging to a previous version. Clotho determines from the device ID the version of the extent requested. Then, it traverses the version list associated with this extent to locate the backup LXT entry that holds the appropriate version of the logical extent. This translation process is illustrated in Figure 3.5.

3.2.5 Reducing Disk Space Requirements

Since Clotho operates at the block level, there is an induced overhead in the amount of space it needs to store data updates. For instance, if an application using a file modifies a few consecutive bytes in the file, Clotho will create a new version for the full extent that contains the modified data. To reduce the space overhead in Clotho we provide a differential, content-based compaction mechanism, which we describe next.

Clotho provides the user with the ability to compact device versions and still be able to transparently access them online. The policy decision on when to compact a version is left to higher-layers in the system, similarly to all policy decisions in Clotho. We use a form of binary differential compression [Ajtai et al., 2002] to only store the data that has been modified since the last version capture. When

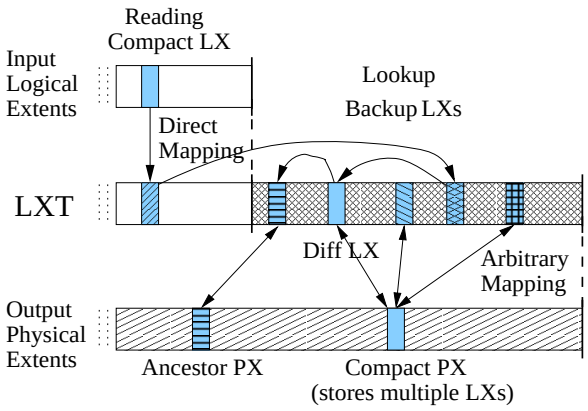


Figure 3.6: Read translation for compact versions.

`CompactVersion()` is called, Clotho constructs a differential encoding (or delta) between the extents that belong to a given version with corresponding extents in its previous version [Ajtai et al., 2002]. Although a lot of differential policies can be applied in this case, such as to compare the content of a specific version with its next version, or both the previous and the next version, at this stage we only explore diffing with the previous version. Furthermore, although versions can also be compressed independently of differential compression using algorithms such as Lempel-Ziv encoding [Ziv and Lempel, 1977] or Wheeler-Burrows encoding [Burrows and Wheeler, 1994], this is beyond the scope of our work. We envision that such functionality can be provided by other layers in the I/O device stack.

The differential encoding algorithm works as follows. When a compaction operation is triggered, the algorithm runs through the backup data segment of the LXT and locates the extents that belong to the version under consideration. If an extent does not have a previous version, it is not compacted. For each of the extents to be compacted the algorithm locates its previous version, diffs the two extents, and writes the diffs to a physical extent on the output device. If the diff size is greater than a threshold and diffing is not very effective, then Clotho discards this pair and proceeds with the next extent of the version to be compacted. In other words, Clotho's differential compression algorithm works selectively on the physical extents, compacting only the extents that can be reduced in size. The rest are left in their normal format to avoid performance penalties necessary for their reconstruction.

Since the compacted form of an extent requires less size than a whole physical extent, the algorithm stores multiple deltas in the same physical extent, effectively imposing a different structure on the output block device. Furthermore, for compacted versions, multiple entries in the LXT may point to the same physical extent. The related entries in the LXT and the ancestor extent are kept in Clotho's metadata.

Physical extents that are freed after compaction are reused for storage. Figure 3.6 shows sample LXT mappings for a compacted version of the output layer.

Data on a compacted version can be accessed transparently online as data on non-compacted volumes (Figure 3.6). Clotho follows the same path to locate the appropriate version of the logical extent in the LXT. To recreate the original, full extent data we need the differential data of the previous version of the logical extent. With this information Clotho can reconstruct the requested block and return it to the input driver. We evaluate the related overheads in Section 3.4.

Clotho supports recursive compaction of devices. The next version of a compacted version can still be compacted. Also, compacted versions can be un-compacted to their original state with the reverse process. A side-effect of the differential encoding concept is that it creates dependencies between two consecutive versions of a logical extent, which affects the way versions are accessed, as explained next.

When deleting versions, Clotho checks for dependencies of compacted versions on a previous version and does not delete extents that are required for un-diffing, even if their versions are deleted. These logical extents are marked as `"shadow"` and are attached to the compacted version. It is left to higher-level policies to decide if keeping such blocks increases the space overhead and it would be better to un-compact the related version and delete any shadow logical extents.

3.2.6 Consistency

One of the main issues in block device versioning at arbitrary times is consistency of the stored data. There are three levels of consistency for online versioning:

System state consistency: This refers to consistency of system buffers and data structures that are used in the I/O path. To deal with this, Clotho flushes all device buffers in the kernel as well as filesystem metadata before creating a volume version. Additionally, modern filesystems, such as XFS, provide user-level support for filesystem “freezing”, where data and metadata flushing occurs while the filesystem is “frozen”. This guarantees that the data and metadata on the block device correspond to a valid snapshot of the filesystem at a point-in-time. That is, there are no consistency issues in internal system data structures. Regarding Clotho’s metadata, they are also synchronized to disk at version creation. This allows safe rollback to a previous version in the case of a crash, as discussed in Section 3.2.1.

Open file consistency: When a filesystem is used on top of a versioned device, certain files may be open at the time of a snapshot. Since Clotho is designed as a transparent block device, it cannot deal with this issue because of the block-level API limitations. However, in our Linux implementation

we have access to higher layers and we provide a mechanism to assist users. When a new version is created, Clotho's user-level module queries the system for open files on the particular device. If such files are open, Clotho creates a special directory with links to all open files and includes the directory in the archived version. Thus, when accessing older versions the user can find out which files were open at versioning time.

Application consistency: Applications using the versioned volume may have a specialized notion of consistency. For instance, an application may be using two files that are both updated atomically. If a version is created after the first file is updated and closed but before the second one is open and updated, then, although no files were open during version creation, the application data may still be inconsistent. This type of consistency is not possible to deal with transparently without application knowledge or support, and thus, is not addressed by Clotho.

3.3 System Implementation

We have implemented Clotho as a block device driver module in the Linux 2.4 kernel and a user-level control utility, in about 6,000 lines of C code. The kernel module can be loaded at runtime and configured for any output layer device by means of an `ioctl()` command triggered by the user-level agent. After configuring the output layer device, the user can manipulate the Clotho block device depending on the higher layer that they want to use. For instance, the user can build a filesystem on top of a Clotho device with `mkfs` or `newfs` and then `mount` it as a regular filesystem.

Our module adheres to the framework of block I/O devices in the Linux kernel and provides two interfaces to user programs: an `ioctl` command interface and a `/proc` interface for device information and statistics. All operations described in the design section to create, delete, and manage versions have been implemented through the `ioctl` interface and are initiated by the user-level agent. The `/proc` interface provides information about each device version through readable ASCII files. Clotho also uses this interface to report a number of statistics, including the times of creation, a version's time span, the size of modified data from the previous version and some specific information to compacted versions, such as the compaction level and the number of *shadow* extents.

The Clotho module uses the zero-copy mechanism of the `make_request_fn()` fashion that is used by LVM [Teigland and Mauelshagen, 2001]. With this mechanism Clotho is able to translate the device driver ID `(kdev_t)` and the sector address of a block request `(struct buffer_head)` and redirect it

to other devices with minimal overhead. To achieve persistence of metadata, Clotho uses a kernel thread created at module load time, which flushes the metadata to the output layer at configurable (currently 30 second) intervals.

The virtual device creation uses the partitionable block device concepts in the Linux kernel. A limit in the Linux kernel minor numbering is that there can be at most 255 minor numbers for a specific device and thus, only 255 versions can be seen simultaneously as partitions of Clotho. However, the number of partitions supported by Clotho can be much larger. To overcome this limitation we have created a mechanism through an `ioctl` call that allows the user to link and unlink on demand any of the available versions to any of the 255 minor number partitions of a Clotho device. As mentioned, each of these partitions is read-only and can be used as a normal block device, e.g. can be mounted to a mount-point.

3.4 Experimental Results

Our experimental environment consists of two identical PCs running Linux. Each system has two Pentium III 866 MHz CPUs, 768 MBytes of RAM, an IBM-DTLA-307045 ATA Hard Disk Drive with a capacity of 46116 MBytes (2-MByte cache), and a 100MBps Ethernet NIC. The operating system is Red Hat Linux 7.1, with the 2.4.18 SMP kernel. All experiments are performed on a 21-GByte partition of the IBM disk. With a 32-KByte extent, we need only 10.5 MBytes of memory for our 21-GByte partition.

We evaluate Clotho with respect to two parameters: memory and performance overhead. We use two extent sizes, 4 and 32 KBytes. Smaller extent sizes have higher memory requirements. For our 21-GByte partition, Clotho with 4-KByte extent size uses 82 MBytes of in-memory metadata, the dirty parts of which are flushed to disk every 30 seconds. We evaluate Clotho using both micro-benchmarks (Bonnie++ version 1.02 [Coker] and an in-house developed micro-benchmark) and real-life setups with production-level filesystems. The Bonnie++ benchmark is a publicly available filesystem I/O benchmark [Coker]. For the real-life setup we run the SPEC SFS V3.0 suite on top of two well-known Linux filesystems, Ext2FS, and the high-performance journaling ReiserFS [ReiserFS]. In our results we use the label *Disk* to denote experiments with the regular disk, without the Clotho driver on top.

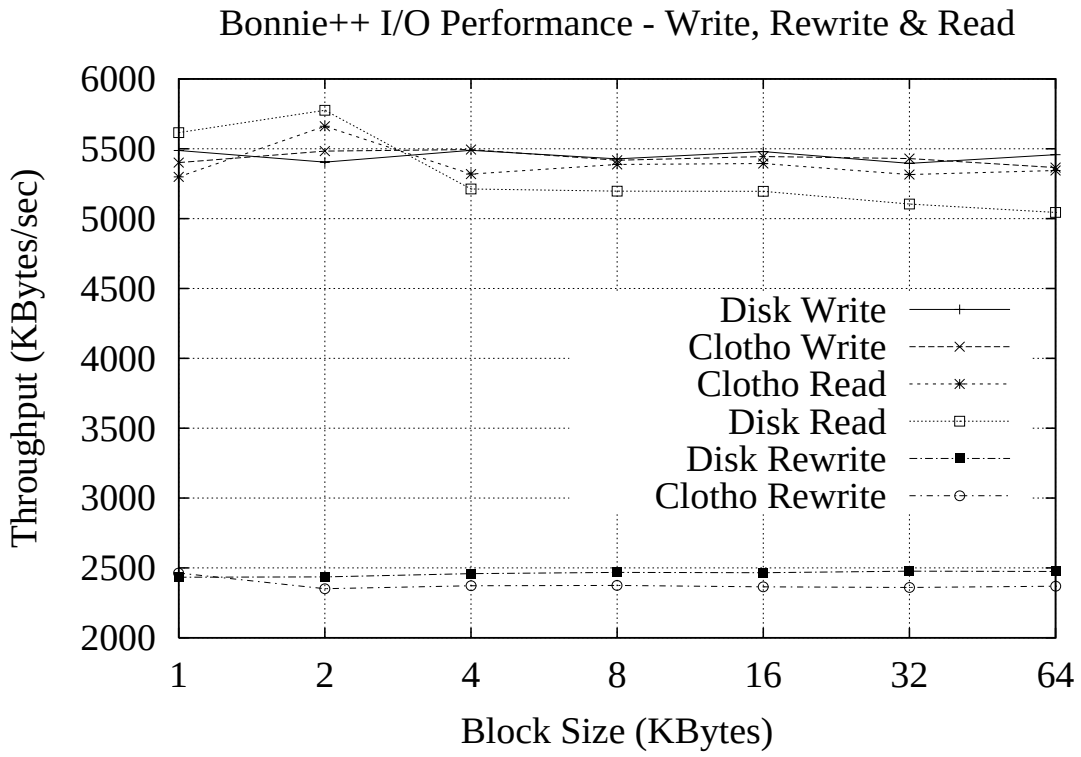


Figure 3.7: Bonnie++ throughput for write, rewrite, and read operations.

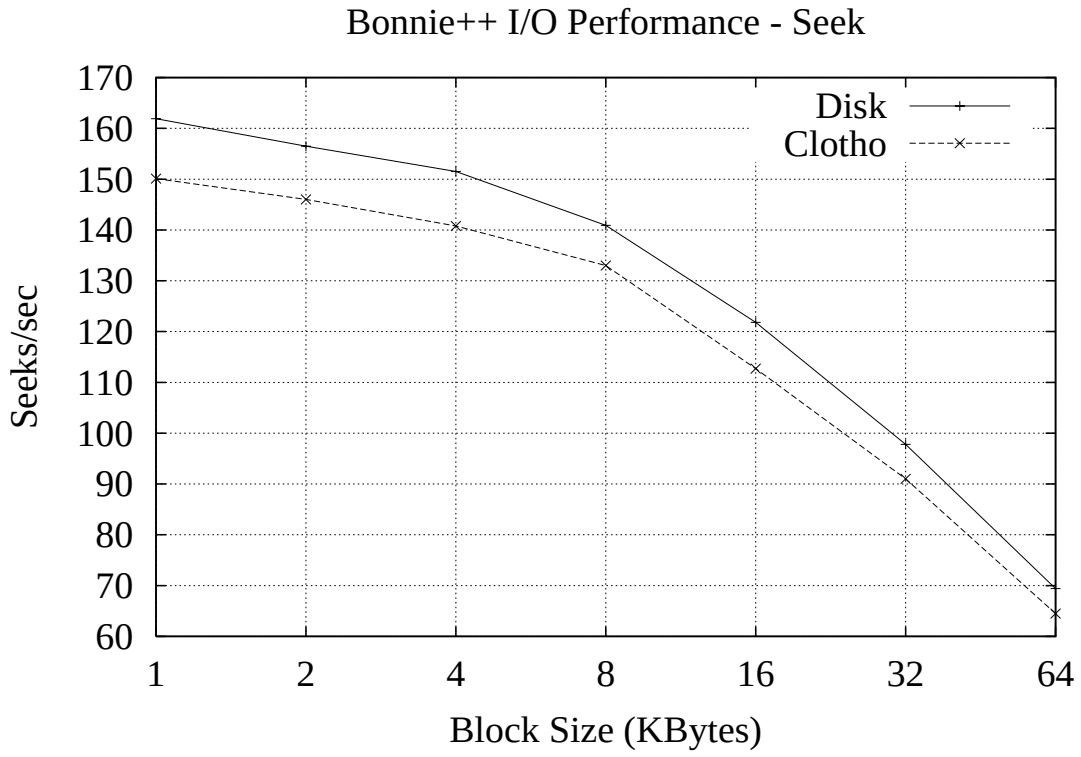


Figure 3.8: Bonnie++ “seek and read” performance.

3.4.1 Bonnie++

We use the Bonnie++ micro-benchmark to quantify the basic overhead of Clotho. The filesystem we use in all Bonnie++ experiments is the Ext2FS with a 4-KByte extent size. The size of the file to be tested is 2 GBytes with block sizes ranging from 1 KByte to 64 KBytes. We measure accesses to the latest version of a volume with the following operations:

- *Block Write:* A large file is created using the `write()` system call.
- *Block Rewrite:* Each block of the file is read with `read()`, dirtied, and rewritten with `write()`, requiring an `lseek()`.
- *Block Read:* The file is read using a `read()` for every block.
- *Random Seek:* Processes running in parallel are performing `lseek()` to random locations in the file and `read()`ing the corresponding file blocks.

Figure 3.7 shows that the overhead in write throughput is minimal and the two curves are practically the same. In the read throughput case, Clotho performs slightly better than the regular disk. This is due to the logging disk allocation policy that Clotho uses on writes, which however may have a negative effect on reads because it alters the data placement of the file system. In the rewrite case, the overhead of Clotho becomes more significant. This is due to the random "seek and read" operation overhead, as shown in Figure 3.8. Since the seeks in this experiment are random, Clotho's logging allocation has no effect and the overhead of translating I/O requests and flushing filesystem metadata to disk dominates. Even in this case, however, the overhead observed is at most 7.5% of the regular disk.

3.4.2 SPEC SFS

We use the SPEC SFS 3.0 benchmark suite to measure NFS file server throughput and response time over Clotho. We use one NFS client and one NFS server. The two systems that serve as client and server are connected with a switched 100 MBit/s Ethernet network. We use the following settings: NFS version 3 protocol over UDP/IP, one NFS exported directory, `biod_max_read=2`, `biod_max_write=2`, and requested loads ranging from 300 to 1000 NFS V3 operations/s with a 100 increment step. Both warm-up and run time are 300 seconds for each run and the time for all the SPEC SFS runs in sequence is approximately 3 hours. As mentioned above, we report results for the Ext2FS and ReiserFS (with -notail option) filesystems [ReiserFS]. A new filesystem is created before every experiment.

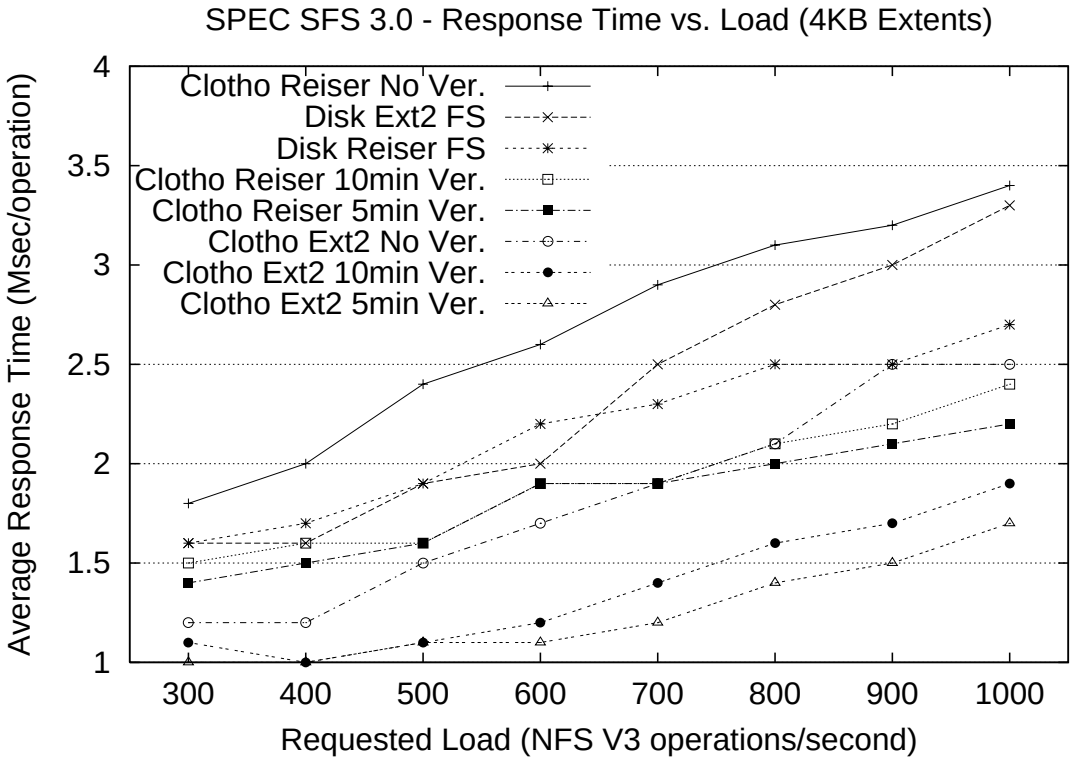


Figure 3.9: SPEC SFS response time using 4-KByte extents.

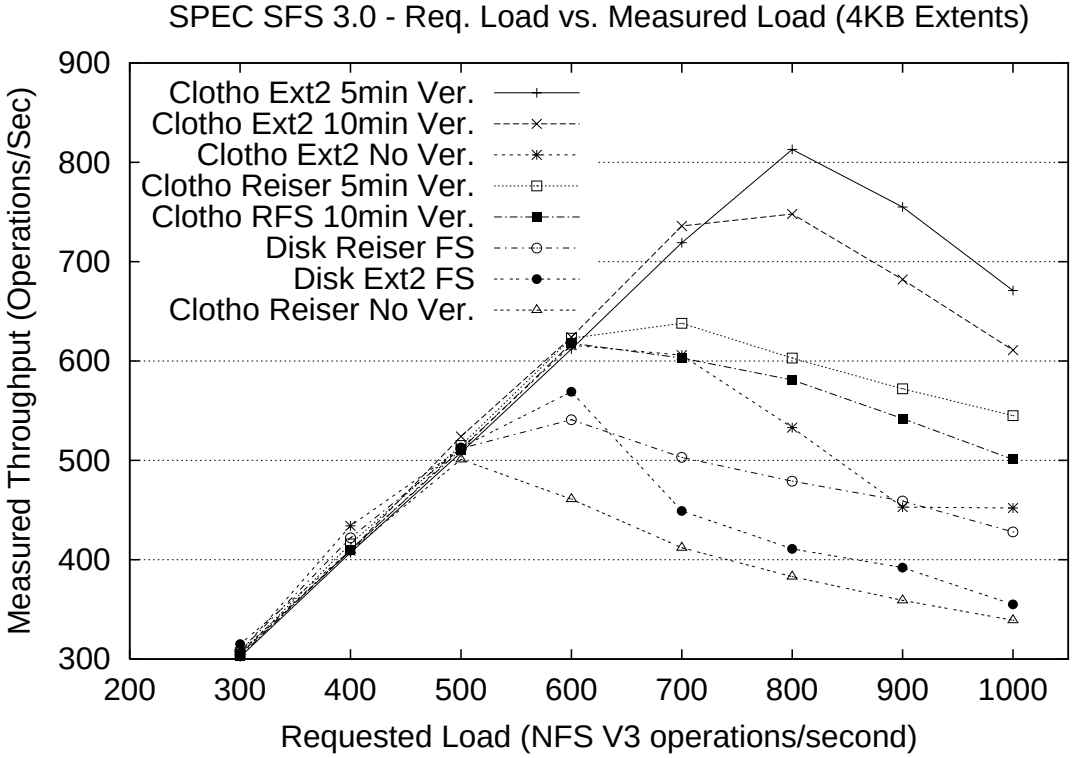


Figure 3.10: SPEC SFS throughput using 4-KByte extents.

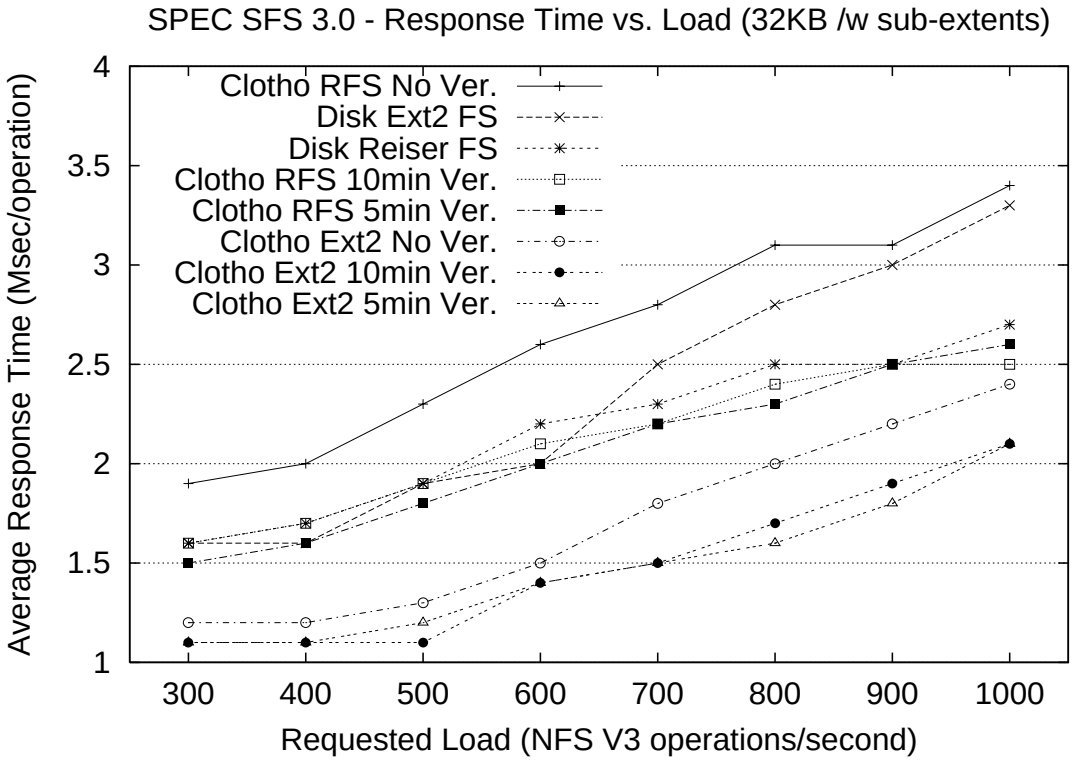


Figure 3.11: SPEC SFS response time using 32 KByte extents with subextents (RFS denotes ReiserFS).

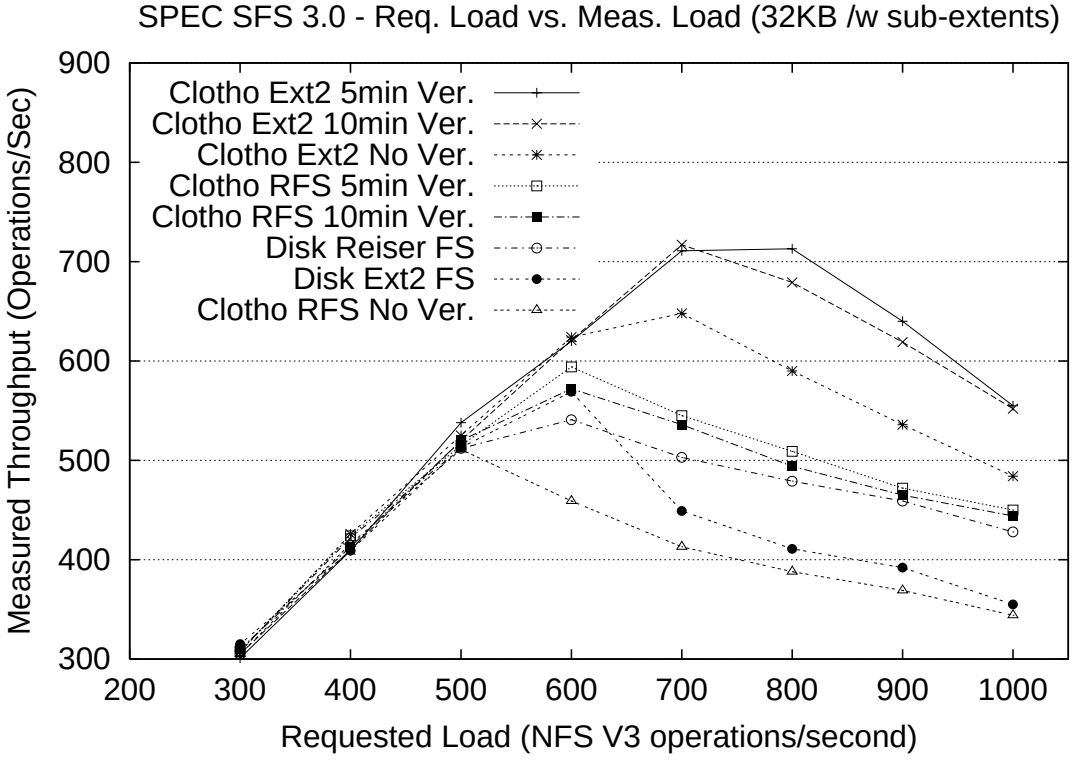


Figure 3.12: SPEC SFS throughput using 32-KByte extents with subextents (RFS denotes ReiserFS).

We conduct four experiments with SPEC SFS for each of the two filesystems: Using the plain disk, using Clotho over the disk without versioning, using Clotho and versioning every 5 minutes, and using Clotho with 10 minute versioning. Versioning is performed throughout the entire 3 hour run of SPEC SFS. Figures 3.9 and 3.10 show our throughput and latency results for 4-KByte extents, while Figures 3.11 and 3.12 show the results using 32-KByte extents with subextent addressing.

Our results show that Clotho outperforms the regular disk in all cases except ReiserFS without versioning. The higher performance is due to the logging (sequential) block allocation policy that Clotho uses. This explanation is reinforced by the performance in the cases where versions are created periodically. In this case, frequent versioning prevents disk seeks caused by overwriting of old data, which are now written to new locations on the disk in a sequential fashion. Furthermore, we observe that the more frequent the versioning, the higher the performance. The 32-KByte extent size experiments (Figures 3.11 and 3.12) show that even with much lower memory requirements, subextent mapping offers almost the same performance as the 4-KByte case. We attribute this small difference to the disk rotational latency, when skipping unused space to write subextents, while in the 4-KByte extent size, the extents are written "back-to-back" in a sequential manner.

Finally, comparing the two filesystems, Ext2 and ReiserFS, we find that the latter performs worse on top of Clotho. We attribute this behavior to Clotho's altered data layout that affects reads for sequential files. ReiserFS tries to optimize data placement for reads, however Clotho's logging policy underneath, places data depending on the sequence of write requests and not on the logical data address. This incurs more disk seeks for reading files that ReiserFS has placed sequentially and thus performance in the SPEC SFS workload is lower, compared to Ext2.

3.4.3 Compact version performance

Finally, we measure the read throughput of compacted versions to evaluate the space-time trade-off of diff-based compaction. Since old versions are only accessible in read-only mode, we developed a two-phase micro-benchmark that performs only `read` operations. In the first stage, our micro-benchmark writes a number of large files and captures multiple versions of the data through Clotho. In writing the data the benchmark is also able to control the amount of similarity between two versions, and thus, the percentage of space required by compacted versions. In the second stage, our benchmark `mounts` a compacted version and performs 2000 random read operations on the files of the compacted version. Before every run, the benchmark flushes the system's buffer cache.

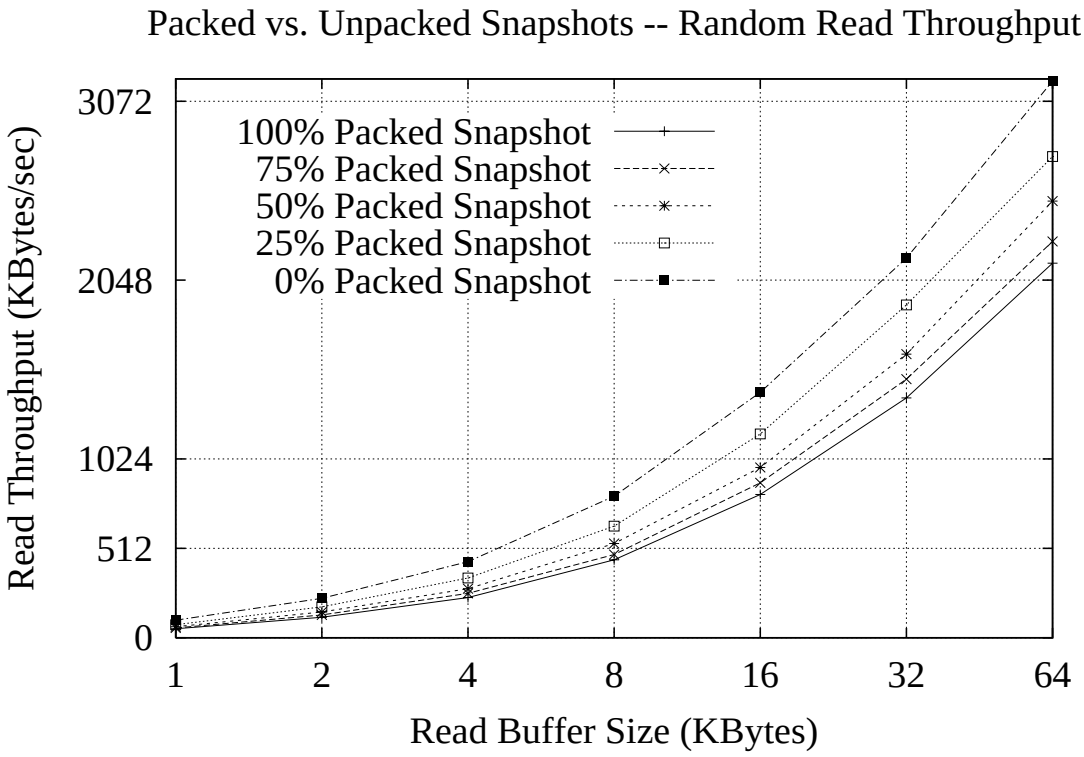


Figure 3.13: Random "compact-read" throughput.

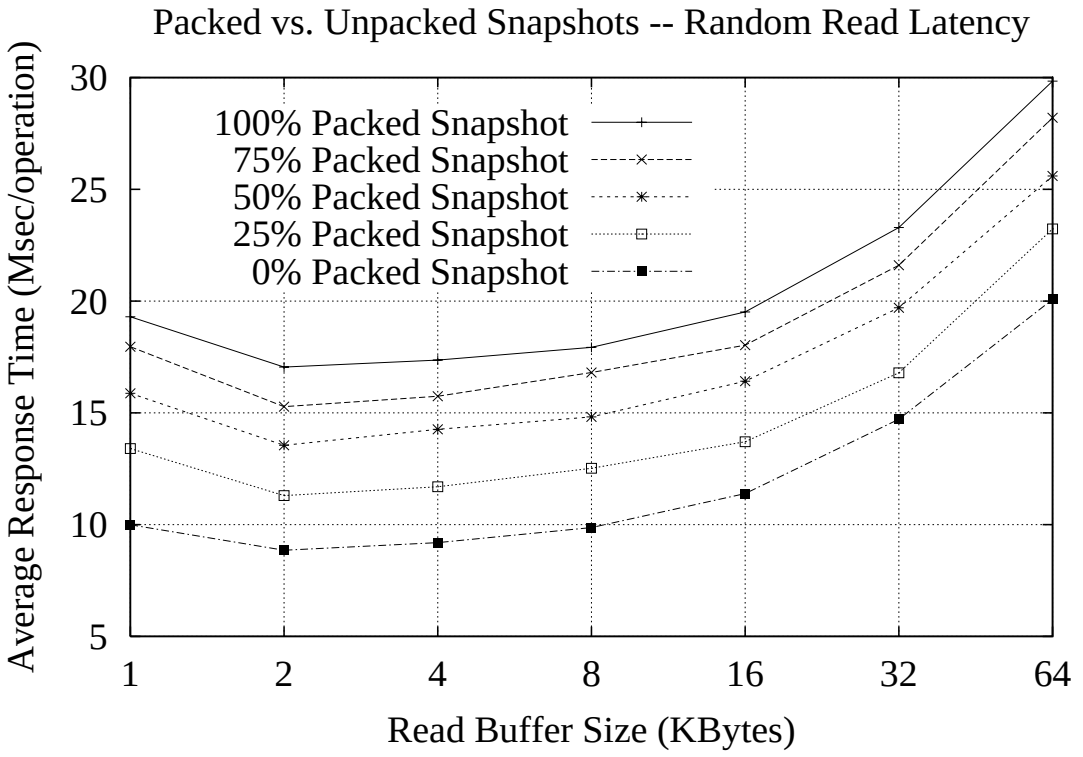


Figure 3.14: Random "compact-read" latency.

Figures 3.13 and 3.14 present latency and throughput results for different percentages of compaction. For 100% compaction, we save the maximum space on the disk, whereas in the case of 0% compaction we do not save any space at all. The difference in performance is mainly due to the higher number of disk accesses per read operation required for compacted versions. Each such read operation requires two disk reads to reconstruct the requested block. One read to fetch the block of the previous version and one to fetch the diffs. In particular, with 100% compaction, each and every read results in two disk accesses and thus, performance is about half of the 0% compaction case.

3.5 Limitations and Future work

The main limitation of Clotho is that it cannot be layered below or above abstractions that do not serialize versioning commands to virtual devices consisting of multiple physical devices. There are two instances of this issue: (i) when Clotho is layered below a virtual RAID device, or (ii) when Clotho is used at the client-side with shared block devices. If Clotho is layered below a volume abstraction that performs aggregation, policies for creating versions need to perform synchronized versioning across aggregated devices to ensure data consistency. However, this may not be possible in a transparent manner to higher system layers. We deal with this issue in RIBD, presented in Chapter 6, which supports globally consistent versions combined with a lightweight transactional API to higher layers. In the second case, when multiple clients have access to a shared block device, as is usually the case with distributed block devices [Lee and Thekkath, 1996; Thekkath et al., 1997], Clotho cannot be layered on top of the shared volume in each client, since internal metadata will become inconsistent across Clotho instances. A solution to this problem is an interesting topic for future work.

Another limitation of Clotho is that it imposes a change in the block layout from the input to the output layer due to the remap-on-write operation. In contrast to the copy-on-write approach, which has major write overhead, remap-on-write with a logging policy may affect read performance, if free blocks are scattered over the disk or if higher layers rely on a specific block placement, e.g. block 0 to x being sequential on the disk. However, this is an issue not only with Clotho, but with any block-level layer that performs logging, such as the systems described in [English and Alexander, 1992; Wang et al., 1999; Wilkes et al., 1996; Sivathanu et al., 2003; Wilkes et al., 1996; Stodolsky et al., 1994]. This issue is essentially related to the limitations of the block-level API and the imbalance in the I/O software stack towards the file system. We believe that as block-level I/O subsystems become more complex and

provide more functionality, general solutions to this problem will become necessary.

Finally, a drawback of using large extents for versioning in order to minimize metadata footprint is that we may waste disk space. Coarse-grain versioning (extents) compared to versioning per sectors or small logical blocks require copying or remapping a whole extent even if part of it has been modified. This is not an issue related to remap-on-write or copy-on-write, but to the granularity of versioning blocks. Depending on the file system and workload, a tradeoff between the size of the metadata footprint and the versioning granularity should be made to minimize wasted disk space.

3.6 Conclusions

Storage management is an important problem in building future storage systems. Online storage versioning can help reduce these costs directly, by addressing data archival and retrieval costs and indirectly, by providing novel storage functionality. In this chapter we propose pushing versioning functionality closer to the disk and implementing it at the block-device level. This approach takes advantage of technology trends in building active self-managed storage systems to address issues related to backup and version management.

We present a detailed design of our system, Clotho, that provides versioning at the block-level. Clotho imposes small memory and disk space overhead for version data and metadata management by using large extents, sub-extent addressing and diff-based compaction. It imposes minimal performance overhead in the I/O path by eliminating the need for copy-on-write even when the extent size is larger than the disk-block size and by employing logging (sequential) disk allocation. It provides mechanisms for dealing with data consistency and allows for flexible policies for both manual and automatic version management.

We implement our system in the Linux operating system and evaluate its impact on I/O path performance with both micro-benchmarks as well as the SPEC SFS standard benchmark on top of two production-level file systems, Ext2FS and ReiserFS. We find that the common path overhead is minimal for read and write I/O operations when versions are not compacted. For compact versions, the user has to pay the penalty of double disk accesses for each I/O operation that accesses a compact block.

Overall, our work in block-level versioning has allowed us to explore the usefulness, the mechanisms and the overheads associated with designing and implementing metadata-intensive functions at the block level. We find that block-level versioning has low overhead and offers advantages over filesystem-based approaches, such as transparency and simplicity. We thus believe that it is worthwhile to add support for

extensions with advanced, metadata-intensive functionality at the block-level, in order to allow flexible creation of customized application-specific storage volumes. We present our efforts in this direction in the next chapter.

Chapter 4

Violin: A Framework for Extensible Block-level Storage

Life is like playing the violin in public
and learning the instrument as one goes on.
—Samuel Butler (1835 - 1902)

The content of this chapter roughly corresponds to publication [Flouris and Bilas, 2005].

4.1 Introduction

The quality of virtualization mechanisms provided by a storage system affects storage management complexity and storage efficiency, both of which are important problems of modern storage systems. Storage virtualization may occur either at the filesystem or at the block level. Although both approaches are currently being used, in this thesis we address virtualization issues at the block-level. Storage virtualization at the filesystem level has mainly appeared in the form of extensible filesystems [Heidemann and Popek, 1994; Schermerhorn et al., 2001; Skinner and Wong, 1993; Zadok and Nieh, 2000].

We believe that the importance of block-level virtualization is increasing for two reasons. First, certain virtualization functions, such as compression or encryption, may be simpler and more efficient to provide on unstructured fixed data blocks rather than variable-size files. Second, block-level storage systems are evolving from simple disks and fixed controllers to powerful storage nodes [Acharya et al., 1998; Gibson et al., 1998; Gray, 2002] that offer block-level storage to multiple applications over a stor-

age area network [Barker and Massiglia, 2002; Patterson, 2000]. In such systems, block-level storage extensions can exploit the processing capacity of the storage nodes, where filesystems (running on the servers) cannot. For these reasons and over time, with the evolution of storage technology a number of virtualization features, e.g. volume management functions, RAID, snapshots, moved from higher system layers to the block level.

Today's operating systems provide flexibility in managing and accessing storage through I/O drivers (modules) in the I/O stack or through the filesystem. For instance, Linux-based storage systems use drivers, such as MD [De Icaza et al., 1997] and LVM [Teigland and Mauelshagen, 2001] to support RAID and logical volume management functions. Another example is block-level versioning systems, such as Clotho presented in the Chapter 3.

However, through our experience with Clotho we found that in order to add more block-level features, we are limited by the fact that current I/O stacks require the use of monolithic I/O drivers that are both complex to develop and hard to combine to build storage volumes with advanced, customized features. That is why current block-level systems offer predefined virtualization semantics without metadata-intensive functions, such as virtual volumes mapped to an aggregation of disks or RAID levels. In this category belong both research prototypes [De Icaza et al., 1997; EVMS; GEOM; Lehey, 1999; Teigland and Mauelshagen, 2001] and commercial products [EMC Enginuity; EMC Symmetrix; HP OpenView; Veritas, b,c]. In all these cases the storage administrator can tune various parameters at the volume level, but cannot extend the system with new, metadata-intensive functions.

In this chapter we address this problem by providing a kernel-level framework for (i) building and (ii) combining virtualization functions. We propose, implement, and evaluate Violin (Virtual I/O Layer INtegrator), a virtual I/O framework for commodity storage nodes that replaces the current block-level I/O stack with an improved I/O hierarchy that allows for (i) easy extension of the storage hierarchy with new mechanisms and (ii) flexible combining of these mechanisms to create modular hierarchies with rich semantics.

Although our approach shares similarities with work in modular and extensible filesystems [Heidemann and Popek, 1994; Schermerhorn et al., 2001; Skinner and Wong, 1993; Zadok and Nieh, 2000] and network protocol stacks [Kohler et al., 2000; Mosberger and Peterson, 1996; O'Malley and Peterson, 1992; Van Renesse et al., 1995], existing techniques from these areas are not directly applicable to block-level storage virtualization. A fundamental difference from network stacks is that the latter are essentially stateless (except for configuration information) and packets are ephemeral, whereas storage

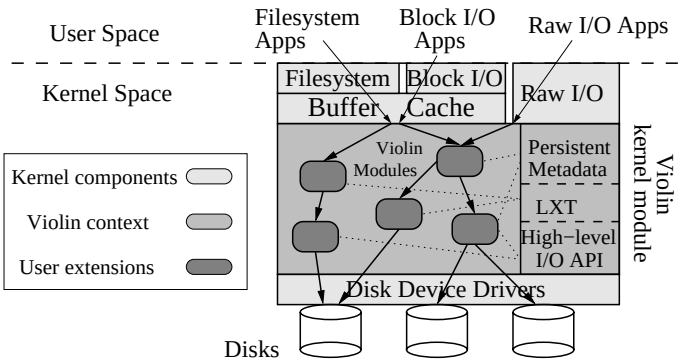


Figure 4.1: Violin in the operating system context.

blocks and their associated metadata need to persist. Compared to extensible filesystems, block-level storage systems operate at a different granularity, with no information about the relationships of blocks. Thus, metadata need to be maintained at the block level resulting potentially in large memory overhead. Moreover, block I/O operations cannot be associated precisely with each other, limiting possible optimizations.

The main contributions of Violin are: (i) it significantly reduces the effort to introduce new functionality in the block I/O stack of a commodity storage node and (ii) it provides the ability to combine simple virtualization functions into hierarchies with semantics that can satisfy diverse application needs. Violin provides virtual devices with full access to both the request and completion paths of I/Os allowing for easy implementation of synchronous and asynchronous I/O. Supporting asynchronous I/O is important for performance reasons, but also raises significant challenges when implemented in real systems. Also, Violin deals with metadata persistence for the full storage hierarchy, offloading the related complexity from individual virtual devices. To achieve flexibility, Violin allows storage administrators to create arbitrary, acyclic graphs of virtual devices, each adding to the functionality of the successor devices in the graph. In each hierarchy, blocks of each virtual device can be mapped in arbitrary ways to the successor devices, enabling advanced storage functions, such as dynamic relocation of blocks.

We have implemented Violin as a block device driver under Linux. To demonstrate the effectiveness of our approach in extending the I/O hierarchy we have also implemented various virtual modules as dynamically loadable kernel devices that bind to Violin's API. We have also developed simple user level tools that are able to perform on-line fine-grain configuration, control, and monitoring of arbitrary hierarchies of instances of these modules.

We evaluate the effectiveness of our approach in three areas: ease of development, flexibility, and

performance. In the first area we are able to quickly prototype modules for RAID levels, versioning, partitioning and aggregation, MD5 hashing, migration and encryption. In many cases, writing a new module is just a matter of recompiling existing user-level library code. Overall, using Violin encourages the development of simple virtual modules that can later be combined to more complex hierarchies. Regarding flexibility, we are able to easily configure I/O hierarchies that combine the functionality of multiple layers and provide complex high-level semantics that are difficult to achieve otherwise. Finally, we use PostMark and IOmeter to examine the overhead that Violin introduces over traditional block-level I/O hierarchies. We find that overall, internal modules perform within 10% (throughput) of their native Linux block-driver counterparts.

The rest of the chapter is organized as follows. Sections 4.2 and 4.3 present the design and implementation of Violin. Section 4.4 presents our results. Finally, Section 4.5 discusses limitations and future work and Section 4.6 draws our conclusions.

4.2 System Architecture

Violin is a virtual I/O framework that provides (i) support for easy and incremental extensions to I/O hierarchies and (ii) a highly configurable virtualization stack that combines basic storage layers in rich virtual I/O hierarchies. Violin's location in the kernel context is shown in Figure 4.1, illustrating the I/O path from the user applications to the disks. There are three basic components in the architecture of Violin:

1. High-level virtualization semantics and mappings.
2. Simple control over the I/O request path.
3. Metadata state persistence.

Next we discuss each of these components in detail.

4.2.1 Virtualization Semantics

A virtual storage hierarchy is generally represented by a *directed acyclic graph* (DAG). In this graph, the vertices or *nodes* represent *virtual devices*. The directed edges between nodes signify *device dependencies* and *control flow* through I/O requests. Control in Violin flows from higher to lower layers. This

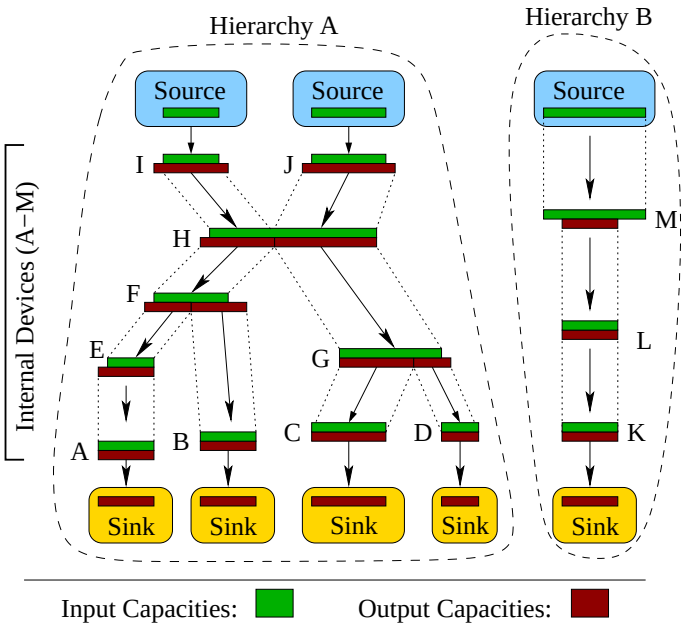


Figure 4.2: Violin's virtual device graph.

arises from the traditional view of the block-level device as a dumb passive device. Each virtual device in the DAG is operated by a virtualization module that implements desired functionality. Virtual devices that provide the same functionality are handled by different instances of the same module. From now on, we will use the terms module and device interchangeably.

Figure 4.2 shows an example of such a device graph. Graph nodes are represented with horizontal bars illustrating the mappings of their address spaces and they are connected with directed vertices. There are three kinds of nodes and accordingly three kinds of I/O modules in the architecture:

- *Source nodes* that do not have incoming edges and are top-level devices that *initiate I/O requests* in the storage hierarchy. The requests are initiated by external kernel components such as file systems or other block-level storage applications. Each of the source devices has an external name, e.g. an entry in /dev for Unix systems.

- *Sink* nodes that do not have outgoing edges and correspond to bottom-level virtual devices. Sink nodes sit on top of other kernel block-level drivers, such as hardware disk drivers and, in practice, are the final recipients of I/O requests.

- *Internal* nodes that have both incoming and outgoing edges and provide virtualization functions. These nodes are not visible to external drivers, kernel components, or user applications.

Violin uses the above generic DAG representation to model its hierarchies. A *virtual hierarchy* is defined as a set of connected nodes in the device graph that do not have links to nodes outside the hierarchy. A hierarchy within a device graph is a self-contained sub-graph that can be configured and managed independently of other hierarchies in the same system. Hierarchies in Violin are objects that are explicitly created before adding virtual devices (nodes).

To manage devices and hierarchies, users may specify the following operations on the device graph:

- Create a new internal, source, or sink node and link it to the appropriate nodes, depending on its type. The DAG is created in a bottom-up fashion to guarantee that an I/O sink will always exist for any I/O path.
- Delete a node from the graph. A node may be deleted only if its input devices have been deleted (top-down).
- Change an edge in the graph, i.e. remap a device.

Violin checks the integrity of a hierarchy at creation time and each time it is reconstructed. Checking for integrity includes various simple rules, such as the presence of cycles in the hierarchy graph and lack of input or output edges in internal nodes. Creating hierarchies and checking dependencies reduces the complexity of each Violin module.

A hierarchy in Violin is constructed with simple user-level tools implementing the above graph operations and linking the source and sink nodes to external OS block devices. Currently, the storage administrator has to specify the exact order in which virtual devices will be combined. The user-level tools can also be used online, during system operation to modify or extend I/O hierarchies. However, it is the administrator's responsibility to maintain data consistency while performing online structural changes to a hierarchy.

Dynamic Block Mapping and Allocation

Nodes in a hierarchy graph do not simply show output dependencies from one device to another but rather map between block address spaces of these devices. As can be seen in Figure 4.2, a storage device in the system represents a specific storage capacity and a block address space, while the I/O path in the graph represents a series of translations between block address spaces. Violin provides transparent and persistent translation between device address spaces in virtual hierarchies.

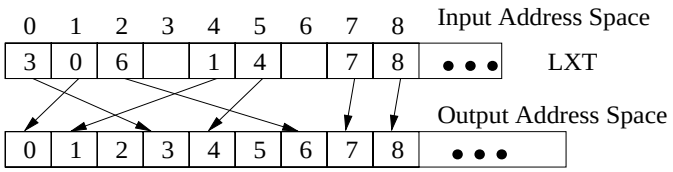


Figure 4.3: The LXT address translation table.

Devices in a virtual hierarchy may have widely different requirements for mapping semantics. Some devices, such as RAID-0, use simple mapping semantics. In RAID-0 blocks are mapped statically between the input and output devices. There are, however, modules that require more complex mappings. For instance, versioning at the block level requires arbitrary translations and dynamic block remappings, as discussed in Chapter 3. Similarly, volume managers [Teigland and Mauelshagen, 2001] require arbitrary block mappings to support volume resizing and data migration between devices. Another use of arbitrary mappings is to change the device allocation policy, for instance, to a log-structured policy. The Logical Disk [De Jonge et al., 1993] uses dynamic block remapping for this purpose.

Violin supports dynamic block mapping through a logical-to-physical block address translation table (LXT). Figure 4.3 shows an example of such an LXT mapping between the input and output address spaces of a device. Dynamic block mapping capabilities give to a virtual device the freedom to use its own disk space management mechanisms and policies, without changing the semantics of its input devices, higher in the I/O stack.

Additionally, Violin provides free-block allocation and management facilities. It offers a variety of disk allocation schemes, including first-available, log-structured, and closer-to-last-block, all of which use either a free list (*FL*) or a physical block bitmap (*PXB*) to distinguish between occupied and free physical blocks. However, since modules may need their own space allocation algorithm, Violin allows module code to directly access the PXB data structure as well as its persistent metadata objects (explained below). In this case, however, code and complexity increase for the module developer.

The API for dynamic block mapping and allocation is shown in Figure 4.4. Since the LXT and the PXB are addressed in extents (explained below), the library functions take as arguments an extent number, which addresses an LXT or PXB entry. Functions `get / set_lxt_bool_flag_on/off()` handle binary flag values that can be maintained on the LXT for the extents of a device. This is useful for modules that *tag* extents according to their properties (e.g. clean/dirty). Functions `vl_phys_address_get / set()` handle the physical addresses stored on the LXT entries. Functions `vl_phys_extent_alloc()`

```
err_code vl_alloc_lxt(_device_t *dev);
err_code vl_alloc_pxb(_device_t *dev);
int  get/set_lxt_bool_flag_on/off(int extent, VL_FLAGMASK_TYPE flagmask);
int  vl_phys_address_get/set(int extent);
int  vl_phys_extent_alloc(int alloc_policy);
void vl_phys_extent_free(int extent);
```

Figure 4.4: Violin API for LXT and PXB data structures.

and `vl_phys_extent_free()` operate on the PXB bitmap. The first routine locates a free physical block (according to an algorithm) and marks it on the PXB, while the second frees a used data block and clears its PXB bit.

The LXT and PXB data structures are allocated as persistent objects using the persistent metadata calls we describe in Section 4.2.3 and their persistence is handled automatically by Violin similarly to the rest of the hierarchy metadata.

An issue with supporting arbitrary block mappings is the size of LXT and FL or PXB persistent objects, which can become quite large, depending on the capacity and block size of a device. Keeping such objects in memory can increase memory footprint substantially. Violin offers two solutions: First, it supports an *internal block size* ("extent size") for its devices, which is independent of the OS device block size. The extent size, in other words, is a fixed logical block for the device and is specified when a device is created. Increasing the extent size can greatly reduce the size of the LXT. For example, the LXT of a 1 TByte device with 32-bit extent addressing and 4 KByte extent size would require about 1 GByte of memory. By increasing the extent size to 32 KBytes, we achieve a reduction of the required metadata to 128 MBytes. Second, Violin provides the capability to explicitly load and unload persistent metadata to and from memory, as explained in Section 4.2.3.

4.2.2 Violin I/O Request Path

The second significant aspect of Violin is how the I/O request path works. Violin is not only reentrant but also supports synchronous and asynchronous I/O requests. I/O requests never block in the framework, unless a driver module has explicitly requested it. Moreover, since Violin is reentrant, it runs in the issuer's context for each request issued from the kernel. Thus, many requests can proceed concurrently in the framework, each in a different context.

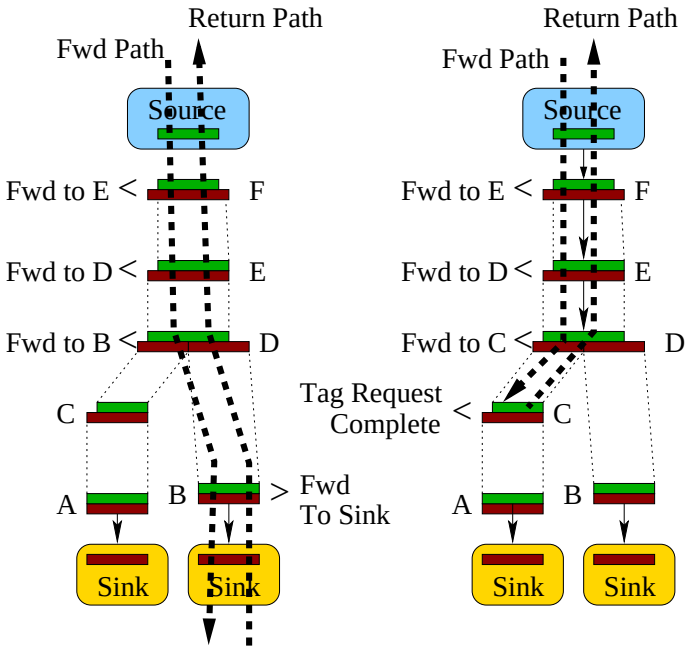


Figure 4.5: Example of request flows (dashed lines) through devices. Forward paths are directed downwards, while return paths upwards.

A generic virtual storage framework must support two types of I/O requests:

- *External requests* are initiated by the kernel. They enter the framework through the source devices (nodes), traverse a hierarchy through internal nodes usually until they reach a sink node and then return back up the same path to the top of the hierarchy.

- *Internal requests* are generated from internal devices as a response to an external request. Consider for example a RAID-5 module that needs to write parity blocks. The RAID-5 device generates an internal write request to the parity device. Internal requests are indistinguishable from external ones for all but the generating module and are handled in the same manner.

I/O requests in Violin move from source to sink nodes through some path in a virtual hierarchy, as shown in Figure 4.5. Sink devices are connected to external block devices in the kernel, so after a request reaches a sink device it is forwarded to an external device. When multiple output nodes exist, routing decisions are taken at every node, according to its mapping semantics. Virtual devices can control requests beyond simple forwarding. When a device receives an I/O request it can make four control decisions and set corresponding control tags on a request:

- *Error Tag* indicates a request error. Erroneous requests are returned through the stack to the issuer

with an error code.

- *Forward Tag* indicates that the request should be forwarded to an output device. In this case, the target device and block address must also be indicated. Forwarding occurs to the direction of one of the output graph vertices.

- *Return Control Path Tag* indicates that a device needs return path control over the request. Some devices need to know when an asynchronous I/O request has completed and need control over its return path through the hierarchy (Figure 4.5). For instance, an encryption module requires access to the return path of a read request, because data needs to be decrypted after it is read from the sink device. However, many modules do not require return path control. If no module in a path through the hierarchy requests return path control, the request is merely redirected to another kernel device outside the framework (e.g. the disk driver) and returned directly to the issuer application, bypassing the framework's return-path control. If, however, some module requests return-path control, the framework must gain control of the request when it completes in another kernel device. In Violin's Linux implementation, the driver sets an asynchronous callback on the request to gain control when it completes. When the callback receives the completed request, it passes it back up through the module stack in the reverse manner, calling the *pop I/O handlers*, described later in the API Section 4.2.4. Errors in the return path are handled in a similar manner as in the forward (request) path.

- *Complete Tag* indicates that a request is completed by this device. Consider the example of a caching module in the hierarchy. A caching device is able to service requests either from its cache or from the lower-level devices. If a requested data block is found in the cache, the device loads the data in the request buffer and sets the "complete" tag. The completed request is not forwarded deeper in the hierarchy, but returns from this point upwards to the issuer as shown at the right of Figure 4.5 for device C. If the return control path tag is set, the framework passes control to the devices in the stack that have requested it. Using this control tag, an internal device behaves as an new internal sink device.

A final issue with I/O requests flowing through the framework is dependencies between requests. For instance, there are cases where a module requires an external request to wait for one or more internal requests to complete. To deal with this, when an internal request X is issued (asynchronously) the

```
void *vl_persistent_obj_alloc(int oid,int size);
void *vl_persistent_obj_[un]load(int oid);
void vl_persistent_obj_flush(int oid);
void vl_persistent_obj_free(int oid);
```

Figure 4.6: Violin API for metadata management.

issuer module may register one or more dependencies of X to other requests (Y, Z, ...) and provide asynchronous callback functions. Requests X, Y, Z are processed concurrently and when each completes the callback handler is called. The callback handler of the module then processes the dependent requests according to the desired ordering (e.g. it may wait for all or a few requests to finish before releasing them). This mechanism supports arbitrary dependencies among multiple requests. Cyclic dependencies that lead to deadlock can occur by erroneous module code which is the responsibility of the module developer. One simple such case is when a device sends requests to devices higher in the same I/O path that can end up in the same module.

4.2.3 State Persistence

One of the most important issues in storage virtualization is metadata management and persistence. Violin supports metadata management facilities. The three main issues associated with metadata are: facilitating the use of persistent metadata, reducing memory footprint, and providing consistency guarantees.

Persistent Metadata

In Violin, modules can allocate and manage persistent metadata objects of varying sizes using the API summarized in Figure 4.6. To allocate a metadata object, a module calls `vl_persistent_obj_alloc()` using a unique object ID and its size, as it would allocate memory.

Modules access metadata directly in memory, as any other data structure. However, since device metadata are loaded dynamically when the device is initialized, pointers may not point to the right location. There are two methods for addressing this issue: (i) disallow pointers in metadata accesses and (ii) always force loading of metadata to the same address. Since the latter can be a problem for large metadata objects and in our experience the former is not overly restrictive, in our current design we disallow pointers in metadata objects. However, this issue requires further investigation and is left for future

work. When a module needs to destroy a persistent object it calls the `vl_persistent_obj_free()` function.

By default, when a virtual device is created in Violin, the system stores its metadata in the same output address space as the data. Alternatively, the user may specify explicitly a metadata device that will be used for storing persistent metadata for the new virtual device. This can be very useful when metadata devices are required to have stronger semantics compared to regular data devices, e.g. a higher degree of redundancy.

Violin's metadata manager automatically synchronizes dirty metadata from memory to stable storage in an asynchronous manner. The metadata manager uses a separate kernel thread to write to the appropriate device metadata that are loaded in memory. The user can also flush a metadata object explicitly with the `vl_persistent_obj_flush()` call.

Internally, each metadata object is represented with an object descriptor, which is modified only by allocation/deallocation calls. During these calls metadata object descriptors are stored in a small index in the beginning of the metadata device. A pointer to the metadata header index is stored with each virtual device in the virtual hierarchy. Thus, when the virtual hierarchy is being loaded and recreated, Violin reads the device metadata and loads it to memory.

Reducing Memory Footprint

Large metadata objects may consume a large amount of main memory or incur large amounts of I/O during metadata synchronization to stable storage. In these cases reducing the metadata memory footprint is important, in order to keep overheads low. Violin provides API calls for explicitly loading and unloading metadata to and from memory within each module (Figure 4.6). Another possible solution is to transparently swap metadata to stable storage. This solution is necessary for extremely large devices, where the size of the required metadata can sometimes exceed the available memory of a machine. Since it is not clear whether such modules actually need to be implemented, Violin does not support this feature yet.

Violin retains a small amount of internal metadata that represent the hierarchy structure and are necessary to reconstruct the device graph. Each graph node and edge requires a pointer, which amounts to a few bytes per virtual device. The hierarchy metadata is saved in all physical devices within a hierarchy. For this purpose, Violin reserves a superblock of a few KBytes in every physical device of a hierarchy. A complete hierarchy configuration can thus be loaded from the superblock of any physical

device that belongs to the hierarchy.

In summary, each module needs to allocate its own metadata, and in the common case, where metadata can be kept in memory, it does not need to perform any other operation. The framework will keep metadata persistent and will reload the metadata from disk each time the virtual device is loaded, significantly simplifying metadata management in individual modules.

Metadata Consistency

In the event of system failures, where a portion of the in-memory metadata may be lost or partially written, application and/or Violin state may be corrupted. We can define the following levels of metadata consistency:

1. *Lazy-update consistency*, that is, metadata are synchronized on disk overwriting the older version every few seconds. This means that if a failure occurs between or during updates of metadata then metadata may be left inconsistent on-disk and Violin may not be able to recover. In this case, there is a need for a Violin-level recovery procedure (similar to `fsck`), which however, we do not currently provide. This level of consistency would be sufficient when systems are expected to crash infrequently (e.g. when they are considered stable and are supported by an uninterruptible power supply (UPS)). If, however, a system is expected to crash often and/or stronger consistency guarantees are required, then one of the next forms of consistency may be used instead.

2. *Shadow-update consistency*, where we use two metadata copies on disk and maintain at least one of the two consistent at all times. If during an update the set that is currently being written becomes inconsistent due to a failure, Violin uses the second copy to recover. In this case, it is guaranteed that Violin will recover the device hierarchy and all its persistent objects and will be able to service I/O requests. However, application data may be inconsistent with respect to system metadata.

3. *Atomic metadata consistency*, guarantees that after a failure, the system will be able to access a previous, consistent state of application data and system metadata. This level of consistency can be implemented using a roll-forward log (journal) [Hagmann, 1987], or a roll-back versioning scheme. In the former case the log is played forward for recovery, while the latter is equivalent to a rollback of metadata to a previous point in time. The second option can be achieved in Violin using a versioning layer similar to Clotho (Chapter 3) at the leaves of a hierarchy. Although such

```
-> initialize (_device_t *dev, ...);
-> open  (_device_t *dev,void *data);
-> close (_device_t *dev,void *data);
-> read_push  (_device_t *dev, ...);
-> read_pop   (_device_t *dev, ...);
-> write_push (_device_t *dev, ...);
-> write_pop  (_device_t *dev, ...);
-> ioctl  (_device_t *dev, int cmd, ...);
-> resize (_device_t *dev, int new_size);
-> device_info (_device_t *dev, ...);
```

Figure 4.7: API methods for Violin's I/O modules.

a layer is available in Violin, given its current implementation we need to slightly modify it so that its own metadata are handled differently in this particular case.

Violin currently supports the first and second forms of metadata consistency. We expect that all three forms of consistency will be available in the future versions of the framework code.

4.2.4 Module API

Extending an I/O hierarchy with new functionality is an arduous task in modern kernels. The interface provided by kernels for block I/O is fairly low-level. A block device driver has the role of servicing block I/O `read` and `write` requests. Block requests adhere to the simple block I/O API, where every request is denoted as a tuple of `(block device,read/write,block number,block size,data)`. In Linux, this API involves many tedious and error prone tasks, such as I/O request queue management, locking and synchronization of the I/O request queues, buffer management, translating block addresses and interfacing with the buffer cache and the VM subsystem.

Violin provides to its modules high-level API calls, that intuitively support its hierarchy model and hide the complexity of kernel programming. The author of a module must set up a *module object*, which consists of a small set of variables with the attributes of the implemented module and a set of methods or API functions that can be seen in Figure 4.7. The variables of the module object in our current implementation include various static pieces of information about each module, such as the module name and id, the minimum and maximum number of output nodes supported by the device, and the

number of non-read and write (ioctl) operations supported by the module. The role of each method prototype is:

- `initialize()` is called once, the first time a virtual device is created to allocate and initialize persistent metadata and module data structures for this device. Note that since Violin retains state in persistent objects, this method will not be called when a hierarchy is loaded, since device state is also loaded.

- `open(), close()` are called when a virtual device is loaded or unloaded during the construction of hierarchy. The module writer can use the `close()` call to perform garbage collection and other necessary tasks for shutting down the device.

- `read(), write()` handle I/O requests passed to a virtual device. Each has two methods, `push` and `pop`, that represent the different I/O paths. `Push` handlers are called in the forward path, while `pop` handlers are called in the return path, if requested. These API methods are higher-level compared to their kernel counterparts and do not require complex I/O buffer management and I/O queue management.

- `ioctl()` handles custom commands and events in a module, in case direct access to it is required. This is used in some of our modules, for example to explicitly trigger a version capture event in the versioning module.

- `resize()` is an optional method that specifies a new size for an output virtual device. When this method is called in a virtual device, it means that one of its output devices has changed size as specified and thus, the virtual device has to adjust all internal module metadata appropriately.

- `device_info()` is used to export runtime device information in human or machine readable form.

The main difference of this API from the classic block driver API in Unix systems is the distinct push() and pop() methods for read and write operations, versus a read(), write() in the classic API. This is the key features of Violin's API, that allows asynchronous behavior. Finally the role of the initialize() call is different from the classic API. Since state persistence is maintained, device initialization is necessary only once in its lifetime, at its creation.

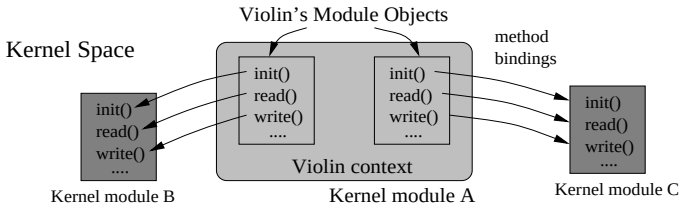


Figure 4.8: Binding of extension modules to Violin.

4.3 System Implementation

We have implemented Violin as a loadable block device driver in the Linux 2.4 kernel accompanied by a set of simple user-level management tools. Our prototype implements fully the I/O path model described in Section 4.2.4. Violin extension modules are implemented as separate kernel modules that are loaded on demand. However, they are not full Linux device drivers themselves but require the framework's core. Upon loading, each module registers with the core framework module, binding its methods to internal *module objects* as shown in Figure 4.8.

A central issue with the Violin driver is the implementation of its asynchronous API on Linux 2.4. Next, we explain how I/O requests are processed in Linux block device drivers and then describe Violin's implementation.

4.3.1 I/O Request Processing in Linux

The Linux kernel uses an *I/O request* structure to schedule I/O operations to block devices. Every block device driver has an *I/O request queue*, containing all the I/O operations intended for this block device. Requests are placed in this queue by the kernel function `make_request_fn()`. This function can be overridden by a block driver's own implementation. Requests are removed from the queue and processed with the *strategy* or *request* function of a block driver. When a request is complete, the driver releases the I/O request structure back to the kernel after tagging it as successful or erroneous. If a released request is not appropriately tagged or the data is incorrect, the kernel may crash.

Thus, there are two ways of processing I/O requests in a Linux device driver. First, a device driver can use a request queue as above where the kernel places requests and the *strategy* driver function removes and processes them. This approach is used by most hardware block devices. Second, a driver may override the `make_request_fn()` kernel function with its own version and thus, gain control of every I/O request before it is placed in the request queue. This mode of operation is more appropriate

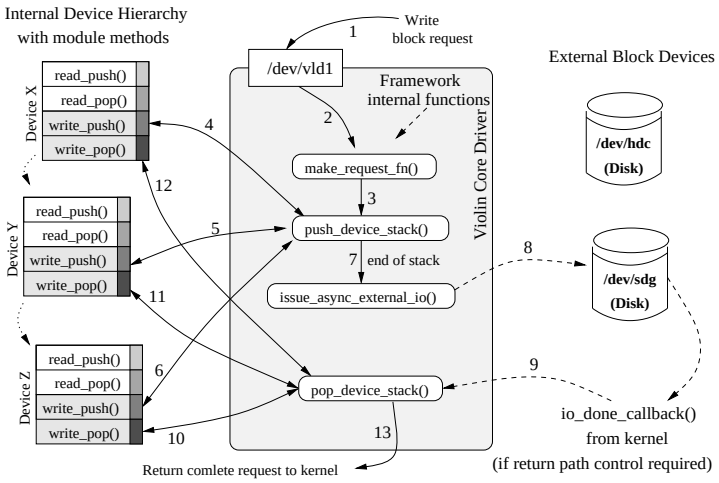


Figure 4.9: Path of a write request in Violin.

for virtual I/O drivers. In this mode, the driver does not use an I/O request queue, but rather *redirects* requests to other block devices. This is the mode of operation used by many virtual I/O drivers, such as LVM and MD. When a new I/O request arrives, the device driver performs simple modifications to the I/O request structure, such as modifying the device ID and/or the block address and returns the request to the kernel that will redirect it to the new device ID.

4.3.2 Violin I/O path

Violin uses the method of replacing the `make_request_fn()` call and operates asynchronously without blocking inside this function. For every external I/O request received through the `make_request_fn()` call, Violin's device driver traverses the I/O stack and calls the necessary module I/O handlers (`read_push()` or `write_push()` depending on the type of request). The I/O path traversal for a write request can be seen in Figure 4.9.

Each handler processes the incoming request (modifying its address, data, or both) and then returns control to the `push_device_stack()` function, indicating the next device in the path from one of its dependent (or lower) devices, marked with outward arrows. Thus, the `push_device_stack()` function passes the request through all the devices in the path. When the request reaches an exit node, the driver issues an asynchronous request to the kernel to perform the actual physical I/O to another kernel driver (e.g. the disk driver). When return path control is needed, the driver uses callback functions that Linux attaches to I/O buffers. Using these callbacks the driver is able to regain control of the I/O requests when they complete in another driver's context and perform all necessary post-processing.

A Linux-specific issue that Violin's kernel driver must deal with, is handling buffer cache data that need module processing and modification (e.g. encryption). The Linux kernel I/O request descriptors contain pointers to memory pages with the data mapped by the buffer cache and other kernel layers, e.g. the filesystem, to disk blocks. While an I/O request is in progress, the memory page remains available for read access in the buffer cache. Data-modifying layers, e.g. encryption, however, need to transform the data on the page before writing it to the disk. Data modification on the original memory page results in the corruption of the filesystem and buffer cache buffers. To resolve this problem with data-modifying modules, Violin creates a copy of the memory page, where modules can modify the data, and writes the new page to the disk. The original page remains available in the buffer cache.

4.4 Evaluation

In this section we evaluate the effectiveness of Violin in three areas: ease of development, flexibility, and performance.

4.4.1 Ease of Development

Measuring development effort is a hard task, since there exist no widely accepted metrics for it. We attempt to quantify the development effort by looking at the code size reduction that we can achieve using Violin. Similar to FiST [Zadok and Nieh, 2000], we use code size as a metric to examine relative complexity. We compare code sizes of various block device drivers that exist for Linux with implementations of these modules under Violin:

- **RAID:** Implements RAID levels 0 and 1. Failure detection and recovery is also implemented for RAID level 1. Using Violin's ability to synthesize complex hierarchies, these modules can be used to create composite RAID levels, such as 1+0.
- **Versioning:** Implements the on-line block-level versioning functionality of Clotho in a Violin hierarchy. The design of Clotho was presented in Chapter 3.
- **Partitioning:** Creates a partition on top of another device. It supports partition tables and partition resizing.
- **Aggregation:** Functions as a volume group manager. It aggregates many devices into one, either appending each after the other or using striping, and allows resizing of the individual or aggregate

Virtualization Layers / Functions	Number of code lines	
	Linux Driver	Violin Module
RAID	11223 (MD)	2509
Partition & Aggregation	5141 (LVM)	1521
Versioning	4770 (Clotho)	809
MD5 Hashing	–	930
Blowfish Encryption	–	804
DES & 3DES Encryption	–	1526
Migration	–	422
Core Violin Framework	14162	–

Table 4.1: Linux drivers and Violin modules in kernel code lines.

device. Also, devices can be added or removed and data can be migrated from one device to another.

- **MD5 hashing:** Computes the MD5 fingerprint for each block. Fingerprints are inserted in a hash table and can be queried. Its purpose is to offer content-based search capabilities and facilitate differential compression.

- **Encryption:** Encrypts/decrypts data blocks, using a user-specified key. We have currently implemented the Blowfish, DES and 3DES encryption algorithms.

- **Online Migration:** Transparently migrates data from one device to another, at a user-controlled speed (throttling).

Table 4.1 shows the code size for each functional layer implemented as a driver under Linux and as a module under Violin. The features implemented in Violin modules are close to the features provided by the block drivers under Linux, including LVM. The only exception exists in the MD Linux [De Icaza et al., 1997] driver. Our implementation, although it includes code for failure detection and recovery in RAID, does not support the feature of spare disks in a RAID volume. The main reason for not adding this functionality is that we expect it to be handled by remapping failed disks to spare ones with Violin.

We see that the size of the Violin framework core is about 14200 lines of code. Looking at individual modules, in the case of LVM, code length is reduced by a factor of three, while in MD and Versioning the reduction is about a factor of four and six respectively. In MD, more than half of the module code (1300 lines) is copied from MD and is used for fast XOR computation. This piece of code is written mainly in assembly and tests various XOR implementations for speed in order to select the fastest one for each system. If we compute code-length differences without the user-level copied code, i.e. the XOR code, the code difference for the RAID module reaches an order of magnitude.

Clearly code size cannot be used as a precise measure of complexity. However, it is an indication of the reduction in effort when implementing new functionality under Violin. For the modules we implement, we can attribute the largest code reductions to the high-level module API and to the persistent metadata management support.

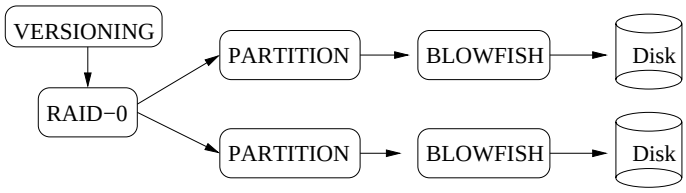
To examine how new functionality can be provided under Violin, we implement two new modules for which we could not find corresponding Linux drivers: MD5 hashing and DES-type encryption. For each module we use publicly available user-level code. Table 4.1 shows that MD5 hashing is about 900 lines of code in Violin out of which 416 lines are copied from the user-level code. Similarly, our DES encryption module uses about 1180 lines of user-level code, out of a total of 1526 lines. Creating each module was an effort of just a few hours.

4.4.2 Flexibility

To demonstrate the flexibility of the proposed framework we show how one can create more complex multi-layered hierarchies by combining simple modules. Providing the same functionality with standard OS mechanisms is much more difficult. In this demonstration, we construct a complex hierarchy of virtual devices over a set of physical devices, adding one layer at a time.

Figure 4.10 depicts how such a hierarchy is built in a bottom-up fashion using simple user-level tools. Initially two disks are inserted in the hierarchy as sink devices. Next, a partitioning layer is added, which creates two partitions, one on each disk. Then, a RAID-0 layer creates a striped volume on top of the two partitions. Next, a versioning layer is added on top of the striped volume to provide online versioning. Finally, a Blowfish encryption layer is added in the hierarchy to encrypt data. While this module can be placed anywhere in the hierarchy, we chose to place it directly above the disks.

Overall, we find that Violin simplifies the creation of complex hierarchies that provide advanced functionality, once the corresponding modules are available.




```
#SCRIPT TO CREATE THE HIERARCHY:
#Initialize a hierarchy named "voila"
hrc_init /dev/vld voila 32768 1
#Import two IDE disks: /dev/hdb and /dev/hdc
dev_import /dev/vld /dev/hdb voila hdb
dev_import /dev/vld /dev/hdc voila hdc
#Create a Blowfish Encryption Layer on each disk
dev_create /dev/vld Blowfish voila BF_Dev_1 1 hdb BF_Dev_1 0
dev_create /dev/vld Blowfish voila BF_Dev_2 1 hdc BF_Dev_2 0
#Create two Partitions on top of these
dev_create /dev/vld Partition voila Part_Dev_1 1 BF_Dev_1 -1 900000 Part_Dev_1 0
dev_create /dev/vld Partition voila Part_Dev_2 1 BF_Dev_2 -1 900000 Part_Dev_2 0
#Create a RAID-0 device on the 2 partitions
dev_create /dev/vld RAID voila RAID0_Dev 2 Part_Dev_1 Part_Dev_2 0 32 RAID0_Dev 0
#Create a Versioned device on top
dev_create /dev/vld Versioning voila Version_Dev 1 RAID0_Dev 40 Version_Dev 200000
#Link hierarchy "voila" to /dev/vld1
dev_linkpart /dev/vld voila Version_Dev 1
#Make a filesystem on /dev/vld1 and mount it
mkfs.ext2 /dev/vld1
mount -t ext2 /dev/vld1 /vldisk
```

Figure 4.10: A hierarchy with advanced semantics and the script that creates it.

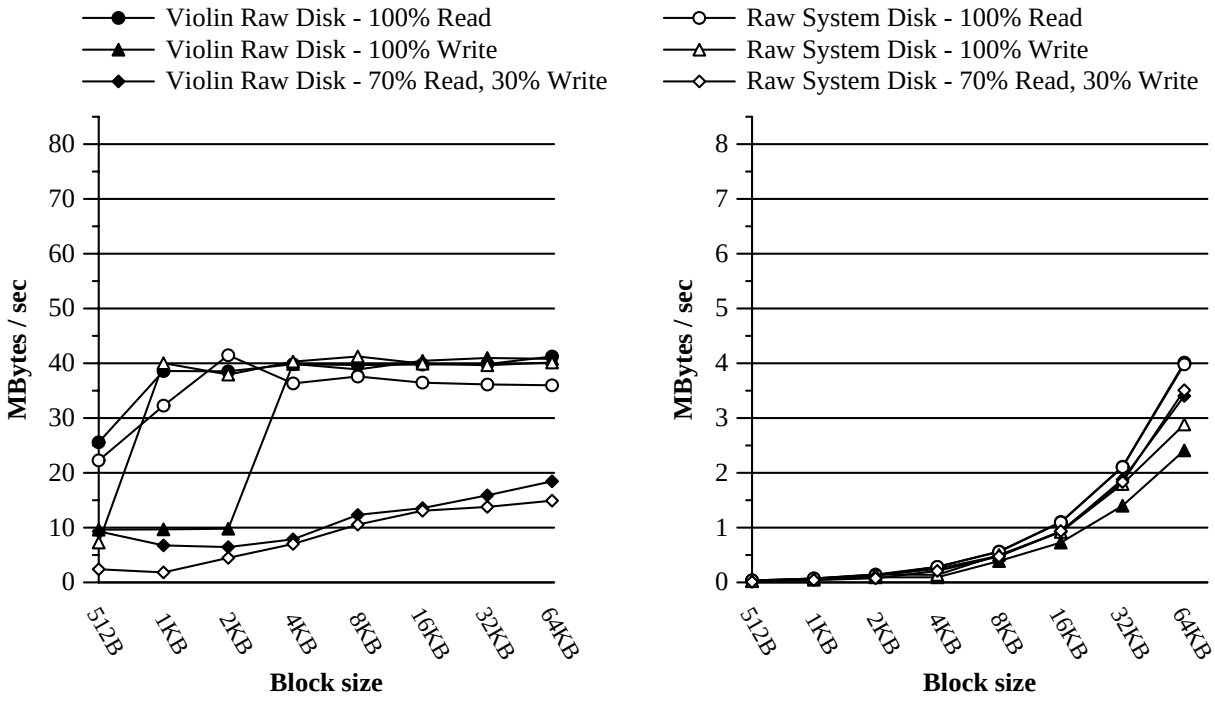


Figure 4.11: Raw disk throughput (MBytes/sec) for sequential (left) and random (right) workloads.

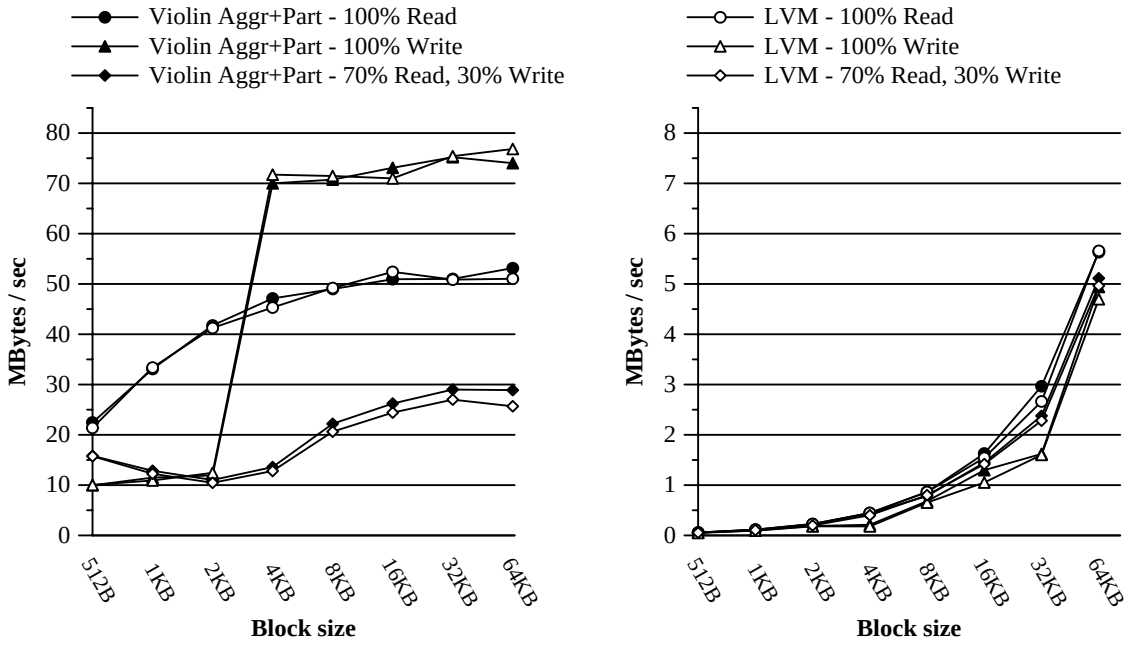
4.4.3 Performance

To evaluate the performance of Violin we use two well-known I/O benchmarks, Iometer [Iometer] and PostMark [Katcher]. Iometer is a benchmark that generates and measures I/O workloads with various parameters. PostMark is a synthetic benchmark that emulates the operation of a mail server. PostMark runs on top of a filesystem, while Iometer runs on raw block devices.

The system we use in our evaluation is a commodity x86-based Linux machine, with two AMD Athlon MP 2200+ CPUs, 512 MB RAM, three Western Digital WDC WD800BB-00CAA1 ATA Hard Disks with 80 GB capacity, 2MB cache, and UDMA-100 support. The OS is Red Hat Linux 9.0, with RedHat's latest 2.4.20-31.9smp kernel.

First, we examine the performance of single Violin modules compared to their Linux driver counterparts. We use three different configurations: raw disk, RAID, and logical volume management. In each configuration we use layers implemented as a block driver under Linux and as modules under Violin:

A. The raw Linux disk driver (Disk) vs. a *pass-through* module in Violin (Disk Module). The same physical disk partition is used in both cases.



IOmeter throughput for Violin Aggregation+Partition vs. LVM for 2 Striped Disks (32K stripe)

Figure 4.12: LVM throughput (MBytes/sec) for sequential (left) and random (right) workloads.

B. The Linux Volume Manager driver (LVM) vs. a dual module setup in Violin using the partitioning and aggregation modules (Aggr.+Part. Modules). In the Violin setup we use two disks that are initially aggregated into a single resizable striped volume. On top of this large volume we create a partition where the filesystem is created. In LVM we use the same setup. We initialize the same disks into physical volumes, create a volume group consisting of the two disks, and finally create a striped logical volume on top. The size of the stripe block is 32 KBytes in both setups.

C. The Linux MD driver (MD RAID-0 & 1) vs. the Violin RAID module (RAID-0 & 1 Module). Both modules support RAID levels 0, 1, and 5. We compare performance of RAID levels 0 (striping) and 1 (mirroring) for MD and Violin. In both setups we use the same two disks and a stripe block of 32 KBytes for RAID-0. Note that the system setup in the RAID-0 case is identical to the LVM experiments, with the exception of different software layers being used.

Iometer

Iometer [Iometer] generates I/O workloads based on specified parameters (e.g. block size, randomness) and measures them on raw block devices. We vary three parameters in our workloads: (i) *Access pattern*: we use workloads that are either sequential or random. (ii) *Read-to-write ratio*: we explore three ratios,

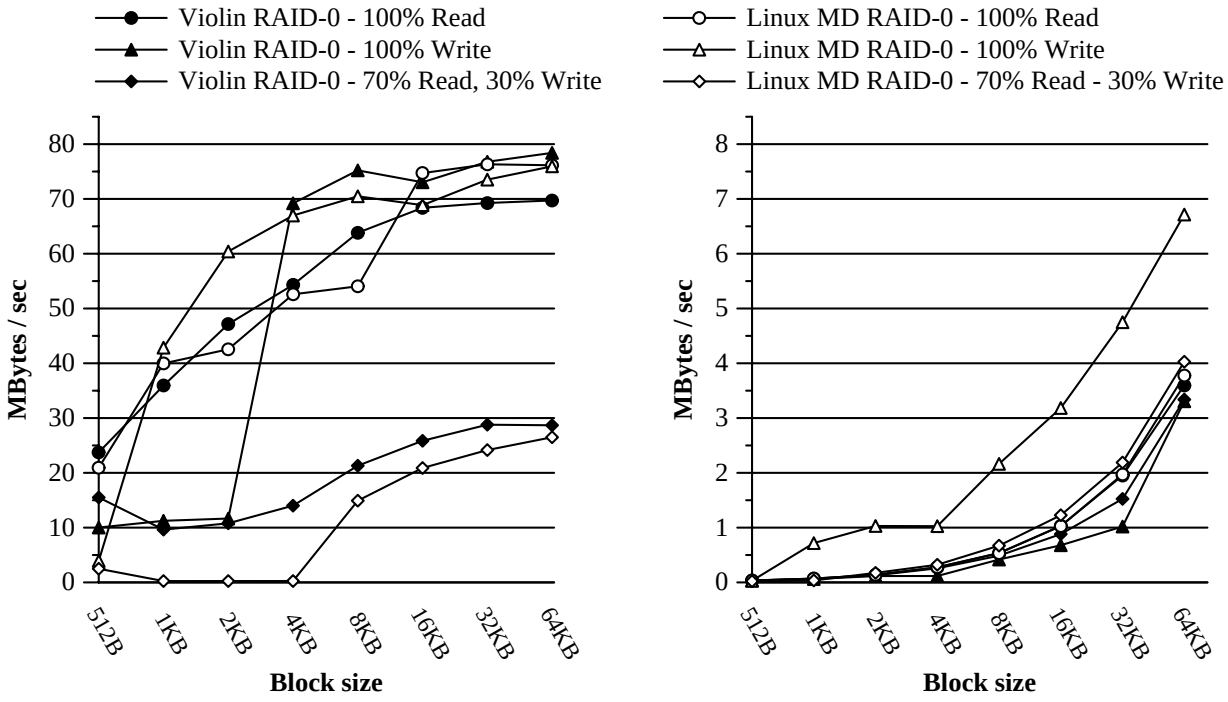


Figure 4.13: RAID-0 throughput (MBytes/sec) for sequential (left) and random (right) workloads.

100% reads, 100% writes and a mix of 70% reads - 30% writes. (iii) *Block size*: we use 512 byte to 64 KByte block sizes.

Figures 4.11-4.14 summarize our results with Iometer version 2004.07.30. Figure 4.11 compares the performance of a pass-through to disk Violin module to the plain system disk. Figures 4.12, 4.13 and 4.14 compare the performance of Violin modules to their kernel counterparts, LVM and MD RAID levels 0 and 1. The left column graphs represent sequential I/O performance, while the right graphs show random I/O performance. In most cases we observe that the performance difference between similar functionality modules is less than 10%. There are cases when Violin has a bit better performance than the kernel drivers and cases when it performs a little worse.

The larger performance differences observed are: (i) a low sequential performance of LVM and Violin modules for block sizes less than 4KB. This is due to their default block size for their device drivers, which is set to 4KB, compared to 1KB for raw disk and the MD driver. This explains the steep performance increase for Violin and LVM that is consistently observed between block sizes of 2KB and 4KB. Using a smaller system block size on Violin is possible, but requires changes to the current Violin code. (ii) MD has higher performance for random write operations, both in RAID-0 and RAID-1 modes.

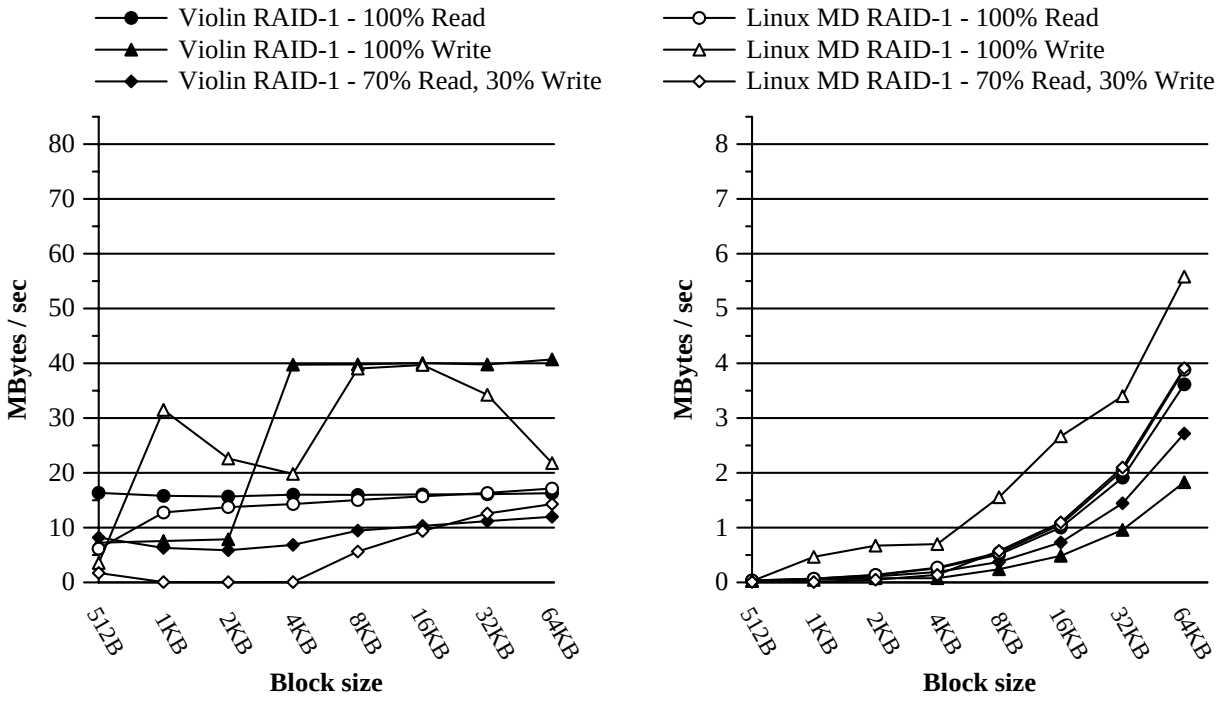


Figure 4.14: RAID-1 throughput (MBytes/sec) for sequential (left) and random (right) workloads.

This is due to MD's optimized buffer cache management, which caches writes and can more effectively cluster write requests. This is mainly a Linux implementation issue and we consider improving buffer cache management in Violin's driver as future work.

PostMark results

PostMark [Katcher] creates a pool of continually changing files on a filesystem and measures the transaction rate for a workload approximating a large Internet electronic mail server. Initially a pool of random text files is generated, ranging in size from a configurable low bound to a configurable high bound. Once the pool has been created, a specified number of transactions occurs. Each transaction can be one of two pairs of I/O operations: (i) create file or delete file and (ii) read file or append file. Each transaction type and its affected files are chosen randomly. When all of the transactions have completed, the remaining active files are deleted.

To evaluate the performance of each configuration, we use a workload consisting of: (i) files ranging from 100KBytes to 10MBytes in size, (ii) an initial pool of 1000 files, and (iii) 3000 file transactions with equal biases for read/append and create/delete. Each run is repeated five times on the same filesystem

to account for filesystem staleness and the results are averaged over the five runs. During each run there are over 2500 files created, about 8.5GBytes of data read and more than 15GBytes of data written.

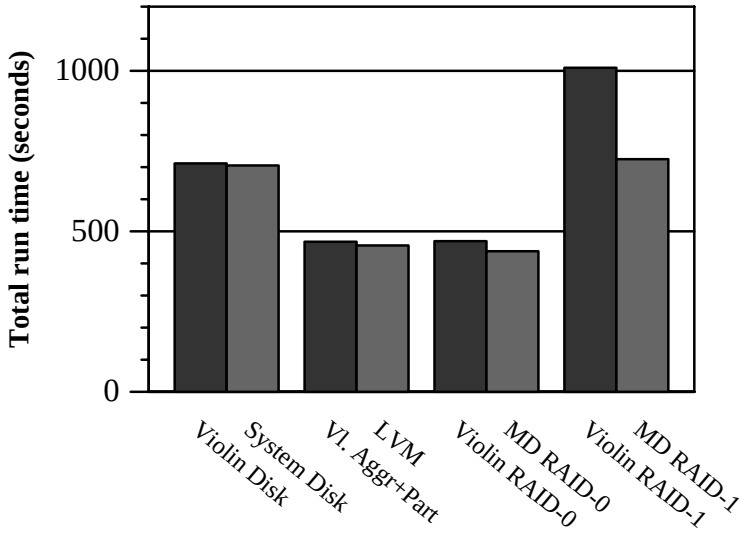


Figure 4.15: PostMark results: Total runtime

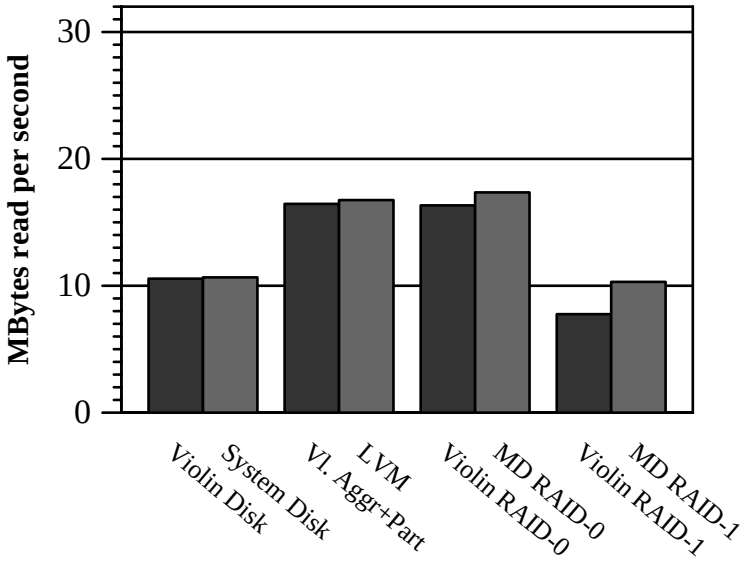


Figure 4.16: PostMark results: Read throughput

Figures 4.15, 4.16 and 4.17 show our PostMark results. In each graph and configuration, the light bars depict the kernel device driver measurements, while the darker bars show the numbers for Violin modules. The bars have been grouped according to each system configuration and PostMark experiment. In all cases, except RAID-1, there is a small difference, less than 10% between the two systems, Linux

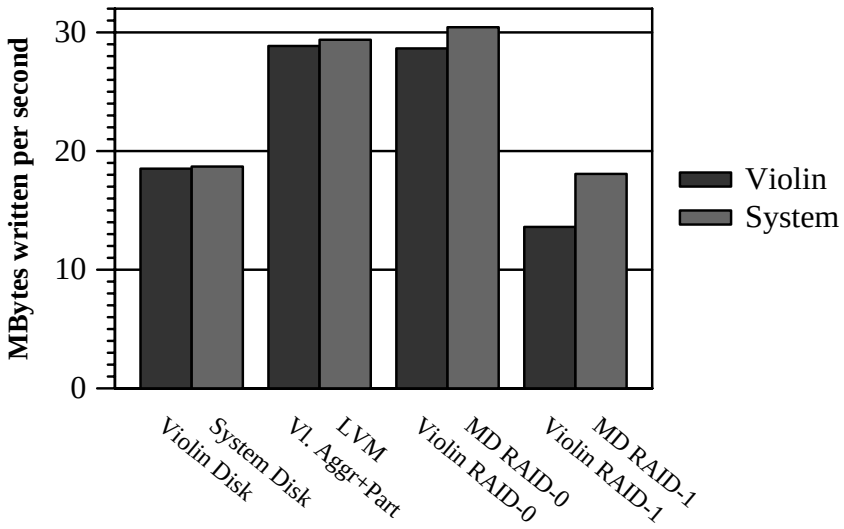


Figure 4.17: PostMark results: Write throughput

driver vs. Violin module. In the RAID-1 case there is a large difference of about 30%, due again to the better buffer-cache management that MD has. As mentioned before, we are currently improving this aspect of Violin's driver.

4.4.4 Hierarchy Performance

Finally, for the hierarchy of Figure 4.10, we evaluate with Iometer the performance of the hierarchy as each module is introduced. Our goal is to provide some intuition on expected system performance with complex configurations. Figures 4.18 and 4.18 show the results for sequential (left) and random (right) workloads. Each curve corresponds to a different configuration as each module is introduced to the hierarchy. In the sequential workloads, we note that variations in performance occur depending on the functionality of the layers being introduced. We see a large performance reduction in the versioning module, which is due to the disk layout that this module alters, as discussed in Chapter 3. Encryption overhead on the other hand, appears to be about 10-15%. In random workloads we observe that the hierarchy performance is influenced only by the versioning layer functionality, which can increase performance substantially. This happens because of the change in the disk layout that versioning incurs, through write request logging, as shown also in Clotho's results (Section 3.4). Encryption or RAID-0, on the other hand, do not seem to influence performance for random I/O.

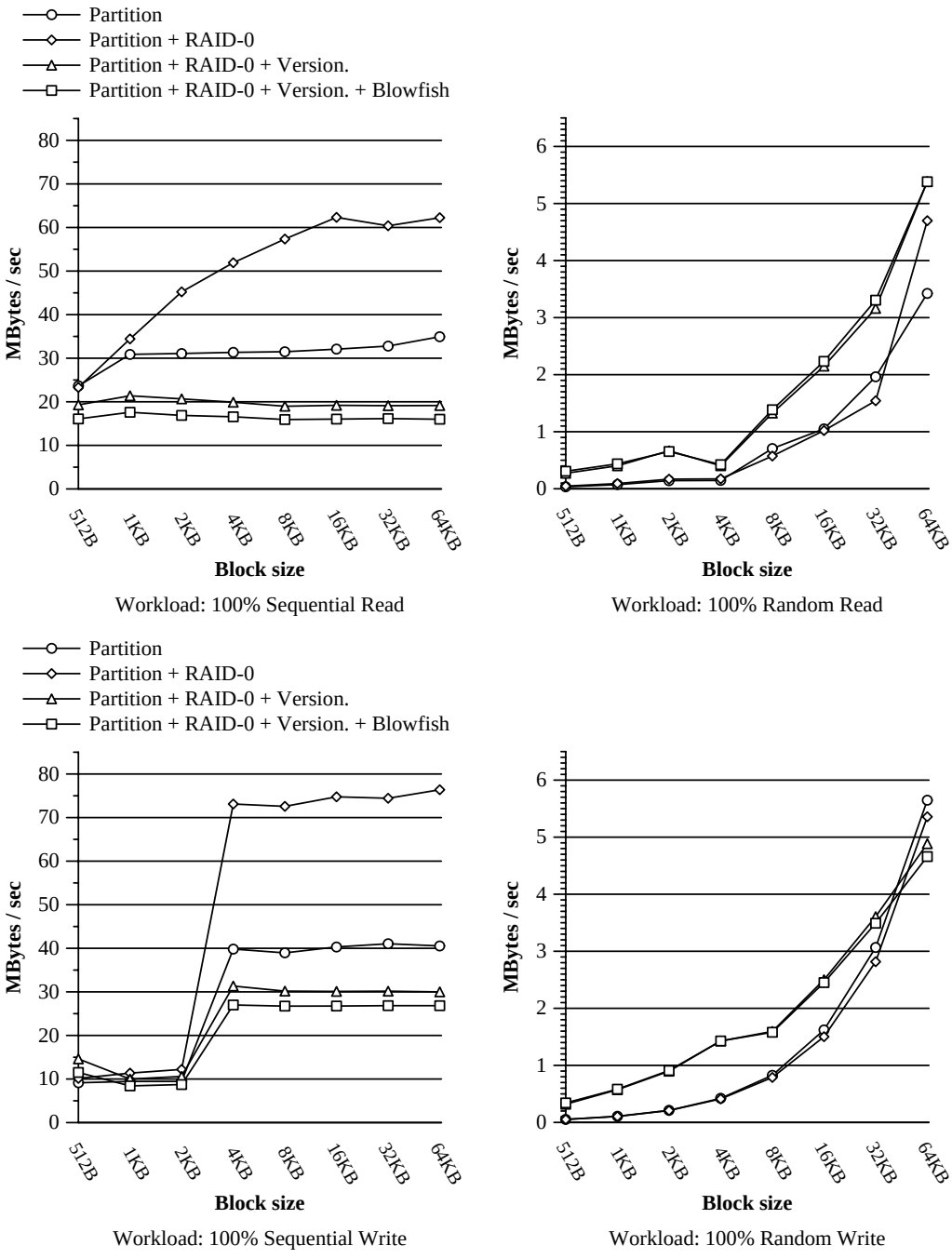


Figure 4.18: IOmeter throughput (read and write) for hierarchy configuration as layers are added.

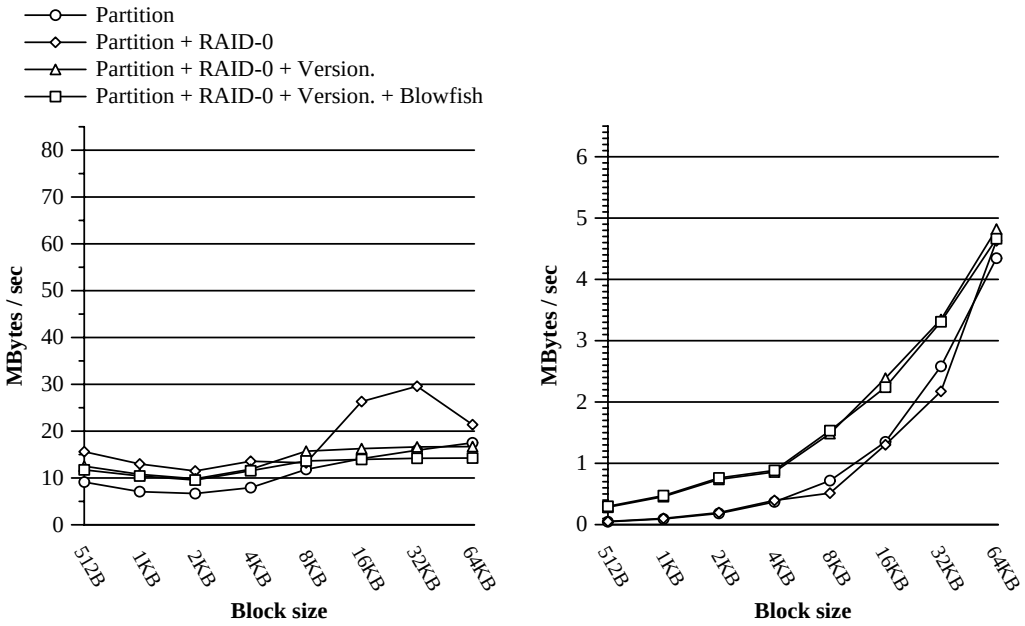


Figure 4.19: IOmeter throughput (70% read and 30% write mix) for hierarchy configuration as layers are added.

4.5 Limitations and Future work

Overall, Violin offers a highly configurable environment to easily experiment with advanced storage functionality. One drawback of our work is that we have evaluated Violin with a specific set of modules. However, we believe that this set is broad enough to demonstrate the benefits of our approach. Further extensions to Violin are possible as we gain more experience with its strengths and limitations.

A current limitation of Violin is that it does not support "upcalls", or calls from a lower layer in the hierarchy to a higher. This stems from the traditional I/O stack and the block-level API, where control and initiation of requests proceeds in a downward fashion. That is, the application and/or the file system initiate I/O commands and query or modify the status of the underlying layers. We believe that in order for the block-level subsystem to implement advanced functionality, a more balanced bi-directional scheme is necessary. In other words, the block-level subsystem should be able to send commands that query and/or modify the status of higher layers in the I/O stack, such as the file system.

Also, this work is focused on examining mechanisms, rather than policies for building virtual I/O hierarchies. Since we can extend the I/O hierarchy with a rich set of mechanisms, it is important to examine how user requirements can be mapped to these mechanisms automatically [Keeton and Wilkes,

2003]. An interesting aspect on this research direction is to answer the question of how much a system can do in terms of optimizations in this translation process, both statically when the virtual I/O hierarchy is built [Wilkes, 2001], but mostly dynamically during system operation.

Finally, it is interesting to examine how Violin can be extended to include the network path in virtual hierarchies. Networked storage systems offer the opportunity to distribute the virtual I/O hierarchy throughout the path from the application to the disk. We explore this topic in the next chapter along with the various possibilities and trade-offs in order to provide insight on how virtualization functionality should be split among components in storage systems.

4.6 Conclusions

In this chapter we design, implement, and evaluate Violin, a virtualization framework for block-level disk storage. Violin allows easy extensions to the block I/O hierarchy with new mechanisms and flexible combining of these mechanisms to create modular hierarchies with rich semantics.

To demonstrate its effectiveness we implement Violin within the Linux operating system and provide several I/O modules. We find that Violin significantly reduces implementation effort. For instance, in cases where user-level library code is available, new Violin modules can be implemented within a few hours. Using a simple user-level tool we create I/O hierarchies that combine the functionality of various modules and provide a set of features difficult to offer with monolithic block-level drivers. Finally, we use two benchmarks to examine the performance overhead of Violin over traditional, monolithic drivers and driver-based hierarchies, and find that Violin modules and hierarchies perform within 10% of their counterparts.

Overall, we find that our approach provides adequate support for embedding powerful mechanisms in the storage I/O stack with manageable effort and small performance overhead. We believe that Violin is a concrete step towards supporting advanced storage virtualization, reducing storage management overheads and complexity, and building self-managed storage systems.

Chapter 5

Orchestra: Extensible Block-level Support for Resource and Data Sharing in Networked Storage Systems

A man who wants to lead the orchestra must turn his back on the crowd.

—Max Lucado

The content of this chapter roughly corresponds to publication [Flouris et al., 2008].

5.1 Introduction

Sharing in a storage system refers usually to two distinct aspects of the system: (i) physical resources and (ii) access to data. Traditionally both of these functions have been supported by the file-system, which has been responsible for pooling the available resources (mainly disks in a directly-attached storage system) and coordinating read/write access to stored data. Over time, sharing of physical resources has been increasingly supported by adding management and configuration flexibility at the block-level. Today's storage area networks allow multiple hosts to share physical resources by placing multiple virtual disks over a single pool of physical disks. However, current levels of resource sharing incur significant limitations; For instance, logical volumes in a Fibre-Channel storage system should not span multiple storage controllers. Besides, distinct application domains have very diverse storage requirements; Systems designed for the needs of scientific computations, data mining, e-mail serving, e-commerce,

operating system (OS) image serving or data archival impose different trade-offs for a storage system in terms of dimensions such as speed, reliability, capacity, availability, security, and consistency. Yet, current block-level systems provide only a limited set of functions, for instance they rarely support versioning or space saving techniques.

In terms of data sharing, e.g. concurrently accessing data in a single logical disk from many application clients, storage systems employ a (distributed) file-system to coordinate data accesses. Such systems require mechanisms for distributed coordination, allocation, and metadata consistency, which have proved extremely challenging to scale at the file-system level.

Finally, combining mechanisms for resource and data sharing in the file-system results in unmanageable storage systems that are both hard to tailor to application needs as well as to scale. Thus, although this emerging, decentralized cluster architecture for networked storage offers a lot of potential for improving scalability and reducing cost, it needs to support efficiently both resource and data sharing.

In this chapter, we present Orchestra, a low-level software framework that extends Violin, presented in the previous chapter, and supports resource and data sharing at the *block level* in a decentralized storage cluster. Our main goal is to ease of development, deployment, and adaptation of block-level networked-storage services for various environments and application needs. Orchestra treats data and control requests in a uniform manner, propagating all I/O requests through the same I/O path from the application to the physical disks. Thus, it essentially eliminates out-of-band services, such as lock servers and block allocation that complicate the design of distributed (clustered) storage systems. We show that providing flexibility does not introduce significant overheads and does not limit scalability. Moreover, our propositions can adapt to high requirements in terms of important concerns such as reliability (fault tolerance, data integrity), which is explored in the next chapter and manageability (dynamic reconfiguration, automatic tuning) which is outside of the scope of the present thesis.

The main contributions of work presented in this chapter can be summarized as follows:

- We provide a novel approach for building cluster storage systems which consolidates system complexity in a single point, the Orchestra hierarchies. We examine how extensible, block-level I/O paths can be supported over distributed storage systems. Our proposed infrastructure deals with metadata persistence and allows for hierarchies to be distributed almost arbitrarily over application and storage nodes, providing a lot of flexibility in the mapping of system functionality to available resources.

- We examine how sharing can be provided at the block level. We design block-level locking and allocation facilities that can be inserted in distributed I/O hierarchies where required. A distinguishing feature of our approach is that allocation and locking are both provided as in-band mechanisms, i.e. they are part of the I/O stack and are not provided as external services. We believe that our approach simplifies the overall design of a storage system and leaves more flexibility for determining the most adequate hardware/software boundary. Moreover, using in-band control operations takes advantage of available support for scaling regular data requests. Our current prototype provides simple but scalable policies for locking and allocation. More involved policies can be implemented through additional modules.

- We examine how higher system layers can benefit from an enhanced block interface. Orchestra's support for block locking, allocation, and metadata persistence can significantly simplify filesystem design. We design and implement a *stateless, pass-through* filesystem (ZeroFS or `0FS`), that takes advantage of Orchestra's advanced features and distributed virtualization mechanisms. In particular, this filesystem provides basic file and directory semantics and I/O operations, while it does not maintain any internal distributed state and does not require explicit communication among its instances.

To demonstrate our approach, we implement Orchestra as a block device driver under Linux and ZeroFS as a user-level library. We perform experiments with a cluster of 16 nodes (8 storage and 8 application nodes) interconnected with Gigabit Ethernet. We evaluate the effectiveness of our approach by examining the overheads introduced by the block level extensions for locking and block allocation, the overheads associating with providing a stateless file system, and the scalability of the system at both the block and filesystem level.

The rest of the chapter is organized as follows. Section 5.2 presents the design of Orchestra, emphasizing its main contributions, whereas Section 5.3 discusses our prototype implementation. Section 5.4 presents our results. Section 5.5 discusses limitations and future work and Section 5.6 draws our conclusions.

5.2 System Design

Orchestra aims at providing the underlying infrastructure for building scalable and extensible storage systems to support a wide range of application needs in a cost-effective manner.

Orchestra is designed around three goals:

- **Distributed virtual hierarchies:** Orchestra uses the concept of *distributed* virtual hierarchies to allow storage systems to extend the functions they support and to provide different views and semantics to applications without compromising scalability. New virtual modules may be used to provide novel storage functions while maintaining the illusion of a single data store to the upper layers (filesystem and applications). This requires "splitting" hierarchies over the network in a transparent manner, and in particular, transferring *control requests* among nodes.

- **Distributed block-level sharing:** Orchestra allows applications to share its virtual devices. To facilitate sharing Orchestra incorporates block-level locking and allocation mechanisms that are implemented as virtual modules. Essentially, this makes locking and allocation in-band operations, eliminating out-of-band services that are usually used in cluster storage systems (e.g. Petal, FAB). The locking mechanism is integrated in the virtual hierarchies as an optional virtual module and may be used by Orchestra devices to lock shared metadata or by applications that share data at the block-level. The block allocator performs distributed free-block management and is also built as an optional virtual device that may be inserted at appropriate places in a virtual hierarchy.

- **Distributed file-level sharing:** To support seamless, coherent sharing for distributed, file-based applications, Orchestra provides a simple user-level library, ZeroFS (`0FS`) that implements a conventional filesystem API. Since Orchestra already includes advanced functions, `0FS` is essentially a *stateless, pass-through* filesystem that translates application calls to the underlying block-level operations.

Orchestra allows users to create distributed hierarchies that span application and storage nodes, based on application and system requirements. A sample distributed Orchestra hierarchy is shown in Figure 5.1. Next, we discuss each of the above issues in more detail.

5.2.1 Distributed Virtual Hierarchies

Orchestra builds on Violin, presented in Chapter 4. Violin is a framework developed for single node storage virtualization, and provides all the mechanisms required to support distributed and shared storage systems. Next we discuss the extensions to Violin developed in the context of Orchestra.

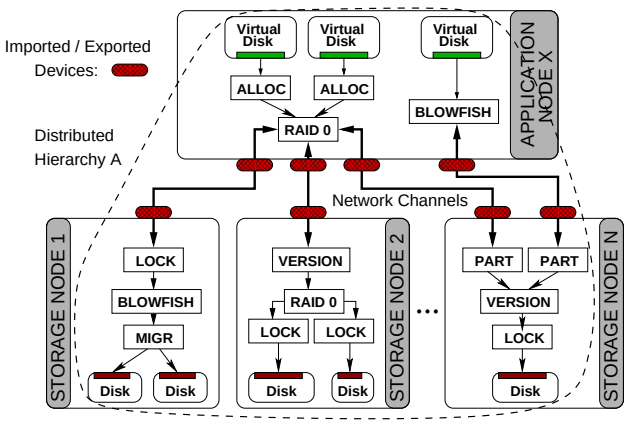


Figure 5.1: A distributed Orchestra hierarchy.

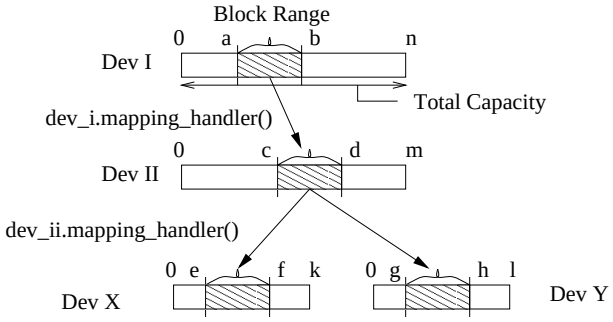


Figure 5.2: Byte-range mapping process through a hierarchy.

In-band control and data requests

As mentioned earlier, Violin supports control and data I/O requests in the same manner, by using the same in-band mechanism for data and control propagation in virtual hierarchies. An issue with both data and control requests that traverse virtual hierarchies is related to how we treat block address arguments. Each layer is able to "see" and thus, reference only the block addresses of its underlying layers. Thus, as control requests propagate in the virtual hierarchy the addresses of the blocks they reference, need to be translated similar to I/O requests.

We have generalized this concept in Orchestra by allowing layers to provide address-mapped control requests (commands) in addition to existing regular control requests. All control requests may be issued by any virtual layer and traverse the virtual hierarchy top to bottom. If a layer does not understand a specific control request it forwards it to its lower layers. Eventually, a control command reaches a layer that handles it, possibly generating or responding to other control and data requests. Control requests provided by individual layers in Orchestra hierarchies are automatically inherited by *all* higher layers in

the hierarchy. When a control request is issued, it is either handled by the current layer in the hierarchy or automatically forwarded to the next layer(s).

In addition, address-mapped commands are subject to translation of arguments that represent byte-ranges. This is achieved by augmenting each layer, i.e., extending the layer API, with an address-mapping API call, `address_map()`. This call is written by module developers for every module that is loaded in an Orchestra hierarchy and translates an input byte-range to any output byte-range(s) in one or more output devices as shown in Figure 5.2.

Block mapping depends on the functionality of individual modules. Although complex mappings are possible, in many cases, mappings are simple functions. Implementing the address-mapping method is a straightforward task for the module developer, since this is usually identical to the mappings used for read and write I/O calls through the layer to the output devices.

Finally, to allow virtual hierarchies to span multiple nodes, Orchestra needs to transfer I/O requests between virtual modules that execute on different nodes. For this purpose, we currently use a simple network protocol over TCP/IP that implements a network block device that is able to handle both data read-write requests as well as control requests between system nodes.

5.2.2 Distributed Block-level Sharing

Sharing virtual volumes requires coordinating (i) accesses to data and, as mentioned above, metadata via mutual exclusion and (ii) allocation and deallocation of storage space. In Orchestra, concurrent accesses to the data are coordinated by means of a locking virtual module, whereas space allocation is managed by a distributed block allocation facility.

Byte-range locking

As mentioned, Orchestra provides support for byte-range locking over a distributed block volume. The main metadata in the locking layer is a a free-list that contains the *unlocked* ranges of the managed virtual volume. When a lock control request arrives, the locking layer uses its internal metadata to either complete or block the request. At an unlock request the locking layer updates its metadata and possibly unblocks and completes a previous pending lock request. Our locking API currently supports locks in both blocking and non-blocking (i.e. trylock) modes.

Support for byte ranges, instead of block ranges, is provided to facilitate the implementation of multiple-readers, single-writer locks at the filesystem (e.g. `OFS`) or application level. A simple algo-

rithm to implement such file locks, used currently by the `OFS` filesystem is as follows: to lock a file in multiple-reader mode, `OFS` locks just one byte in the inode block address range for each reader. This allows as many concurrent readers, as the number of bytes in the file inode, which is currently 4096. In the case of a writer the filesystem locks the whole inode block address range, thus gaining exclusive access to the file.

Lock and unlock requests are address-mapped commands, which allows us to distribute the locking layers to any desirable serialization point in a distributed hierarchy. However, to achieve mutual exclusion, locks for a specific range of bytes should be exclusively serviced by a single locking virtual layer. This is achieved by placing locking layers at specific points in the distributed hierarchy, where they can intercept all I/O requests for a part of the storage address space. Such points include for example places where a local device is exported from a storage node, as shown in Figures 5.4 and 5.5. A locking device in this case allows locking support for any remote layer using this exported device. The lock and unlock commands are forwarded to the specific node through Orchestra's communication mechanism and the associated addresses are mapped according to the distributed hierarchy mappings. Additionally, partitioning the locking address space and allowing many locking layers to service separate byte ranges in parallel, achieves locking parallelization and load balancing across multiple nodes. Furthermore, the locking service scales as more storage nodes are added.

Note that the metadata of a locking layer does not need to be persistent. Instead, a lease-based mechanism to reclaim locks from a failed client is adequate. Finally, lock availability can be achieved through mere replication of storage nodes, in the same way that one would configure a storage hierarchy for increased data availability.

Block allocation

The role of Orchestra's block allocator is to handle distributed block management in a consistent manner for clients sharing the same block volume. The allocator distributes free blocks to the clients and maintains a consistent view of used and free blocks. All such block-liveness information is maintained by the allocator, offloading all the potentially complex free-block handling code from higher system and application layers.

In addition, providing a space allocation mechanism at the block level allows Orchestra to handle (dynamic) reconfiguration in an easy way: the allocation policy and size of a volume can be modified transparently to higher layers. Besides, the allocator's block-liveness state can be used to make

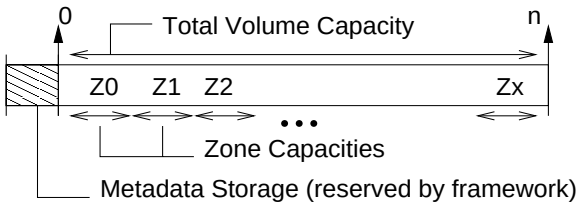


Figure 5.3: View of the allocator layer's capacity.

necessary management tasks, such as backup and migration, more (time- and space-) efficient.

The allocator metadata for managing free blocks consist of free-lists and bitmaps to handle blocks of various sizes. The allocator uses a free-list for keeping track of large blocks and bitmaps for small blocks. Currently, the allocator is configured to use two block sizes, a large block size (4 KBytes) for data and a small block size (256 Bytes) for filesystem i-nodes. However, we envision that future versions of Orchestra may rely on a single block size using the notion of "stuffed i-nodes" [Mullender and Tanenbaum, 1984].

The allocator metadata need to be maintained consistent across allocator instances running in different nodes. This is achieved by using the persistent metadata locking primitives to synchronize access to the shared free-list and bitmaps. Frequent locking at fine granularity will result in high allocation overheads. To address this issue we amortize the overhead associated with locking metadata by dividing the available (block) address space of a shared volume in a sufficiently large number of allocation zones as shown in Figure 5.3. Each zone is independent and has its own metadata, which can be locked and cached in the node using this particular zone.

The locking algorithm works as follows. When the first allocation request for a free block arrives, the allocator identifies an unused (and unlocked) zone in the shared device and locks the zone. When a module successfully locks a zone, it loads the zone's metadata in memory. Subsequent allocation requests will use the same zone, as long as there are available free blocks, thus, avoiding locking overheads. When a zone does not have enough free blocks to satisfy a request, a new zone is allocated and locked. This scheme essentially increases the locking granularity to full zones. The metadata of locked zones are automatically synchronized to stable storage, similarly to all other module metadata in Violin, in two occasions: (i) periodically every few seconds and (ii) when a zone is unlocked and its metadata released from the cache.

Locking full zones of free blocks for allocation purposes results in internal fragmentation. We expect that in future storage systems with large amounts of physical disk space this will not be an important

issue. To minimize internal fragmentation there is a need to select carefully the number of zones in a shared volume. Using a large number of zones may reduce internal fragmentation but increase locking frequency (and thus allocation) overhead, whereas a small number of larger zones may have the opposite effect. We believe that the number of zones for a volume should be calculated according to the expected number of users of a shared volume and we intend to determine acceptable ratios through experiments with realistic workloads.

Freeing blocks can be more complex than block allocation. If the blocks to be freed are in a locally locked zone, the module will free the blocks in the local metadata maps. If, however, the block belongs to a zone not locally locked, the module will first attempt to lock the corresponding zone in order to free the blocks. If it is successful, it will free the blocks and will keep the zone locked for a short time to avoid the locking penalty of successive free operations. Even though this case may incur high overhead, we expect that block deallocation will be clustered in zones due to the allocation policy and the (expected) large zone sizes, thus, minimizing deallocation overheads.

If, during free operations, the allocator fails to lock a specific zone, it uses small logs for *deferred free operations*, called *defer logs*. The defer log for a zone is a persistent metadata object. When an allocator cannot lock a zone for deallocation purposes, it locks the zone's defer log, appends the pending free operation, and releases the defer log lock in case other allocator modules run into the same situation. If the defer log is full or already locked, the allocator module waits until the log is emptied or unlocked. The responsibility for processing the defer log lies upon the allocator that has locked the corresponding zone. Every allocator module that owns a lock on a zone, periodically checks the defer log for pending deallocation requests. If there are any pending operations in the log, they are performed on the locked metadata and the defer log is emptied.

An mentioned in Section 2.3, our design of the block allocator bears similarity with Hoard [Berger et al., 2000], a scalable memory allocator for multi-threaded applications.

5.2.3 Distributed File-level Sharing

Many applications access storage through a filesystem interface. To allow such applications to take advantage of the advanced features of Orchestra and to demonstrate the effectiveness of our volume sharing mechanisms there is a need to provide a filesystem interface on top of Orchestra.

One approach to achieve this is to use an existing distributed file system. Depending on the requirements of the distributed filesystem, Orchestra can be used to either provide a number of virtual volumes

each residing in a single storage node [Preslan et al., 1999], or a single distributed virtual volume built out of multiple storage nodes [Thekkath et al., 1997]. However, existing filesystems in either case would not take advantage of distributed block allocation and locking primitives provided by Orchestra at the block-level.

For these reasons, we provide our own filesystem on top of Orchestra, using a user-level library that provides the basic filesystem functions, such as file and directory naming, as well as standard file I/O operations[1]. `0FS` is a *stateless, pass-through* file system that translates file calls to the underlying Orchestra block device. Our approach demonstrates also that by extending the block layer we are able to significantly simplify filesystem design in distributed environments and especially when it is necessary to support system extensibility. Thus, the block allocator uses the block volume facilities for free-list allocation and locking.

The main feature of ZeroFS is that, unlike distributed file systems, it does not require explicit communication between separate instances running on different application nodes. Usually, communication is required for two purposes: (i) mutual exclusion and (ii) metadata consistency. `0FS` uses the corresponding block-level mechanisms provided by Orchestra volumes for these purposes:

- **Mutual exclusion:** `0FS` uses the byte range locking mechanism provided by Orchestra to achieve mutual exclusion between multiple applications accessing a single filesystem through one or more application nodes. `0FS` implements multiple-reader, single-writer file locks as described in Section 5.2.2. Currently, `0FS` locks and unlocks files during the `open / close` calls, which is coarser than locks on read/write operations, but realistic enough for many applications. Support for locking on reads/writes, is nonetheless important for clustered applications that heavily rely on shared files. We expect to introduce these features in a future version without significant effort.

- **Metadata consistency:** The only metadata required by `0FS` are i-nodes and the directory structure. To avoid maintaining internal consistent state in memory, `0FS` does not perform any caching of metadata or data but uses the underlying `0FS` block device. Thus, accesses to files or directories in `0FS` may result in multiple reads to the underlying block device for the corresponding i-nodes and directory blocks. In the worst case, four read requests may be necessary for very large files.

[1]Most of the conventional filesystem operations are supported, such as open(), close(), remove(), creat(), read(), write(), stat(), rename(), mkdir(), rmdir(), opendir(), readdir(), chdir(), getcwd(), etc.

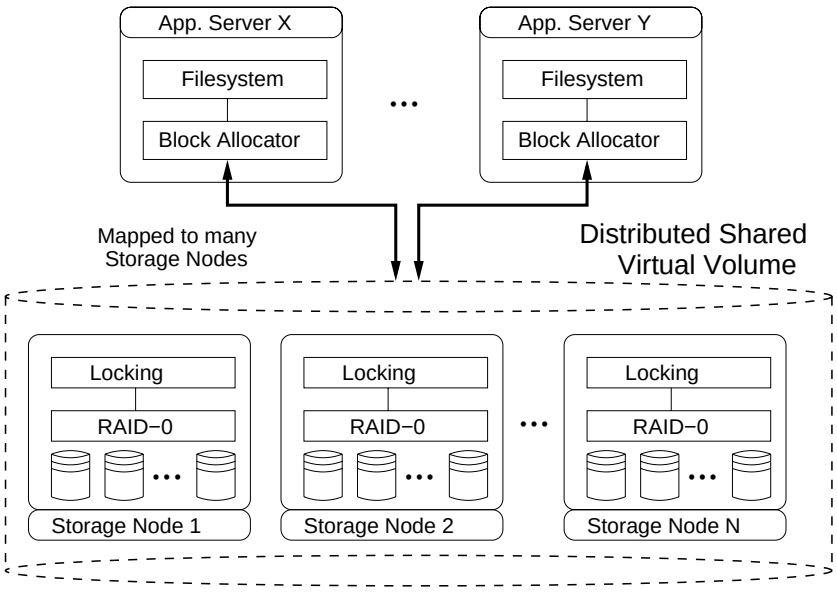


Figure 5.4: The allocator and locking layers allow many FS clients to share a distributed volume mapped on many nodes.

Figure 5.4 shows how `OFS` is combined with Orchestra to provide file-level sharing over distributed Orchestra volumes.

5.2.4 Differences of Orchestra vs. Violin

Violin, presented in Chapter 4, and Orchestra presented in this chapter are both virtualization frameworks and thus share basic virtualization principles, such as easily extensible hierarchies made of virtual devices. However, there are significant differences between the two systems, the most important being that Violin offers single-node virtualization, while Orchestra is a distributed virtualization framework for a cluster.

In terms of design/implementation, the major differences of Orchestra and Violin are: (i) The networking layer that allows hierarchy distribution in the cluster. (ii) Extensions to command API for propagating commands across nodes and manipulating control and configuration information. (iii) Support of block-level locking and allocation in distributed configurations, which is essential for sharing volumes between many clients. (iv) Design, implementation, and evaluation of a stateless filesystem (`OFS`) built on top of distributed Orchestra hierarchies using block-level locking and allocation facilities. To our knowledge, Orchestra is the only existing system that decouples free-block allocation and locking from the distributed filesystem and integrates them as independent components in the block-level I/O

stack for improving scalability.

5.3 System Implementation

Orchestra is implemented as a loadable block device driver module in the Linux 2.6 kernel, extending Violin's device driver presented in Chapter 4. User-level tools can be used online, during system operation to create new or modify existing I/O hierarchies. However, the user is responsible for maintaining data consistency while modifying the structure of virtual hierarchies. The user-level tools communicate with the framework device driver through the kernel's `ioctl()` interface.

Virtual I/O layers are implemented as separate kernel modules that are loaded on demand. Upon loading, Orchestra layers register their presence to the Orchestra framework driver. These modules are not "classical" device drivers themselves, but rather use the framework's programming API, which extends Violin's API. Modules implemented for Violin, such as RAID 0 and RAID 1 (including recovery), versioning, device partitioning, device aggregation, DES and triple-DES encryption, and on-line data migration between devices have been ported to Orchestra. New modules, such as the block allocator, byte range locking and I/O request tracing have also been implemented.

The networking channels for exporting Orchestra devices to other storage nodes or clients are currently implemented over TCP/IP in the core Orchestra framework. The networking component implements the client and server kernel threads and the request queues needed for the networking channels, as well as the Orchestra commands to connect and disconnect remote devices from the local Orchestra stack.

When a virtual I/O hierarchy is configured in the framework and linked with a `/dev` device, it can be manipulated as a normal block device with existing tools. For instance, the user can build a file system on top of it with `mkfs` and then `mount` it as a regular single-node filesystem or share it through `0FS`.

ZeroFS is currently implemented as a user-level library that may be linked with any application during execution. We chose to implement `0FS` at user-level to reduce the development effort as well as to allow for easy customization. `0FS` currently supports most of the conventional filesystem operations, such as open, close, remove, creat, read, write, stat, rename, mkdir, rmdir, opendir, readdir, chdir, getcwd. It does not support the full functionality of a kernel-level filesystem, such as fcntl, file permission and attribute operations, and memory-mapped files. We were, however, able to compile and run standard filesystem benchmarks, such as IOzone and PostMark without any code modifications.

Kernel Component	# code lines
Orchestra Core Framework	19558
RAID (0,1 levels)	950
Partition & Aggregation	1546
Versioning (Clotho)	800
DES & 3DES Encryption	1550
Blowfish Encryption	791
Migration	442
Networking	3033
Byte-range Locking	1137
Block Allocator	2479
Kernel Code (approx.)	32300
`OFS` (user-space)	7600
Orchestra Total (approx.)	40000

Table 5.1: Orchestra modules in kernel and user-space code lines.

Table 5.1 shows the size of the code for Orchestra. The current version, as used in our experiments, is about 40000 lines of code.

Finally, the current Orchestra implementation uses a number of kernel threads mainly for network channels between devices: (a) One thread that listens, authenticates and accepts new connections from remote Orchestra hierarchies. This thread also creates the threads needed for a new remote device connection. (b) A client request sender thread that removes I/O requests or control command requests from a FIFO queue and sends them to the appropriate server. (c) A server request receiver that receives requests from all client connections. (d) A server response sender thread that sends responses when the I/O requests or I/O control commands are done. (e) A client request receiver thread that receives server responses and sends call-backs to the waiting application(s).

Our lock module provides a simple API that allows locking block ranges. Both the allocator and lock components are implemented as virtual modules.

The commands currently supported by the allocator layer are:

- `get_block_size()`: returns the large block size.

- `get_small_block_size()`: returns the small block size.
- `allocate_blocks()`: returns the requested number of sequential blocks allocated, either large or small blocks.
- `free_blocks()`: frees the requested number of sequential blocks, either large or small blocks.

5.4 Experimental Results

In this section we present preliminary results on the overhead of basic operations in Orchestra and we examine the scalability of Orchestra and `OFS` on a setup with multiple storage and application nodes.

Our evaluation platform is a 16-node cluster of commodity x86-based Linux systems. All the cluster nodes are equipped with dual AMD Opteron 242 CPUs and 1 GByte of RAM, while storage nodes have four 80GB Western Digital SATA Disks. All nodes are connected with a 1 Gbit/s (Broadcom Tigon3 NIC) Ethernet network through a single 48-port GigE switch (D-Link DGS-1024T). All systems run Fedora Core 3 Linux with the 2.6.12 kernel.

We use three I/O benchmarks: xdd [Xdd], IOzone [Norcott and Capps] and PostMark [Katcher]. We use PostMark and IOzone to examine the basic overheads in `OFS`, and xdd on raw block devices to examine block I/O overheads.

xdd [Xdd] generates I/O workloads based on specified parameters, such as read/write mix, request size, access pattern, number of outstanding I/Os. We vary three parameters in our workloads: (i) *Number of outstanding I/Os*: This is equivalent to specifying the maximum queue depth in the I/O path. In our experiments, we vary the queue depth from 1 to 32 I/Os. (ii) *Read-to-write ratio*: We use 100% reads and 100% writes, and a mix of 70% reads–30% writes. (iii) *Block size*: we use block sizes ranging from 4 KBytes to 1 MByte. In all cases, we run xdd on a 8 GByte file (8 times the RAM of the nodes) and we report numbers averaged over multiple runs for each block size. All runs for a specific device are run in sequence and the total running time for all block sizes and workloads is approximately 4 hours.

IOzone is a filesystem benchmark tool that generates and measures a variety of file operations. We use IOzone version 3.233 to study file I/O performance for the following workloads: Read, write, re-read, and re-write. We vary block size between 64 KBytes and 8 Mbytes and we use a file-size of 2 GBytes for each client.

PostMark [Katcher] is a synthetic filesystem benchmark that measures the transaction rate for a

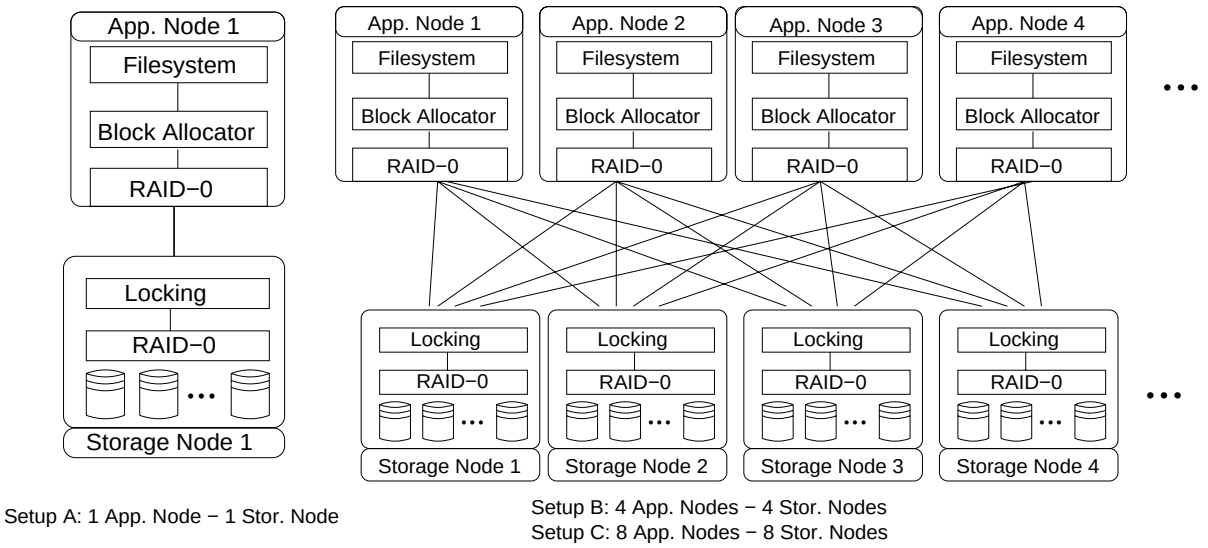


Figure 5.5: Configuration for multiple node experiments.

workload simulating a large Internet electronic mail server. PostMark's operation is described in detail in Section 4.4.3.

In our evaluation we examine basic overheads and scalability of Orchestra and `OFS`. We present three setups, where we vary the total number of nodes between 2, 8, and 16. To facilitate interpretation of results, we use the same number of storage and application nodes, resulting in three configurations: 1x1, 4x4, and 8x8 (Figure 5.5).

5.4.1 Orchestra

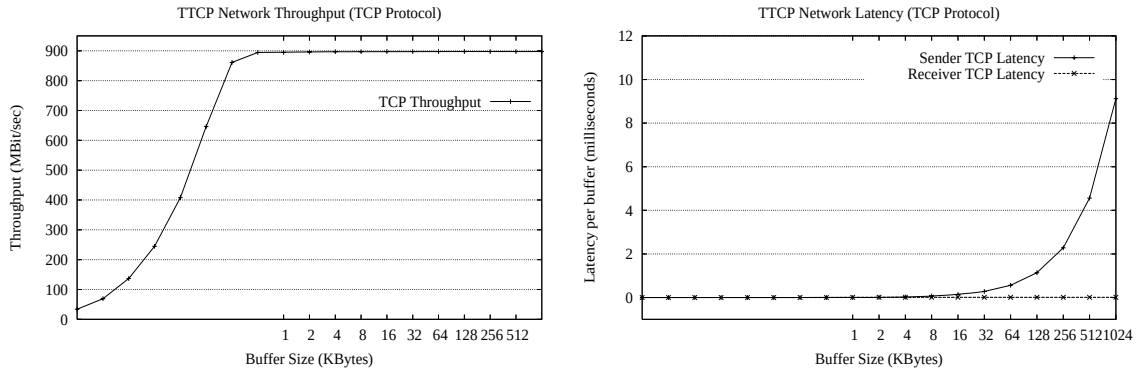
In this subsection we examine the overheads associated with Orchestra in single- and multi-node setups.

Basic costs

We measure the cost of individual operations in a local and a distributed virtual hierarchy (volume). Table 5.2 summarizes our results. Values are averaged over one thousand calls. We see that a local null ioctl costs about 4 μs. Allocating and freeing blocks through a local allocator virtual module costs about 12 and 13 μs respectively. This cost does not include saving the allocator metadata to disk, which is performed periodically (every 10 seconds) in the background. In the distributed volume case, there is an additional 310 μs overhead, that includes mainly the network delay and remote CPU interrupt

	NULL	Allocate	Free	Lock	Unlock
Local Ioctls	4	12	13	14	15
Remote Ioctls	295	-	-	310	317

Table 5.2: Measurements of individual control operations. Numbers are in μsec.



(a) Throughput (b) Latency

Figure 5.6: TCP/IP network throughput and latency measured with TTCP.

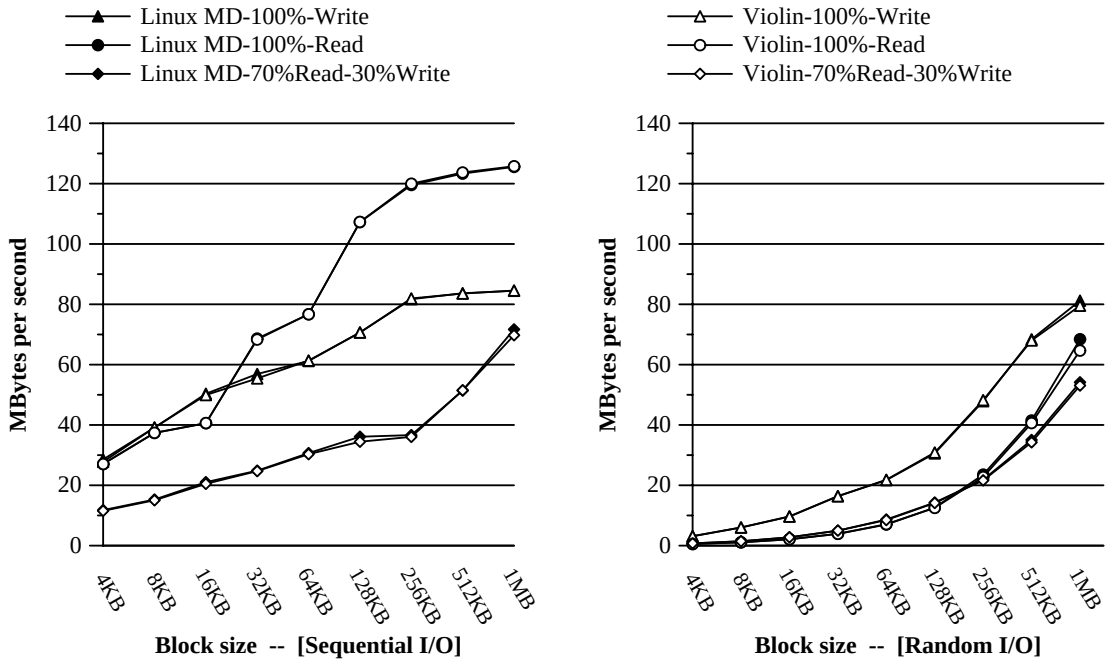
and thread scheduling costs. Overall, we notice that the main overhead in networked volumes comes from the communication subsystem, which is expected to improve dramatically in storage systems with current developments in system area interconnects, such as PCI Express/AS and 10 Gigabit Ethernet.

Figure 5.6 shows the throughput and latency between two system nodes as reported by the TTCP benchmark. We see that the maximum throughput achieved between two nodes in our system is about 112 MBytes/s (900 Mbits/s), which is the expected value of a commodity gigabit ethernet interconnect.

Figure 5.7 compares the throughput and latency of a simple Orchestra hierarchy of a single RAID0 virtual device on four SATA disks, with a similar hierarchy built with existing Linux drivers (MD) in a single node. As shown, the throughput obtained for all workloads (sequential and random) over Orchestra is similar to the throughput of MD. In terms of latency, shown in Figure 5.7(b), Orchestra also exhibits similar performance to MD.

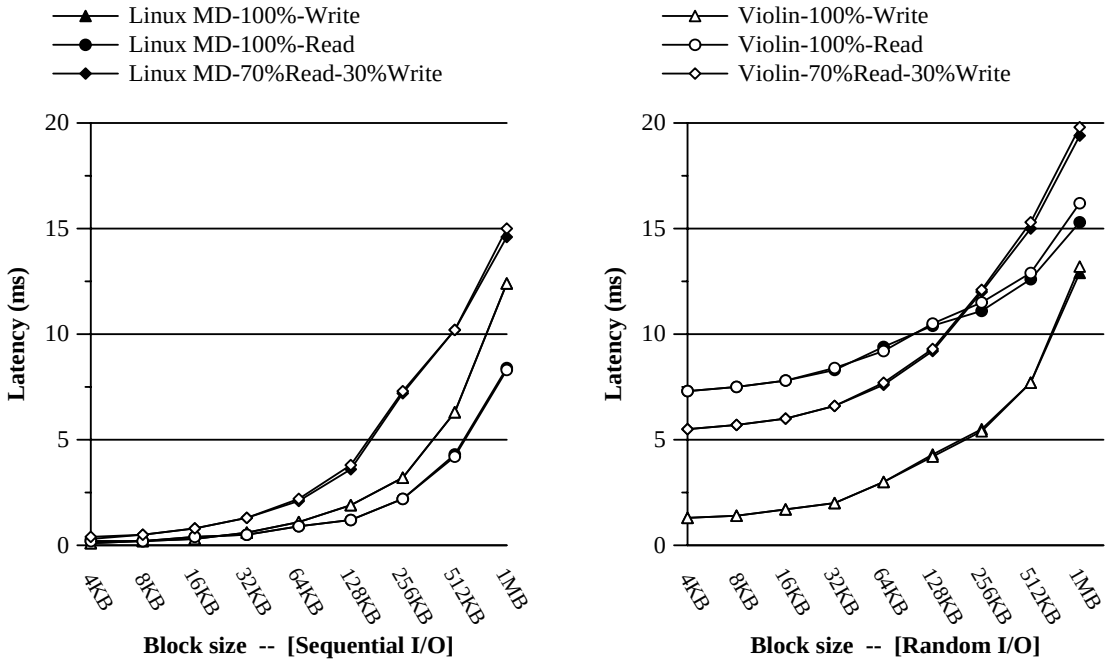
Scalability

Now we examine the scalability of Orchestra at the block-level using xdd. Figure 5.5 shows the setup we use for these experiments. Each xdd process runs at different block offsets and thus accesses separate



XDD Direct Attached: Violin vs. MD

(a) Throughput



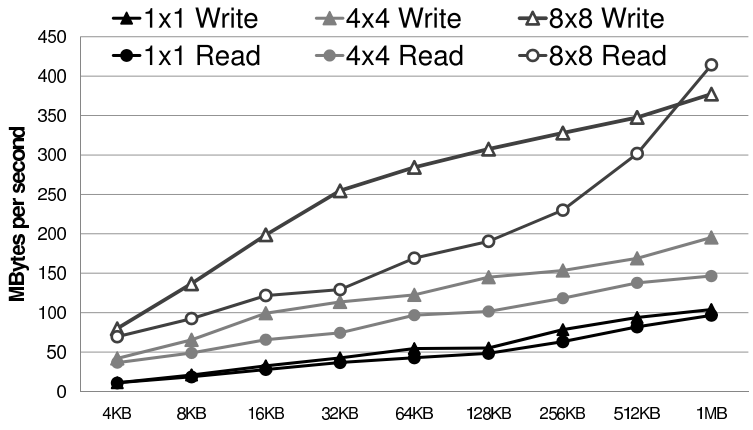
XDD Direct Attached: Violin vs. MD

(b) Latency

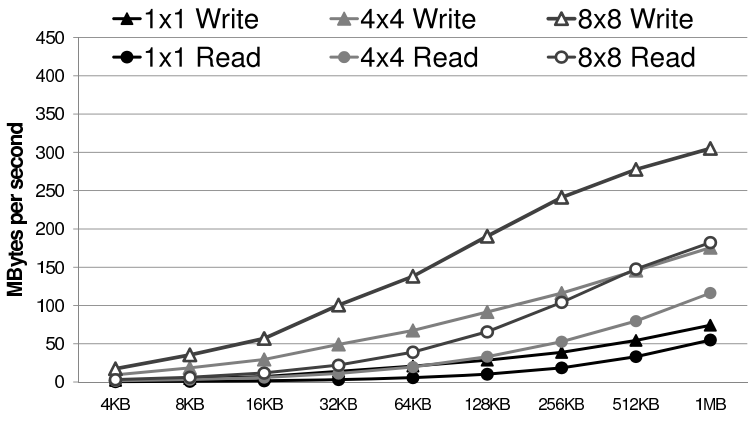
Figure 5.7: Throughput and latency for Orchestra vs. Linux MD over direct-attached disks for sequential (left) and random (right) workloads.

block ranges, minimizing caching effects on the storage nodes.

Figure 5.8 shows sequential and random read and write throughput and latencies with xdd for each configuration: 1x1, 4x4, and 8x8. Overall, we notice that as we increase the number of nodes, both read and write performance scale. For instance, in the 1x1 setup, maximum throughput is about 100 MBytes/s, while in the 8x8 setup it exceeds 400 MBytes/s. However, scalability is limited by the disks, because as the number of nodes (and in turn xdd processes) is increased, the efficiency of individual disks drops, as they perform more seeks between additional separate block ranges. The network, on the other hand, is not close to saturation, since for instance in the 4x4 setup maximum throughput is about 200 MBytes, whereas total network throughput is at least twice as much.

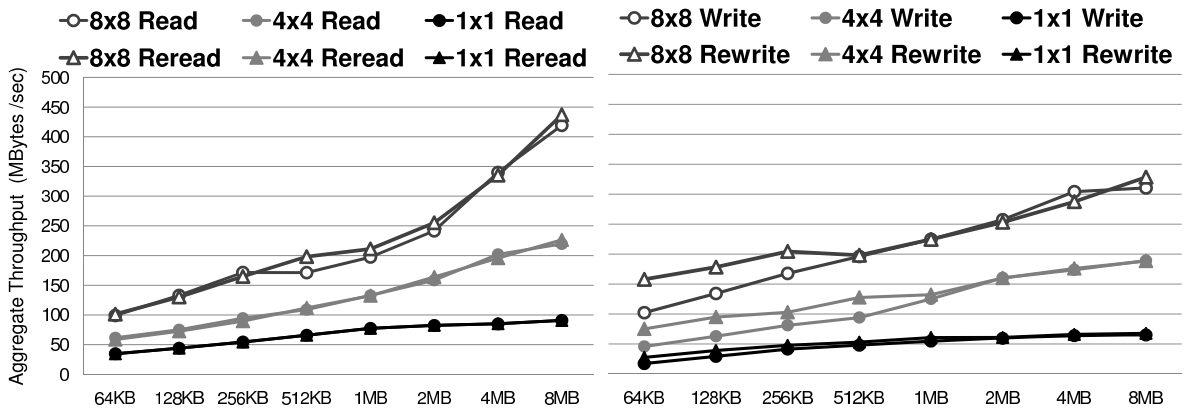


(a) Sequential I/O



(b) Random I/O

Figure 5.8: Throughput scaling with xdd for the 1x1, 4x4, and 8x8 setups.



(a) Read / Reread

(b) Write / Rewrite

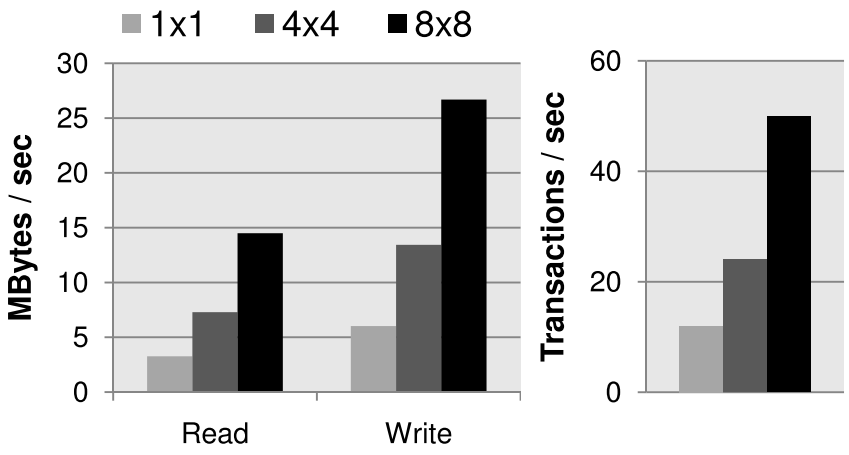
Figure 5.9: Aggregate throughput of 0FS with IOzone.

5.4.2 ZeroFS (0FS)

In this subsection we examine the overheads associated with 0FS using IOzone and PostMark.

First, we look into the base performance of 0FS on a single node. We contrast its behavior to the standard Linux Ext2FS using both IOzone and PostMark. IOzone results, show similar performance over reads, while for writes, ext2FS is about 20% faster. In the PostMark experiments and for small workloads that fit in the buffer cache, 0FS is about 25% slower. We attribute this performance difference to the increased number of system calls of the user-level 0FS compared to the in-kernel Ext2FS and the metadata caching that Ext2FS uses. Thus, we see that 0FS performs, in the local case, close to current filesystems.

Next, we examine the scalability of 0FS in distributed setups. In the base distributed configuration (1x1) we create a single volume over all available system storage and then create different directories for every application node that will access this volume. The separate instances of 0FS use the locking, allocation modules, and RAID0 modules, as shown in Figure 5.5. Each application node uses a separate directory on the distributed 0FS to perform its file I/O workload either with IOzone or PostMark. In our opinion this is the most realistic scenario for evaluating the scalability of a distributed FS, since no system should be expected to scale well when there is heavy sharing of files between clients. Finally, note that since 0FS is stateless, it does not perform any client-side caching, and thus all I/O requests generate network traffic. Figures 5.9 and 5.10 show our multi-node results with IOzone and PostMark, respectively.



(a) Throughput (MBytes/sec) (b) Transactions / sec

Figure 5.10: PostMark results over `OFS` in multiple node setups (1x1, 4x4, 8x8). The workload consists of: (i) files ranging from 128 KB to 1 MByte in size, (ii) an initial pool of 2000 files, and (iii) 5000 file transactions with equal biases for read/append and create/delete. During each run there are over 4500 files created, about 1.5 GBytes of data read and more than 3 GBytes of data written by each client.

Figure 5.9 shows that IOzone (over `OFS`) scales as the number of nodes increases for both read and write, reaching a maximum throughput of about 330 MBytes/s in the 8x8 configuration. We also see that IOzone performance and scaling follows the behavior of block-level I/O (xdd) in Figures 5.8 and 5.8. Similarly to the block-level I/O, scaling from 1x1 to 4x4 to 8x8 nodes is not linear because of reduced disk efficiency as the number of nodes (and disks increases). Thus, given that disks are not the bottleneck, `OFS` is able to scale well in cases where these is limited contention. Similarly, our PostMark results (Figure 5.10) show scaling behavior similar to IOzone. We see that quadrupling the number of storage nodes from 1x1 to 4x4 results in about doubling all metrics, whereas further increasing the number of storage nodes from 4 to 8 results in doubling performance.

Comparison to other distributed filesystems.

Considering how different our approach is from current distributed storage systems, we believe it is difficult to make an apple-to-apple comparison with existing, freely-available cluster filesystems such as the Global File System (GFS) [Preslan et al., 1999] (note that GPFS [Schmuck and Haskin, 2002] is a commercial closed-source system), since: (i) Cluster filesystems usually rely on hard-wired reliabil-

ity mechanisms (e.g. journaling) which incur a significant performance penalty. Since currently `OFS` provides only weak reliability guarantees, a comparison with a filesystem like GFS would be neither meaningful nor very fair. Note also that it's very hard to extend or remove features from GFS because of its monolithic structure. (ii) Our FS prototype is currently implemented at the user level and thus has more overheads than a filesystem implemented at the kernel level.

Our main objective in this chapter is to show that the proposed architecture does not introduce major overheads or scalability bottlenecks and that our novel approach (i.e. decoupling block allocation and locking from the filesystem) simplifies the distributed filesystem.

Next, in Chapter 6 we present RIBD, a system that extends Orchestra with mechanisms required to provide reliability, enforce consistency of distributed metadata and ensure availability. In Section 6.8 we compare RIBD to two popular filesystems, PVFS2 [Carns et al., 2000] and GFS [Preslan et al., 1999].

5.4.3 Summary

Overall, Orchestra is able to scale well when throughput is not limited by increased seek overheads in physical disks. Moreover, `OFS` follows the same scaling pattern as Orchestra in cases where there is little sharing contention. Finally, these results suggest that our approach for providing increased functionality and higher-level semantics at the block-level has potential for scaling to larger system sizes.

5.5 Limitations and Future work

Although Orchestra provides support for both easily extending storage systems as well as pushing new functions closer to the storage devices, it is currently limited in three ways: (i) continuous operation in the presence of faults, (ii) data protection, and (iii) client-side consistent caching.

To recover from failures Orchestra uses two (shadow) copies of virtual device metadata for the full hierarchy. Thus, when one copy is updated on the disk, the other (shadow) copy of the metadata is stable and reflects a previous consistent state of the system. When a failure occurs, the system can revert to the previous copy of metadata, possibly losing updates that have occurred in-between. However, Orchestra does not offer support for continuous operation (availability) and consistency when redundancy across storage nodes is used (e.g. network RAID levels 1,5 or 6). The filesystems implemented on top of it need to provide their own recovery mechanisms, such as write-ahead logs. Finally, another concern regarding high availability that deserves further investigation is dynamic, consistent modifications of the

structure of a distributed hierarchy, e.g. to transparently add or remove storage resources and migrate data accordingly.

RIBD, presented in the Chapter 6, extends Orchestra with mechanisms required to enforce consistency of distributed metadata (data) in the presence of failures and to ensure availability. Our approach relies on using lightweight transactions for providing atomicity with respect to failures and locks for serializability.

Regarding data protection, Orchestra currently relies on traditional protection mechanisms, i.e. checking for file attributes in kernel-level file systems. However, the ability to associate blocks with metadata dynamically and on-demand provides the potential for (i) performing finer-grain protection checking, and (ii) dynamically adjusting protection levels for blocks based on high-level features, e.g. time-based protection mechanisms.

Finally, Orchestra does not support client-side consistent caching. It uses the OS buffer cache in storage nodes for caching purposes, however this results in at least one network round-trip time for each I/O operation. Support for client-side caching, however, requires support for consistency since data and metadata may be cached in multiple application nodes. Such mechanisms can be implemented in Orchestra and RIBD as virtual modules, but are currently left as a topic for future work.

5.6 Conclusions

In this chapter we present Orchestra, a low-level software framework aimed to take advantage of brick-based architectures. Orchestra facilitates the development, deployment, and adaptation of low-level storage services for various environments and application needs, by providing distributed virtual hierarchies over storage systems built out of commodity compute nodes, storage devices, and interconnects.

The main contributions of Orchestra can be summarized as follows:

- Orchestra provides a novel approach for building cluster storage systems which consolidates system complexity in a single point, the Orchestra hierarchies. We examine how extensible, block-level I/O paths can be supported over distributed storage systems. Orchestra deals with metadata persistence and allows hierarchies to be distributed almost arbitrarily over application and storage nodes, providing a lot of flexibility in the mapping of system functionality to available resources.
- Orchestra provides sharing at the block level, through block-level locking and allocation facilities that can be inserted in distributed I/O hierarchies where required. A distinguishing feature of

Orchestra is that allocation and locking are both provided as in-band mechanisms, i.e. they are part of the I/O stack and are not provided as external services. We believe that our approach simplifies the overall design of a storage system and leaves more flexibility for determining the most adequate hardware/software boundary.

- We examine how higher system layers can benefit from an enhanced block interface. Orchestra's support for block locking, allocation, and metadata persistence can significantly simplify filesystem design. We design and implement a *stateless, pass-through* filesystem, ZeroFS (`0FS`), that takes advantage of Orchestra's advanced features and distributed virtualization mechanisms. `0FS` does not maintain internal state, uses Orchestra facilities for locking and block-allocation, and does not require explicit communication among its instances.

We implement Orchestra and `0FS` under Linux and evaluate them using various setups with single and multiple storage and application nodes. We find that the modularity of Orchestra introduces little overhead beyond TCP/IP communication and existing kernel overheads in the I/O protocol stack. Results on a cluster with 16 nodes show that Orchestra scales well both at the block as well as the filesystem level.

Overall, we show that providing flexibility and extensibility in a distributed storage stack does not introduce significant overheads and does not limit scalability. More importantly, our approach of providing extensible, virtual, block-level hierarchies over clustered storage systems takes advantage of current technology trends and provides the ability to repartition storage functionality among various system components, based on performance and semantic trade-offs.

Chapter 6

RIBD: Recoverable Independent Block Devices

To find a fault is easy; to do better may be difficult.

—Plutarch (46 AD - 120 AD)

An early version of the protocol design in this chapter, without the implementation and evaluation has been presented in [Flouris et al., 2006].

6.1 Introduction

Cluster-based storage architectures in use today resemble the structure shown in Figure 6.1-(a) where a high-level layer such as a filesystem uses the services of an underlying *block-level storage system* through a standard block-level interface such as SCSI. Concerns such as data availability and metadata recovery after failures are commonly addressed through data redundancy (e.g., consistent replication) at the file or block (or both) layers, and by metadata journaling techniques at the file layer. This structure, where both redundancy and metadata consistency is built into the filesystem, leads to filesystems that are complex, hard to scale, debug and tune for specific application domains [Prabhakaran et al., 2005a; Yang et al., 2004; Prabhakaran et al., 2005b].

Modern high-end storage systems incorporate increasingly advanced features such as thin provisioning [Compellent, 2008; IBM Corp., 2008c], snapshots [IBM Corp., 2008d], volume management [EVMS; GEOM; Teigland and Mauelshagen, 2001], data deduplication [Quinlan and Dorward, 2002],

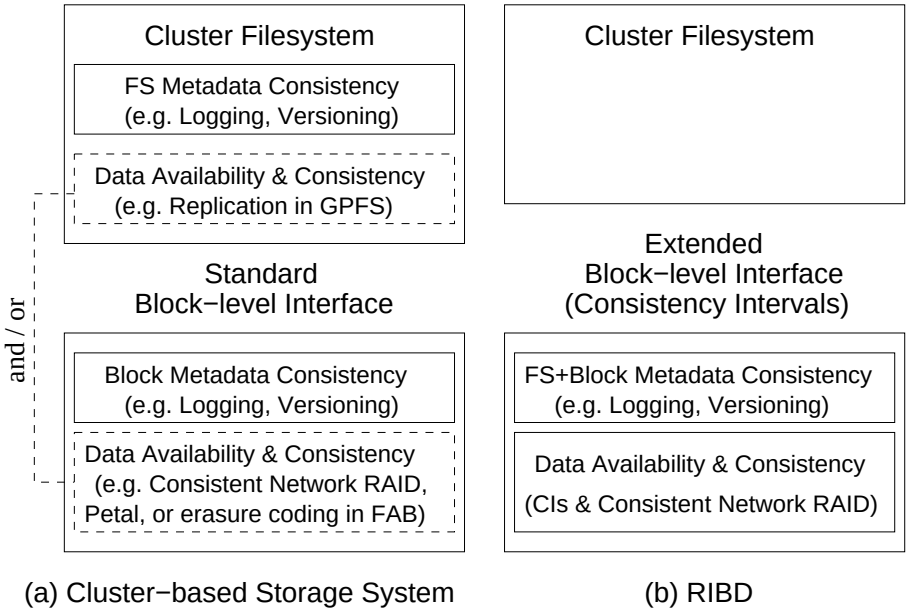


Figure 6.1: Overview of typical cluster-based scalable storage architectures and RIBD.

block remapping [English and Alexander, 1992; Wang et al., 1999; Wilkes et al., 1996; Sivathanu et al., 2003], and logging [Wilkes et al., 1996; Stodolsky et al., 1994]. The implementation of all such advanced features requires the use of significant block-level metadata, which are kept consistent using techniques similar to those used in filesystems (Figure 6.1-(a)). Filesystems running over such advanced systems often duplicate effort and complexity by focusing on file-level optimizations such as log-structured writes, or file defragmentation. Besides doubling the implementation and debugging effort, these file-level optimizations are usually irrelevant or adversely impact performance [Denehy et al., 2002].

In this Chapter we propose *Recoverable Independent Block Devices (RIBD)*, an alternative storage system structure (depicted in Figure 6.1-(b)) that moves the necessary support for handling both data and metadata consistency issues to the block layer in decentralized commodity platforms. RIBD extends Orchestra presented in Chapter 5, introducing the notion of *consistency intervals* (CIs) to provide fine-grain consistency semantics on sequences of block level operations by means of a lightweight transaction mechanism. Our model of CIs is related to the notion of *atomic recovery units* (ARUs), which were defined in the context of a local, centralized system [Grimm et al., 1996]. A key distinction between CIs and ARUs is that CIs handle both atomicity and consistency over multiple distributed copies of data and/or metadata, whereas ARUs do not face consistency issues within a single system. RIBD ex-

tends the traditional block I/O interface with commands to delineate CIs, offering a simple yet powerful interface to the file layer.

One distinctive feature of RIBD is its roll-back recovery mechanism based on low-cost versioning, that allows recovery and access not only to the latest system state (as journaling does), but to *a set of previous states*. This allows recovery from malicious attacks and human errors, as discussed in detail in Section 3.1. We believe that versioning is a particularly promising approach for building reliable storage systems as it matches current disk capacity/cost tradeoffs. To the best of our knowledge, RIBD is the first system to propose and demonstrate the combination of versioning and lightweight transactions for recovery purposes at the block storage level.

We have implemented RIBD in the Linux kernel as a set of kernel modules extending Orchestra, and exporting virtual devices layered over local or remote physical storage devices. We have also extended ZeroFS (`0FS`), the simple file layer presented in Chapter 5, to operate on top of RIBD. `0FS` is a *user-level, stateless, pass-through* filesystem that merely translates file names to sets of blocks and specifies CI boundaries in file operations. We evaluated RIBD using micro-benchmarks on a platform of 24 nodes (12 disk servers and 12 application/file servers). To understand the overheads in our approach, we compared RIBD with two popular filesystems, PVFS2 [Carns et al., 2000] and the Global File System (GFS) [Preslan et al., 1999], which however offer weaker guarantees. We found that RIBD offers simple and clean interface to higher system layers and performs comparable to existing solutions with weaker guarantees.

An early version of our protocol design, without the implementation and evaluation has been presented in [Flouris et al., 2006]. In this Chapter, we present the complete system design and its implementation, and evaluation.

The rest of the Chapter is organized as follows. In Section 6.2 we present our motivation for RIBD. Sections 6.3, 6.4, and 6.5 present an overview of RIBD and the design of the underlying protocols and components as well as the steps taken for recovering from faults. Sections 6.6 and 6.7 discuss our prototype implementation and the platform we use for our evaluation. Section 6.8 presents our experimental results. Finally, section 6.10 draws our conclusions and outlines future work.

6.2 Motivation

In this Chapter we design a system that uses block level, roll back recovery (BC-RB/RF) and examine its performance and scalability using a real prototype. Our design is motivated by the following issues.

6.2.1 File vs. block level

Local filesystems have traditionally used journaling (roll forward) for recovering from failures [Tweedie, 2000; Sweeney et al., 1996; Chutani et al., 1992]. Thus, for historical reasons, providing recovery by means of journaling in scalable storage systems has been the natural evolution. However, this leads to a number of issues:

- It requires the filesystem to be aware of replica allocation and placement. However, over the past few years these functions have already moved mostly in the block level (e.g. with RAID systems). This approach leads to duplicating certain functionality and thus, necessarily increasing system complexity and possibly incurring higher overheads.

- If replication is to be provided by the block level, e.g. using a shared virtual disk such as Petal [Lee and Thekkath, 1996] or RVSD [Attanasio et al., 1994], then the shared virtual disk has to be designed specifically for the filesystem to ensure that faults are tolerated. For instance, when GPFS [Schmuck and Haskin, 2002] is used with filesystem replication, it requires RVSD [Attanasio et al., 1994] to provide a shared disk abstraction, but does not guarantee recovery from all types of failures, since RVSD does not provide consistent replication. Frangipani [Thekkath et al., 1997] relies on Petal [Lee and Thekkath, 1996] for data and metadata replication and performs metadata logging. However, Frangipani has been designed specifically for Petal, and it is not easy to use either layer with other systems.

- Assuming that replication and metadata consistency are probably divided among the file and block levels, journaling at the file level in an efficient manner becomes challenging. Typically, the filesystem journal has to be accessible by other file servers, in case of a file server fails and there is a need to replay its log. Also, the log itself needs to be replicated for dealing with disk failures. As block placement, e.g. in disk arrays, is typically not under filesystem control, the file system cannot always optimize accesses to the journal.

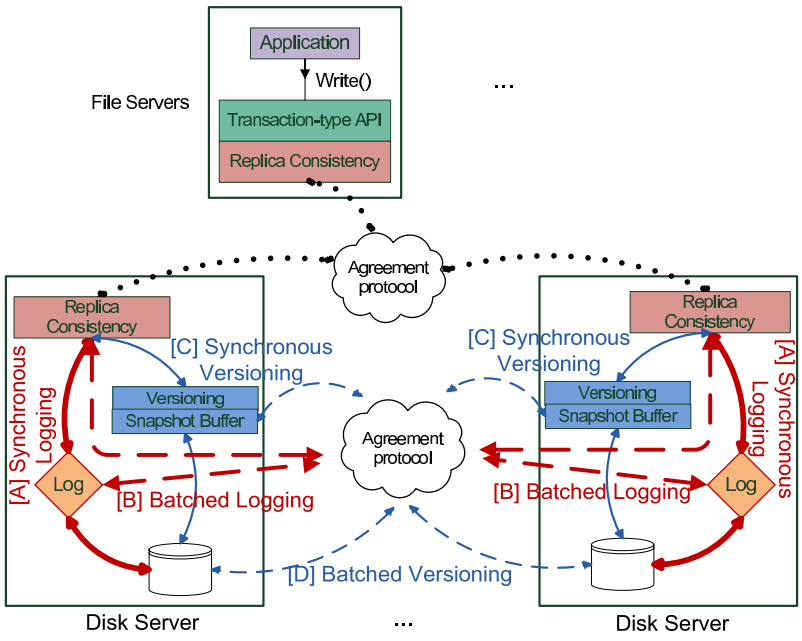


Figure 6.2: Roll forward vs. roll back recovery at the block level.

Managing consistency issues for both replication and metadata in the block level has the potential to address these issues and provide higher system layers (including the filesystem) with a simple and clean interface to reliably using available storage.

6.2.2 Roll forward vs. roll backward

Figure 6.2 summarizes the main configurations with respect to roll forward or roll backward recovery at the block level. The "thick" path in the figure describes the logging approach, with solid lines indicating synchronous logging (marked A) and dashed lines indicating batched asynchronous logging (marked B). Similarly, the "thin" path describes the versioning approach (mark C and D for synchronous and batched asynchronous versioning respectively). The two (versioning and logging) approaches have many similarities:

In all cases the filesystem provides the boundaries of consistency intervals using a transaction-type API. The block level ensures that these intervals are handled appropriately. For this purpose, each disk server can either maintain a log of operations, periodically flushing the log to the disk, or always apply operations to the disk data, periodically creating different versions of the data on disk.

In the strictest (synchronous) case, both logging and versioning need to happen at each interval. For performance reasons, however, it makes sense to perform these operations periodically, and every few

intervals. In this (batched) case, there is a need for an agreement protocol to ensure that the system can recover to a consistent point in terms both of data and metadata.

The main difference between the two approaches is that in logging, there is a need for a cleanup procedure that will re-apply the log to the actual disk blocks, whereas in versioning, there is a need for maintaining additional metadata for handling block versions. We believe that versioning has significant advantages as it maintains in a structured manner previous versions of the data, allowing recovery to *any* previous state of the system. Given, current disk capacity trends, we believe that recovery issues should be resolved using versioning techniques.

6.3 RIBD Overview

RIBD tries to address issues at three levels: (a) abstraction and primitives it presents to higher layers, (b) cost-effective scalability without compromising reliability, and (c) flexibility in its use. In this section we present an overview of RIBD and discuss the abstraction it provides. The next sections discuss primarily how it deals with consistency issues and to some extend how its implementation results in flexibility.

Previous work and other systems show that explicitly handling atomicity in higher layers, e.g. by using journaling alongside non-trivial file semantics, results in complex systems. Filesystems that support journaling tend to be hard to implement and even harder to modify and tune. Moreover, as lower system layers provide higher level abstractions of system resources, e.g. by abstracting block allocation and placement in a networked system, solutions for atomicity become difficult to optimize for performance. For these reasons, RIBD hides this complexity by providing to higher system layers and applications a simple abstraction based on *consistency intervals* (CIs), similar to transactions. Higher layers can delineate a set of consecutive (block) operations as a consistency unit that will be handled appropriately by RIBD. Figure 6.3 shows an example code segment from ZeroFS (`0FS`) that uses the RIBD API for the file creation operation.

To provide cost-effective scalability without compromising reliability, RIBD uses a *single, lightweight* transactional mechanism for both replica and metadata consistency, based on CI interface. Typically, filesystem approaches require different mechanisms for dealing with metadata consistency and replica consistency, as these refer to different entities, data blocks, file system metadata, and consistency metadata (e.g. the log itself). RIBD, by operating at the block level is able to use a single mechanism for all

```
...
begin_ci();
  lock( directory );
  if ( ! lookup( filename, directory ) {/* file exists? */
     allocate_inode( &inode_addr );
     lock( inode_addr );
     write_file_metadata( inode_addr, name, attrib );
     unlock( inode_addr );
     init_dirent( dirent, inode_addr, name, attrib );
     write_dir_metadata( directory, dirent );
  }
  unlock( directory );
end_ci();
...
```

Figure 6.3: Pseudo-code for a simple file create operation with one CI. It can be also implemented with two smaller CIs, one for updating the file inode and one for the directory.

types of blocks, regardless of whether they refer to file data or metadata. Our transactional mechanism uses agreement protocols for consistency, low-overhead versioning for atomicity, and relies on explicit locking for isolation. Given that all requests are eventually written to disk, durability is provided in a straight-forward manner at a configurable granularity by specifying a flush interval. The combination of these low-level mechanisms results in little contention when there is no (true) access sharing at the filesystem, does not impose dependencies during system operation for concurrent I/Os, and only increases the response time of individual I/O requests, if the application requires the ability to recover after each, single I/O operation.

Finally, the few optimized distributed storage systems (such as GPFS [Schmuck and Haskin, 2002]) addressing dependability issues in a single layer, the filesystem, have a monolithic structure and can hardly be adapted or extended according to the needs of a given application, e.g. by providing customizable replication. In contrast, the protocols that we propose can be easily added to (or removed from) a modular software stack.

6.3.1 System abstraction

Overall, CIs provide atomicity, consistency, and durability. Isolation is provided by a separate locking API that is provided by RIBD. All block operations enclosed in a CI are guaranteed to be treated as a single operation during recovery, i.e. all or none of the operations persist after a failure. A CI is opened and closed by the client application using explicit block-level API commands. Given that we are operating on the critical I/O path, we do not allow nested CIs. Essentially, a CI buffers all updated operations on the client side, until the commit operation.

Higher system layers use RIBD CIs to delineate units of work that may leave the system in inconsistent state after a failure. This includes all critical updates to metadata operations, because the logical structure of the storage system, such as a filesystem directory tree, must never become corrupted. CIs may also be used to guarantee the consistency of data. However, a different trade-off may be adopted in this regard: sometimes, users may be willing to accept occasional risks of data corruption, e.g. if they can rely on backups or can easily regenerate the data, in exchange of increased performance. In this case, the higher system layer need not include accesses to data in CIs. This decision on how CIs are used is left to the filesystem. In our work and `OFS` we choose to ensure both metadata and data consistency, because managing inconsistencies with increasing volumes of information, quickly becomes a difficult problem for the (in-experienced) user.

As mentioned, RIBD CIs do not provide isolation. This is achieved with appropriate block-level locking operations in the filesystem code. Typically, such locking operations will occur within CIs. Thus, along with write operations, we also need to buffer unlock operations that incur within CI boundaries, to avoid cascading effects during recovery.

We choose to *not* implement locking transparently at the CI boundaries, to allow for possible optimizations at the file level, e.g. by using a different granularity for atomicity and for mutual exclusion. Locks are only required for mutual exclusion, that is to ensure that two sets of operations on a given resource do not overlap but are serialized. For instance, a client thread may lock a given file (or range of blocks) to ensure that its updates will not be interleaved with updates from other clients. However, in a file system, specific metadata maybe associated with specific data blocks. In this case, it may be adequate for the client to include all metadata and data operations in a transaction but only lock the metadata blocks.

In summary, RIBD provides atomicity, consistency, isolation, and durability properties for CIs as

follows:

Atomicity Our mechanism guarantees that the updates encapsulated in a CI will be performed atomically, i.e. that a server will perform either all or none of the updates but never any other combination. This is achieved by buffering the updates on the client-side until the CI is closed.

Consistency CIs complete only after they are applied to all involved servers and replicas. Thus, at any time, all data replicas impacted by CIs will be in a consistent state. This is achieved by means of a two-phase commit protocol among disk servers.

Isolation CIs do not support implicit isolation. Instead, higher layers can guarantee isolation by using explicit block-range locks that are provided by RIBD.

Durability All I/O operations eventually are performed on the disk. Although this can happen before a CI is committed resulting in strong guarantees, this may have an adverse impact on performance. RIBD uses a versioning mechanism of configurable granularity to allow tuning for data loss tolerance vs. performance impact for different applications. This configurable granularity is achieved through an agreement protocol among disk servers.

Finally, there are practically two possible levels of granularity for CIs: node-level or process/thread-level. Node-level CIs are easier to implement, require less metadata for versioning purposes (agreement phase) and map directly to the most common failure domain, assuming that hardware or OS crashes are more frequent than application-level thread crashes. On the other hand, thread-level CIs are conceptually clearer (there is no imposed grouping of client threads based on the node they run on), induce less contention on the client side for delineating CIs (one counter per thread), and offer the potential for stronger recovery semantics, due their finer granularity. In our work we choose to use thread-level CI granularity. This choice only impacts only the client side module in charge of CIs, and does not affect the rest of the code.

6.3.2 Fault model

We assume that the networked storage system we are proposing behaves according to the *timed asynchronous model* [Cristian and Fetzer, 1999]. We also assume that failures of server components (CPUs, memories, disks, software faults) are fail-stop and the whole server containing the component crashes. Similarly, we assume that network components fail permanently, and that transient network failures

at links or switches are dealt with by the communication subsystem, as is typical for high-throughput low-latency interconnects.

In case of any failure, data remains available as long as there is a functional path from the application to the specific data blocks. In other words, all non-faulty components of the system keep operating in the presence of faults, however data is only available, if at least one copy is accessible through non-faulty components.

RIBD does not actively employ mechanisms to detect incorrect behavior. It assumes that component errors are reported through return codes of synchronous or asynchronous operations. Thus, RIBD does not deal with undetected errors or Byzantine behavior of components. Finally, we assume that file and disk servers belong to a single administrative domain and are co-located. Thus, we do not deal with disaster recovery issues.

Overall, RIBD ensures that:

- Any transient or permanent (hardware or software) component failure does not corrupt or destroy stored information, unless it is the only remaining copy of the data.
- Any transient or permanent (hardware or software) component failure does not prevent a volume from being accessed, unless there is no alternative path to accessing volume data.
- After any global failure, e.g. a total power failure, the system is to restart quickly from a consistent and recent state, based on a configurable policy (tradeoff).

6.4 Underlying Protocols

Figure 6.4 illustrates the system structure. Table 6.1 describes the base functionality of each module.

6.4.1 Client-server transactional protocol

The protocol implementing atomicity involves two entities: the client-side CI manager (CCM) deployed on all the client nodes and the server-side CI manager (SCM), on all the disk servers, as shown in Figure 6.5.

CCM buffers the write and lock operations associated with a given CI (and services read request from the buffer if necessary). Once the end of a CI has been detected, the CCM starts the two-phase commit protocol: First, it batches the data updates in a single `prepare` request, which is propagated to

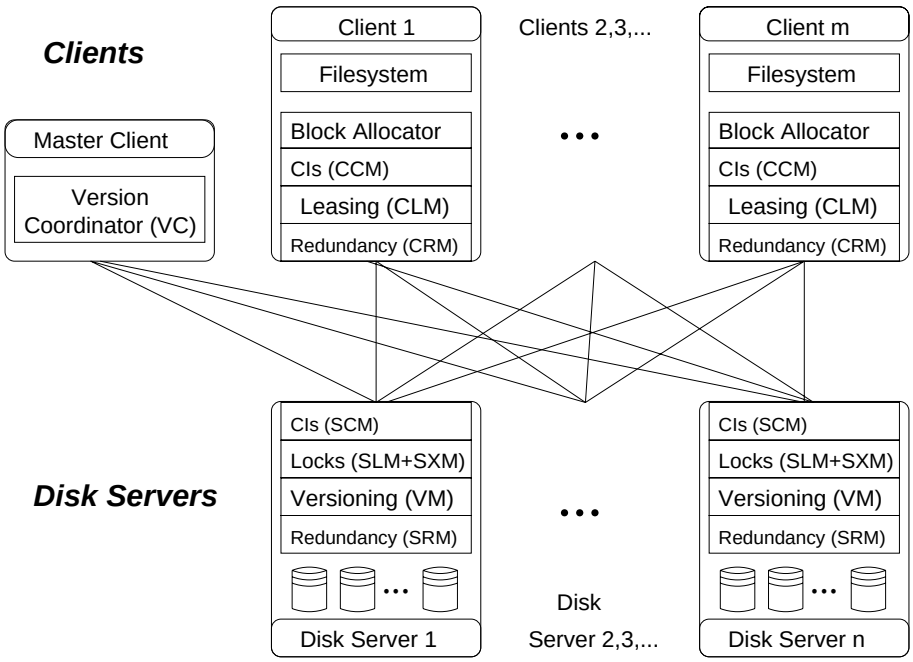


Figure 6.4: RIBD system architecture and protocol stack.

the SCM modules through the hierarchy (steps 1 and 2). At some point, this request is acknowledged by the SCM on the servers and CCM checks its status (step 3). If the `prepare` request was successful, this means that all the concerned disk servers have received the data to be written and agreed on it. Then, CCM sends a `commit` request to all the involved disk servers (step 4). Note that a `commit` request can be acknowledged (step 5) before it has reached the disk (step 6). This is done on purpose, in order to lower the cost of the two-phase protocol, at the expense of weaker guarantees in terms of reliability.

On the SCM, the committed write requests are placed in a queue, whose behavior depends on the current state of the versioning protocol. When the protocol is not running, the queue is pass-through (it just logs the IDs of the CIs that are issued and keeps tracks of which ones have completed). When the versioning protocol is running, the queue acts a buffer that allows to enforce a distributed agreement among the server nodes regarding the CIs that should be included in the next version.

The lock(s) associated with a CI are released as soon as the commit message is sent to all involved servers, without waiting for the commit acknowledgment. This optimization only works under the assumption that the communication channel between any client and any server is reliable and ordered, and that requests are not processed out of order on the server, at least until the level of the SCM module. These assumptions are in line with the features of our prototype system and with most cluster communication subsystems today.

Client Modules		Server Modules	
CCM	CI Manager	SCM	CI Manager
VC	Version Coordinator	VM	Versioning Module
CRM	Redundancy Module	SRM	Redundancy Module
CLM	Leasing Module	SLM	Leasing Module
		SXM	Lock Manager

Table 6.1: Client (left) and server (right) module functions in RIBD. *Client* denotes block-level modules on file servers, whereas *Server* denotes block-level modules on disk servers.

SCM only handles CIs and behaves as a pass-through layer for all kinds of requests that are outside the scope of CIs (including read, write, lock, unlock). As explained before, some setups may only use CIs for metadata updates and use basic write requests to access blocks associated with user data.

One should note that our terminology differs, in this regard, from the one traditionally employed in the database community. In particular, we do not provide isolation and durability.

6.4.2 Versioning protocol

Versioning involves three main modules: the version coordinator (VC), the server CI manager (SCM), and the version module (VM) that in charge of local versions. To simplify our description, we assume that there is only one version coordinator, i.e. only one client node in charge of periodically triggering the protocol to create a new version. However, in a realistic setup, this role may be attributed to several clients, for better load balancing and increased fault-tolerance. Figure 6.6 shows an overview of the versioning protocol in RIBD.

The creation of a new version relies on a two-phase protocol driven by the VC. The first phase aims at determining a globally consistent point for the version. Upon reception of the message from the VC (step 1), the SCM of each server temporarily queues the newly committed CIs and replies to the VC with a list specifying, for each client stream, the ID of the last CI that was written to disk (step 2). The VC can subsequently examine the replies from all the servers and compute a globally consistent version map, which is sent to the servers (step 3).

The second phase triggers the creation of local versions on the servers according to the version map (step 5 and 6). Before a version is actually taken on a server, CIs that are included in the version map and not yet committed to disk are extracted from the queue and flushed to stable storage. The protocol

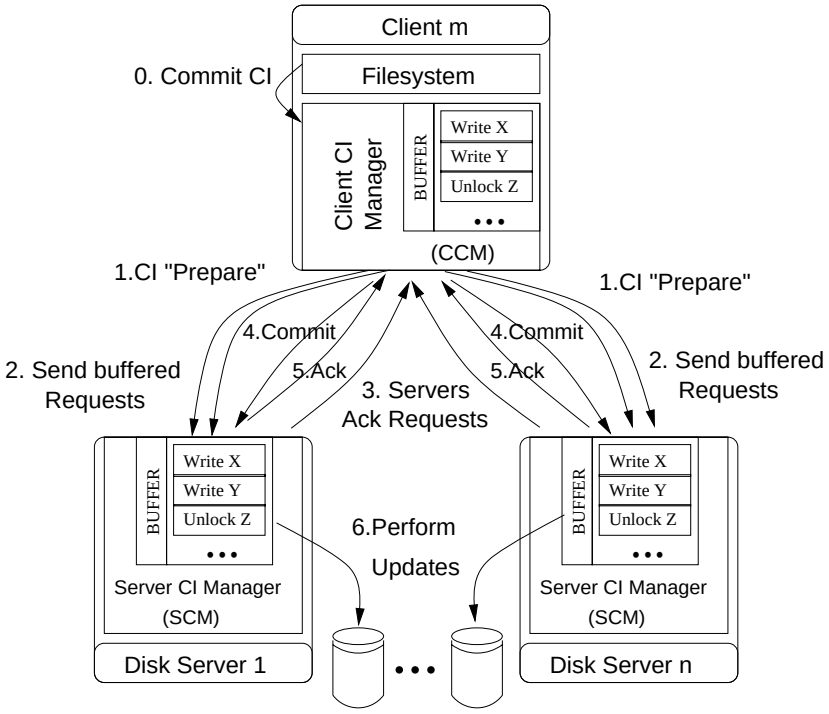


Figure 6.5: Transactional protocol between the client CI manager and multiple server CI managers.

used for CIs guarantees that, for any CI included in the version map, each involved server has received the corresponding `prepare` request and thus, the required data updates. Triggering (periodic) versions can occur in any one or more of several instance of VC.

After a global crash of the system, VC coordinates the recovery process by communicating with the storage nodes in order to determine the most recent (and complete) version that can be used. This role (coordination upon recovery) is assigned to a single instance of the module. Besides, this module can also be used when a client application wants to trigger the creation of a new version.

The VC is also in charge of a periodic garbage collection protocol, aimed at reclaiming physical storage space. The latter essentially detects versions that are too old or that did not (globally) succeed and asks the disk servers to destroy the corresponding version of their local volume.

We use versioning to offer different recovery guarantees, depending on the requirements of (end user) applications. As described, the versioning facility is based on CIs. Stronger recovery guarantees allow for smaller data loss during recovery roll back, but incur the penalty of more frequent versioning. In the extreme case, each CI commit can cause a new version, resulting in no data loss at all during recovery. This batching ability is provided by an agreement protocol during the versioning protocol described above.

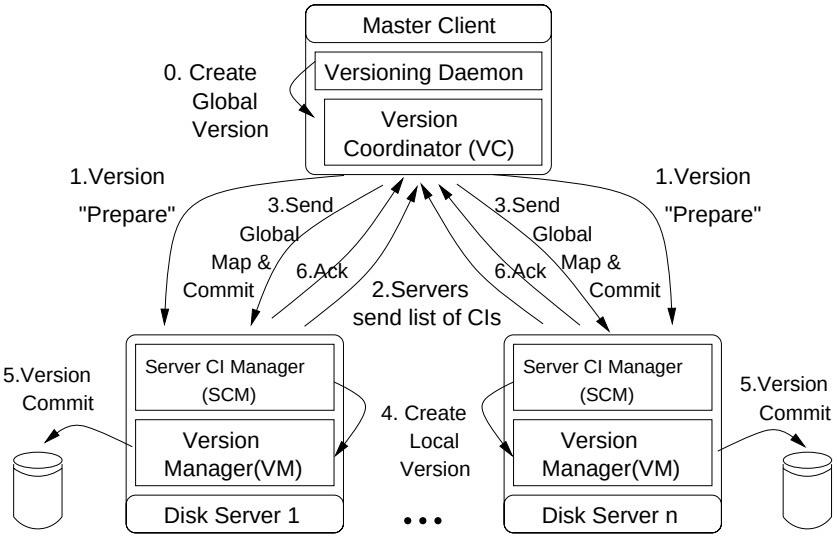


Figure 6.6: Versioning protocol between the versioning coordinator, all server CI managers, and server versioning modules.

A second, more involved decision is whether to acknowledge a CI as soon as all the servers have agreed on its status (but not performed the related I/O operations on disk) or only after the CI is persistent. In the first case, if a global crash of the storage system occurs, it is not guaranteed that the recovery point will include the modifications even though the updates were successfully acknowledged to the applications. The system can still recover to a previous consistent point, however, it will have acknowledged a CI that was eventually not committed. In the second case, all CIs committed are guaranteed to be part of the system after recovery. Our current prototype is based on the first strategy as this seems to be adequate for most applications today. However, we believe that this parameter should be set at the volume granularity.

6.4.3 Replication protocol

Availability can be achieved through redundancy mechanisms, such as replication, parity or erasure coding. In our system we use data replication across different disk servers (called network RAID [Kenchammana-Hosekotea et al., 2004] or RAIN: Redundant Array of Independent Nodes [Bohossian et al., 2001]) as the only form of redundancy. We nonetheless believe that our approach can be adapted to more complex formulas such as parity or erasure codes. The degree of replication (i.e. number of replicas) is configurable and independent of the reliability protocols we present. Currently, we only support RAIN-0, RAIN-1 and RAIN-10 functionality. This choice is based on current architectural and

cost trends of storage devices [Gray, 2005], but is not fundamental in the operation of RIBD. The CRM and SRM modules are responsible for providing local and network replication.

6.4.4 Lock protocol

We use locks based on leases for achieving mutual exclusion. Locks are provided for block ranges and are handled by RIBD. A disk server module (SXM) manages lock requests from higher-layer modules. The information on the current state of the locks is not stored in stable storage but only in memory. Disks servers also keep track (SLM), on a per-client basis, of the locks that have been granted and checks that a client sends periodic heartbeat messages to renew its leases. When a client is found to be unresponsive, the server leasing module tells the server locking manager to release the locks and takes measures so that future requests from this client will not corrupt the system (lock and read/write requests will be discarded). Finally, a client leasing module (CLM) has the role of periodically renewing the leases associated with the locks that it holds[1].

6.4.5 Liveness protocol

Temporary or permanent network partitions, although relatively unlikely, can put the system in a incorrect state and must be addressed. To operate correctly despite network partitions, the system must respect the two following invariants: (i) clients agree on the set of alive storage servers and (ii) storage servers agree on the set of alive clients. The typical solution to deal with network partitions is to use a cluster manager, which uses heartbeat messages and voting, in order to establish a quorum among the cluster nodes. [2].

Once a majority of the nodes agree on the current members of the cluster, the remaining nodes (considered faulty) must be isolated from their well-behaving peers through a fencing service. The fencing can be enforced at the hardware level (brutal power off via a remote power switch or network filtering thanks to a manageable switch fabric) or, in some cases, at the software level (through reconfiguration of the stack of the remaining nodes).

In our system, clients are not aware of each other and do not cooperate directly, which compli-

[1] It is also possible to place an additional module (on top of the client replication module), which will cache locks that are not held anymore in order to increase responsiveness for future accesses by the same node. This module works in conjunction with another module on the server node (so that the locks can be reclaimed through up-calls, if necessary).

[2] Studies have shown that a cluster manager can be scalable provided that the underlying network features hardware support for multicast [Vogels et al., 1998], which is the case for Ethernet.

cates handling "split brain" scenarios, because there is no mechanism allowing the servers to make an agreement on the current set of clients. A key difference with Petal/Frangipani in this regard is that, in Frangipani, a given lock is only managed by a single lock server, whereas in our design, we may have multiple servers in charge of the same (conceptual) lock (because we have one lock for each copy of a block). Thus, the complexity stems from the fact that we have multiple managers in charge of the same logical resource.

We deal with this by deploying a (distributed) cluster manager (CM) only for the server nodes, which does not only make decisions on the liveness of the member nodes but also elects a leader among them. The leader server is assigned with an additional and specific network address, which is known by the clients. When the CM detects that a server node is not alive, it triggers a fencing procedure for it.

6.5 System Roll-back and Recovery

In this section we briefly discuss system roll-back and recovery from five main types of failures: network partitions, client (file server) failures, disk server failures, disk device failures, and global failures.

6.5.1 Roll-back operation

Roll-back to any of a set of previous states, is useful to quickly recover the system from malicious activity of human error. In this case, the administrator issues a roll-back command from one of the file servers. Through the server-side CI managers (SCMs) the "rollback" command reaches all the client-side CI managers (CCMs), which freeze all outstanding CIs traffic to the system. Subsequently, the versioning modules on the disk servers execute the roll-back command in their local versioned storage. All disk servers, thus, roll-back in parallel to a globally consistent version. When the local roll-back is complete in all disk servers, the CI managers restart accepting CIs and the system continues operation in the rolled-back version. The file systems running on the file servers should not need to be restarted, since they do not keep state about the underlying storage layer. However, applications relying on them may need to be restarted.

The duration of a roll-back operation in our current prototype depends on the capacity of the storage within a disk server, however processing of versioning metadata in our current prototype proceeds quickly because they are placed sequentially on the disks. For a single disk server with a capacity of 16 TBs, roll-back time has been measured to less than a minute. Since disk servers roll-back in parallel,

we expect approximately the same time for a system with similar capacity in each of the disk servers.

6.5.2 Network failures

On the clients, the management of network partitions only involves the communication layer. When a client sends a request to a server, it arms a timer, which is destroyed when an acknowledgment is received. The expiration of a timer indicates that a server is unreachable. In this case, the client must contact the leader server to ask if the unresponsive node currently belongs to the group of alive servers[3]. There are three possibilities:

(i) The leader replies that the server is not alive, where the communication layer returns an error to the upper layer[4].

(ii) The leader replies that the server is alive, then this means that a network partition prevents the client from communicating with the server. In this case, the client considers itself disconnected from the all the servers and the failed request should be acknowledged with a "disconnected" status. Upon propagation of the acknowledgment in the RIBD hierarchy, all the modules will take the necessary measures to invalidate the state information that they hold (locks, cached data and metadata). All future requests should be rejected in the same way until a proper reconnection procedure succeeds.

(iii) The leader does not reply, which means that a network partition prevents the client from communicating with some or all of the servers. In this case, the disconnection procedure is employed as well. Since the above protocol is handled at the communication layer, these issues do not impact the rest of the I/O stack (the client-side modules only need to ensure proper measures for state invalidation in case they receive a notification of disconnection).

Both clients and servers may need to be fenced in order to preserve the integrity of a system experiencing network partitions. However, a distinction should be made on the severity of the fencing method to be used, according to the role played by a "failed" node. A server must be physically fenced because we need to make sure that all the clients always get the same view of the set of available servers. On the other hand, a client is responsible for fencing itself by voluntarily disconnecting from the set of servers. A self-fenced client does not have to be shut down or rebooted[5] and can (periodically) try to reconnect

[3] Note that the leader server should always be available. Using a cluster manager ensures this (if the leader crashes, a new one will be elected).

[4] Typically, upon notification of this error, the replication layer will switch to degraded mode and discard the failed server.

[5] However, the errors returned to the (user-level) application may require an application-specific treatment (possibly including a restart) but this is beyond the scope of our contribution.

to the servers. Failure to reach all the alive servers (a list can be obtained from the leader) would indicate that communications issues are still occurring and that reconnection can not happen.

6.5.3 File server/client failures

RIBD does not rely on client metadata for correct operation, and thus, client failures require little maintenance. The main issues are: releasing any acquired locks and guaranteeing appropriate operations of CIs.

If a client fails while holding locks, this will eventually be detected by timeouts of the leases on a subset of the servers. The server lease module on these nodes will react to the timeout by reclaiming all the locks held by the client (sending `unlock` requests to the server-side locking module).

The atomicity of a CI is ensured by two properties: (i) the updates associated with a CI are buffered on the client side until its closure and (ii) the 2-phase update protocol ensures that all the servers agree on whether a CI should be committed or not.

If a client fails before a CI is closed (or before any `prepare` request is sent), then the CI will be automatically discarded because it will not reach any server. If a client fails after sending the `prepare` requests (a fraction or all of them), then the two-phase protocol is not sufficient to decide. This is a well known issue of the 2PC protocol, which is blocking when the coordinator fails [Skeen, 1981]. Our solution, without assuming that the client may be able to recover quickly, if at all, relies on server timeouts and the versioning protocol. On each server, when the SCM module receives a `prepare` request, it also arms a corresponding timer. The timer is normally discarded when the associated `commit` request arrives. If the timer expires, the `prepare` request is considered as suspect and further inquiry will be necessary to determine if it deserves to be committed or not. The solution actually comes from the next round of the versioning protocol: if at least one server has received a `commit` request, for the CI, then the latter can be committed safely. Otherwise, the CI should be discarded. This resolution process happens through the next “regular” round of the versioning protocol.

Note that the protocol described above is not optimal because it may discard a CI that could actually be committed (if all the concerned servers have acknowledged the `prepare` request and the client failed just before sending the first `commit` request). Yet, this scenario would seldom occur in practice and thus, our approach trades recovery to a less recent point for simplicity in this regard.

6.5.4 Disk failures

Disk failures on a server can be masked to the client if enough redundant data has been stored on other available drives in the same disk server. In this case the server RAID layer will nonetheless notify the administrator about the failure. After the faulty disk is replaced, the administrator triggers the synchronization (a.k.a. "rebuild") of the new disk by the server module. Since we are in the context of a single node, the server RAID module acts as a central controller and traditional rebuild techniques can be used.

If the disk failure cannot be masked by the disk server, the corresponding request will fail and an error will be returned to the client replication module (CRM). For a read request, the CRM can then retry the request to a replicated server node where it should hopefully succeed. If it fails, and no redundant data are available, an error will be returned to the application.

A question is whether the faulty-disk node should be discarded eagerly (i.e. as soon as a the problem is spotted) or if it should be kept on-line because a subset of the data may still be available. The first option is simple but radical[6]. The second option is more complex and requires (i) to keep information on the client RAID module (in order to keep track of the blocks that are still valid on the faulty node) as well as (ii) to inform (through up-calls) all the clients that the server is in a degraded mode. For simplicity, we choose to eagerly discard a server node with faulty disk(s), similar to discarding a failed disk in a local RAID-1 module.

6.5.5 Disk server failures

In the occasion of a disk server failure, all disks attached to this server will become in-accessible. This situation is equivalent to multiple disk failures and is thus, handled in a similar manner. Re-adding the disk server (after repair) to the system requires synchronizing the on-disk data and the (volatile) state of RIBD's I/O stack with the rest of the active servers. Note that when the server restarts, the data on its disks are considered lost and all data need to be recovered from its replica[7].

When a disk server (re)starts, RIBD's modules connect to the replica disk server and retrieve their state from their replica modules. The versioning module, in particular, requires transferring live meta-

[6]The fencing operation could be triggered by the defective server itself. The latter may just discard (return with an error) all the subsequent requests that it receives.

[7]Using the state in the versioning metadata maps it may be possible to avoid copying all the data from the replica. This process, however, is beyond the scope of this work and is left for future work.

data from the replica disk server node. For this purpose, while the disk server is recovering, the replica node transfers all data and commands to the recovering server. When the resynchronization process is complete, the disk server may resume normal operation in the next round of the versioning protocol.

6.5.6 Global failures

After a global crash of the system (for instance due to a power failure), and after all disk servers boot, the first file server that connects to the system acts as a recovery coordinator. The version coordinator (VC) runs on this file server and coordinates the recovery process by communicating with the available disk servers in order to determine the available globally successful, consistent versions that can be used for recovery. After all disk servers agree on the version to recover (possibly chosen by a human administrator), they roll-back to that version and the system resumes normal operation and more file servers are allowed to connect.

6.6 System Implementation

We implement all related protocols under Orchestra, an extensible block-level framework for decentralized, cluster-based storage architectures presented in Chapter 5. Orchestra is implemented as a block device driver module in the Linux 2.6 kernel accompanied by a set of simple user-level management tools. The modular nature of Orchestra allows tailoring the functionality provided by a clustered storage system to the diverse needs of applications (e.g. encryption, versioning, branching, caching, replication, duplicate elimination). As discussed in Chapter 5, Orchestra supports sharing at the block level by providing (optional) locking and allocation facilities.

We implement all RIBD protocols related to reliability as a set of building blocks, which can be layered appropriately in a Orchestra virtual hierarchy, as shown in Figure 6.4 and Table 6.1. It is therefore, possible to enable/disable all reliability extensions, regardless of the “functional” features of the storage system, e.g. the specific type of replication provided. Next, we comment on implementation issues of each system component.

CI management modules (CTM, STM) implement the core of the CI mechanism.

The versioning module (VM) creates and manages remap-on-write versions of a local volume. It is placed right on top of the SRM so that the “real” data and the metadata from all the above modules can be treated in the same fashion. It interacts with the server CI manager (STM) in the context of the

versioning protocol.

Version coordination (VC) is a lightweight process and will typically be performed by a single VC that is elected once, at boot time. Another VC election can be triggered elected anytime the current VC fails and is removed from the system.

Redundancy modules (CRM, SRM) operate in the same way both for the disk and file server sides. Neither SRM nor CRM does need to run an agreement protocol. SRM handles only local replication, whereas for CRM agreement is handled by the two-phase protocol by the CTM.

In our current implementation, the server lock and leasing modules (SLM, SXM) are integrated in a single virtual device. Also, the client leasing modules (CLM) works in conjunction with the server leasing module (SLM). These modules need to be "paired" in all RIBD configurations, where locking is necessary.

6.6.1 ZeroFS (`0FS`)

To evaluate our approach we have extended ZeroFS (`0FS`), a *stateless, pass-through* filesystem presented previously in Chapter 5. `0FS` allows shared, distributed access to files from different nodes. Unlike distributed filesystems but similar to Frangipani [Thekkath et al., 1997], `0FS` does not require explicit communication between separate instances running on different application nodes. Typically, communication is required for agreement purposes. Instead, `0FS` uses the corresponding block-level mechanisms provided by RIBD volumes.

`0FS` is implemented as a user-level library that provides I/O operations on files and directories. Being user-level, `0FS` needs to cross the kernel boundary several times during a file operation, whereas a kernel implementation need only perform a single crossing.

The current version of `0FS` does not support a (client-side) cache but relies on RIBD for any caching it may perform. Our current design and implementation of RIBD does not support client-side caching, mainly because storing state on the file-servers (clients) would complicate versioning and roll-back recovery functionality. We believe that for scaling to large numbers of clients, client state should not affect system state required for recovery purposes. Thus, existing approaches for client-side caching need to be re-thought, especially given the availability of high-throughput low-latency interconnects, which is beyond the scope of our work. Furthermore, as shown in our results (Section 6.8), the use of a client-side cache does not always improve, but may also degrade performance, for example when the workload is not file system metadata-intensive (i.e. consists of few large files and directories as our

RIBD Component	LOC
Core Orchestra Framework Driver	28285
Versioning Module (VM)	1606
Server Leasing Module (SLM) & Server Locking Manager (SLM)	2109
Server CI Manager (SCM)	3452
Client Version Coordinator (VC) & Client CI Manager (CTM)	2920
Server Redundancy Module (SRM) / Client Redundancy Module (CRM)	1835
Client Leasing Module (CLM)	924
User-level config tools	4729
`OFS` user-level lib & tools	10670
Total	56530

Table 6.2: Lines of code (LOC) for RIBD components.

IOzone experiments). Such workloads are typical to many parallel applications.

We acknowledge, however, that the lack of a client-side cache in `OFS` and RIBD is a significant limitation, but we consider this outside the scope of this thesis. We intend to work on this in the future.

Finally, Table 6.2 shows the number of lines of code for each system component in our Linux 2.6 prototype. Components are divided in kernel modules, user-level tools, and `OFS`. In total, RIBD consists of approximately 40K lines of kernel code and 15K lines of user-level code.

6.7 Experimental Platform

Our evaluation platform is a 24-node cluster of commodity x86-based Linux systems. All cluster nodes are equipped with dual AMD Opteron 244 CPUs and 1 GByte of RAM, while 12 nodes acting as disk servers have additionally four 80GB Western Digital SATA disks each. All nodes are connected with a 1 Gbit/s Ethernet network (on-board Broadcom Tigon3 NIC) through a single 24-port GigE switch (D-Link DGS-1024T). All systems run Redhat Enterprise Linux 5.0 with the default 2.6.18-8.el5 kernel.

To examine the overhead of our protocols compared to existing systems we compare RIBD against two other systems: A cluster filesystem, Global File System (GFS)[Preslan et al., 1999] system, and a parallel filesystem, Parallel Virtual File System (PVFS2) [Carns et al., 2000; PVFS2]. These file systems offer weaker guarantees compared to RIBD. We choose to use them because (a) they are widely used in real setups and (b) contrasting RIBD to them will reveal the cost of offering stronger guarantees based on our approach.

All data and metadata passing through RIBD's stack, in all experiments are processed by the versioning modules at the disk-server side. In Section 6.8.5 we study the overhead of the versioning agreement protocol, measuring 0FS with Postmark under four versioning frequencies. In the rest of the experiments, however, we do not capture versions during their duration, in order to demonstrate the overhead of the transaction protocol which is in the critical I/O path. Versioning overhead is not part of the system's critical path (i.e. on every update), and we expect that usually it will be triggered in the order of tens of seconds or minutes to capture new globally consistent versions. Finally, we do not report degraded mode performance, since our goal is to show versioning and CI protocol overheads under normal operation.

In the GFS setup we use GNBD and CLVM provided by RedHat's Enterprise Linux 5.0. Note that neither GNBD nor CLVM support data replication. Thus, to configure a replicated (RAIN-10) setup for GFS we use the Linux software RAID driver (MD), which, however, does not maintain replica consistency and does not incur any related overheads. In this setup, GFS reliability guarantees are weaker than `0FS`. PVFS2 is only available with support for striping configurations because it uses its own networking protocol, which does not support data replication. In our setup we use the kernel module of PVFS2 for the clients and ext3 for the server-exported storage.

All three filesystems (GFS, PVFS2, and `0FS`) are installed and evaluated on the same 12 cluster nodes acting as clients and using the same 12 server nodes with 48 disks in total. In the case of GFS and PVFS2 we have tuned the filesystems for optimal performance according to the vendors manuals. In our evaluation we use two cluster I/O benchmarks: IOZone [Norcott and Capps] and *clustered* PostMark [Katcher].

IOZone is a filesystem benchmark tool that generates and measures a variety of file operations. We use the distributed IOZone setup in version 3.233 to study file I/O performance for the following workloads: sequential read and write, random read and write, reverse read and stride read. In all workloads we vary the block size between 32 KBytes and 16 MBytes. We use a different 8-GByte file

	File Sizes	Initial Files (per client)	Transactions (per client)	Files Created (aggregate)	Read traffic (aggregate)	Write traffic (aggregate)
Workload A	1 - 10 MB	800	4000	33588	162 GBytes	223 GBytes
Workload B	10 - 100 MB	100	300	3072	126 GBytes	191 GBytes

Table 6.3: Cluster PostMark workloads.

for each client. The aggregate data volume for every IOZone run is 96 GBytes of data for all 12 clients.

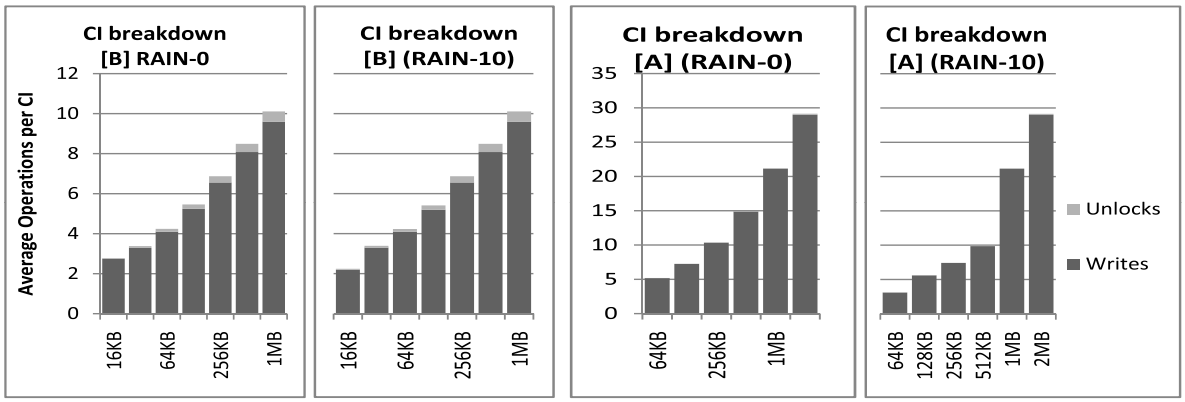
PostMark [Katcher] is a synthetic filesystem benchmark that measures the transaction rate for a workload simulating a large Internet electronic mail server. PostMark's operation is described in detail in Section 4.4.3.

The original version of PostMark is a single-node application. To use it in our setup, we modify its initialization and termination code using MPI to: (a) spawn processes on a cluster of nodes, (b) synchronize the various benchmark phases, and (c) communicate aggregate results at the end. Note that the benchmark code itself remains unchanged. In our setup, each client node/process uses a different directory on the cluster filesystem. The transactions issued by each process consist of (i) a create or delete file operation and (ii) a read or append file operation. Each transaction type and its affected files are chosen randomly. When all transactions complete, the remaining files are deleted.

We use two workloads for PostMark (Table 6.3) that differ in file size distribution (1-10 MBytes for the first and 10-100 MBytes for the second), the number of initial files per client, and the number of transactions per client. Both workloads create a sufficiently large number of files and enough read and write traffic to fill in the node caches and allow for consistent results.

To facilitate interpretation of results, we use a symmetric system configuration with the 12 disk servers 12 file servers/application clients for all filesystems. For the configuration of `OFS` we have used the protocol stack shown in Figure 6.4 and two data distribution setups: (i) a striped volume with no redundancy, where all 48 disks in the cluster are striped at the disk server side (RAID-0) and the disk servers are striped at the file server side (RAIN-0). (ii) a RAIN-10 volume with consistent RAIN-1 replication, where the disks are striped at the disk servers using RAID-0 and the disk servers are mirrored and striped at the file server side (RAIN-10). In all RAID and RAIN setups we use a chunk-size of 128 KBytes.

Figure 6.7 shows the average number of write and unlock operations in each CI for workloads A, B, for different request sizes. Writes, refer to individual I/Os that are performed to disks by the disk



(a) Workload B (b) Workload A

Figure 6.7: PostMark CI breakdown to write, unlock operations in RIBD.

servers. Although the number of operations specified by the application in a CI is statically known, the number of I/Os that are eventually generated by these operations depends on the block placement. Since we measure this statistic on the disk server side, RAIN replication does not affect the number of I/Os in each CI (but affects the number of CIs that disk servers see).

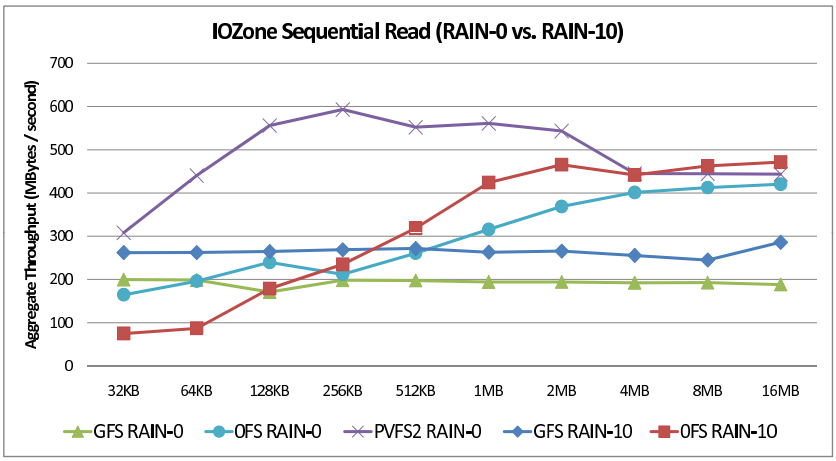
6.8 Experimental Results

In this section we examine the overhead associated with our approach on a setup with multiple storage and filesystem nodes.

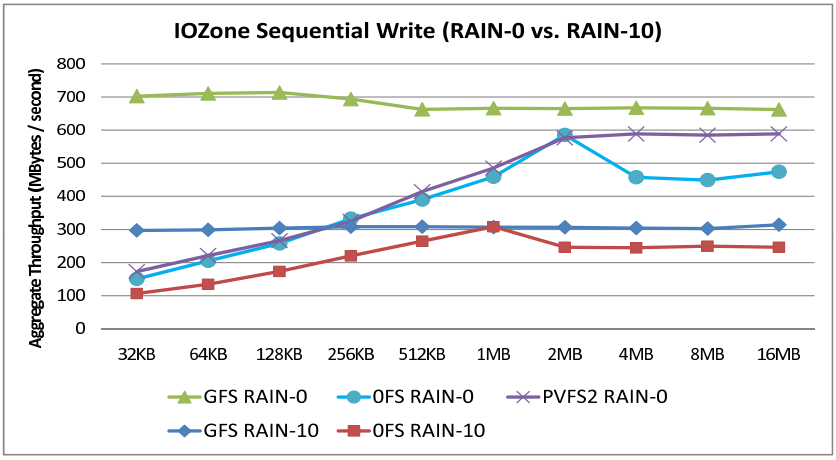
6.8.1 Overhead of Dependability Protocols

Figures 6.8, 6.9, and 6.10 show IOZone results for sequential, random and mixed workloads. There are five curves on each graph, showing the three filesystem in a RAIN-0 setup and additionally GFS and `0FS` on a RAIN-10 configuration. As mentioned, GFS in the RAIN-10 configuration does not support a replica consistency scheme, and thus, has lower overhead.

In the case of sequential read (Figure 6.8-a) `0FS` outperforms GFS and is very close to PVFS2 for large block sizes. In the case of small requests, PVFS2 and GFS perform better due to their client-side cache which aids sequential prefetching. For sequential writes (Figure 6.8-b) `0FS` performs worse than PVFS2 and GFS in large block sizes, showing the overhead of the reliability protocols. In RAIN-10 and for average request sizes (512KB-1MB), `0FS` is similar to GFS. However, when the request size increases, the increased number of acknowledgments for the two-phase protocol incur a performance



(a) Sequential Read



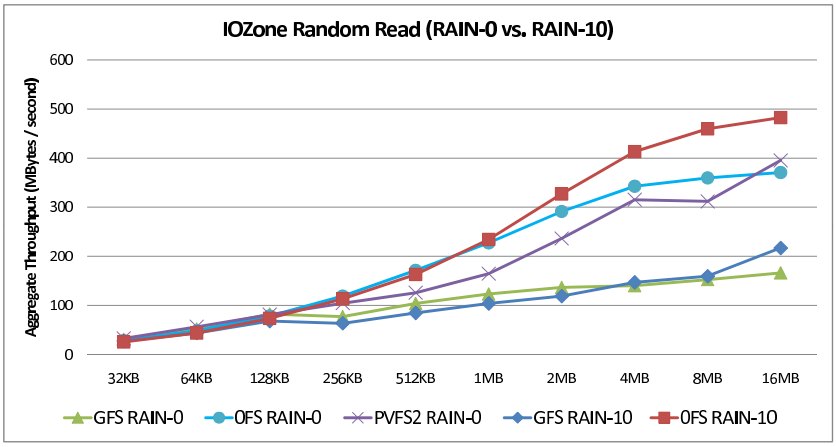
(b) Sequential Write

Figure 6.8: IOZone sequential read/write results.

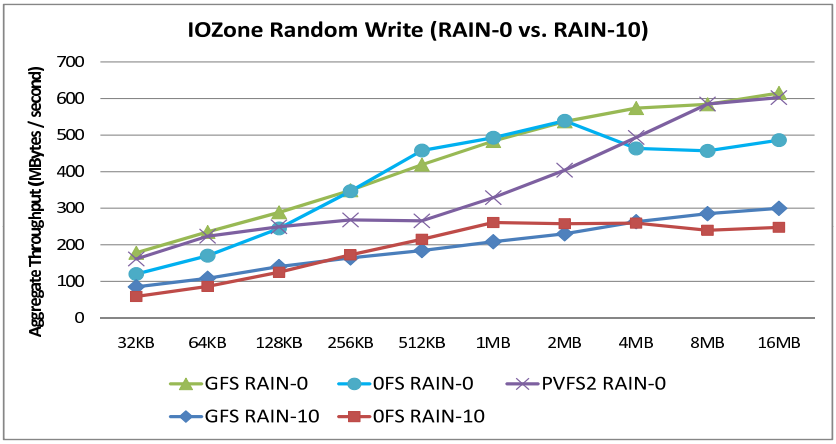
degradation of 15-20%. For small request sizes, performance is again lower due to the lack of client-side caching.

In the case of random I/O, Figures 6.9(a)(b), `OFS` performs better than both GFS and PVFS2, since the random workload minimizes the client-side caching effects. Random write for the RAIN-10 volume (Figure 6.9b) shows practically no performance overhead compared to GFS, which has no reliability protocol. In the RAIN-0 case, the overhead of the protocol appears in the very large block sizes. Please note that, even in the case of RAIN-0, our prototype uses CIs and the two-phase protocol for writes since this is required to guarantee atomic updates of distributed data blocks.

In the case of random mixed I/O (70% reads - 30% writes) results, Figure 6.10 `OFS` performs



(a) Random Read



(b) Random Write

Figure 6.9: IOZone random read/write results.

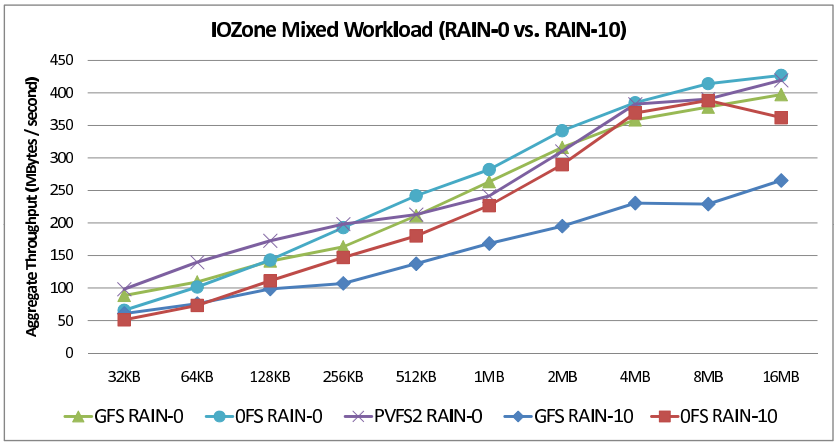
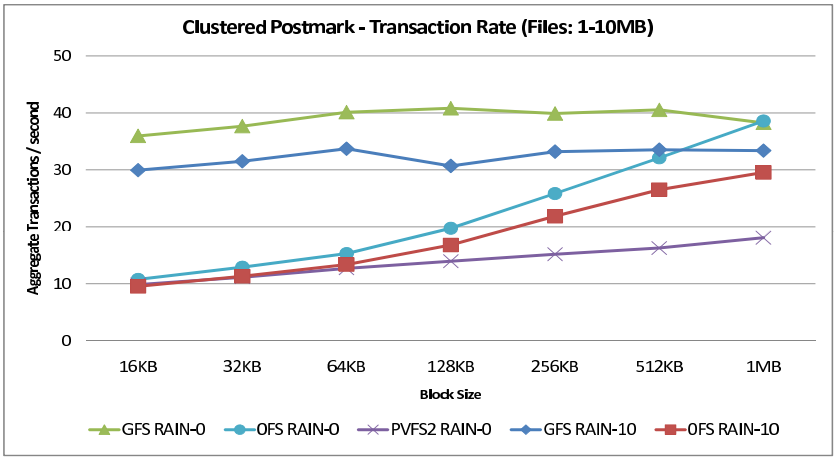
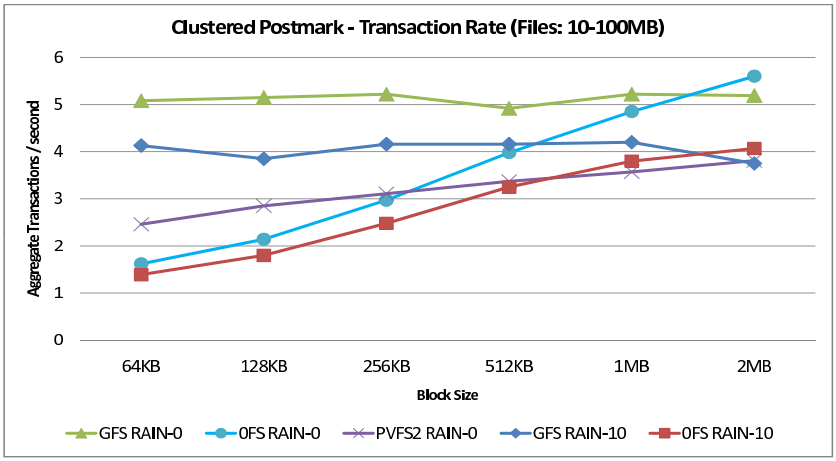


Figure 6.10: IOZone random mixed workload (70% read - 30% write) results.



(a) Workload A (1 - 10MB files).



(b) Workload B (10 - 100MB files).

Figure 6.11: PostMark aggregate transaction rate (transactions/sec) for workloads A and B.

slightly better than GFS and PVFS2, in the RAIN-0 configuration. In the replicated setup (RAIN-10), `0FS` performs significantly better than GFS, when the record size is increased. As mentioned above, this occurs because the random workload minimizes the client-side caching effects.

Finally, in PostMark (Figure 6.11[8]) we observe that `0FS` mostly outperforms PVFS2 on RAIN-0. GFS, on the other hand, maintains a lead in all cases, except in large request sizes, where `0FS` performs very close. We attribute this performance lead of GFS in metadata-intensive workloads to the use of the client cache for metadata and the fact that `0FS` needs to cross the kernel boundary 3-4 times per filesystem operation. As block size increases, this does not happen as often, and we are able

[8]Note that transactions reported in this figure are application-level transactions as reported by PostMark and not related to RIBD's operation.

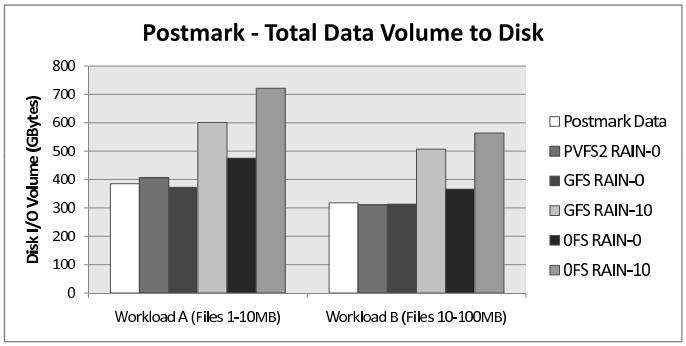


Figure 6.12: PostMark: Total data volume that reaches system disks.

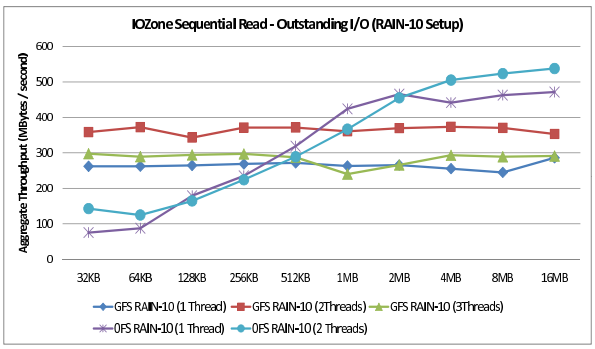
to match GFS performance in large block sizes. The performance of GFS also shows that it performs asynchronous logging, flushing the metadata log to disk infrequently.

As mentioned, `0FS` does not use a client side cache. To examine the impact of the lack of a cache on `0FS` we show in Figure 6.12 the total data volume that reaches physical disks. First, we see that in `0FS` configurations this amount is higher by up to about 20%. Second, (not shown here) the amount of data for each request size remains approximately the same. Given that GFS performs better only for smaller requests, we believe that the extra traffic is not a bottleneck that skews our results. On the other hand, the GFS cache affects request response time, especially for small requests, and since PostMark is latency-bound, we believe that this role of the client-cache results in the better GFS results for small request sizes. This effect is exacerbated by the use of TCP/IP as our communication subsystem, which not only incurs high latencies compared to state-of-the-art networks, but also exhibits problematic behavior, such as the Incast problem [Phanishayee et al., 2008].

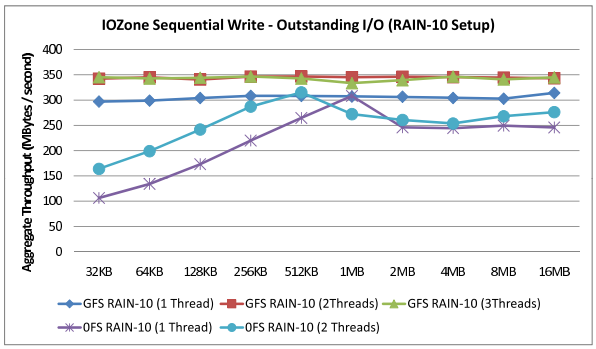
6.8.2 Overhead of Availability

To examine the performance overhead of providing availability, we compare the performance of RAIN-0 vs. RAIN-10 with IOZone (Figures 6.8-6.9). This comparison is only applicable to GFS and `0FS`. We see that performance degrades by approximately a factor of two as expected, because of mirroring. Read operations however, are an exception, where both GFS and `0FS` perform better in RAIN-10 than RAIN-0 for large requests because of the efficient load-balancing of replica reads.

For PostMark workloads (Figure 6.11) we observe also an average 20% performance degradation in the transaction rate, despite the fact that available disk throughput is effectively reduced to half. Overall we find that the performance impact of availability with `0FS` in metadata-intensive workloads is in the

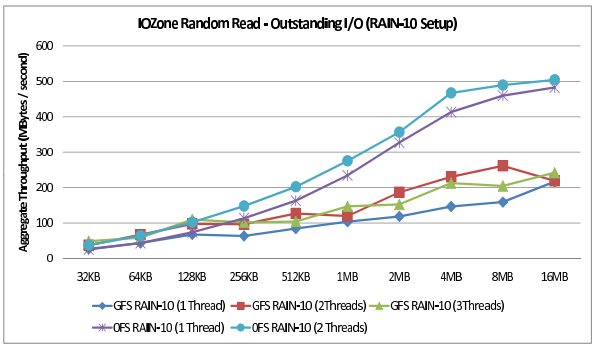


(a) Sequential Read for RAIN-10 Setup

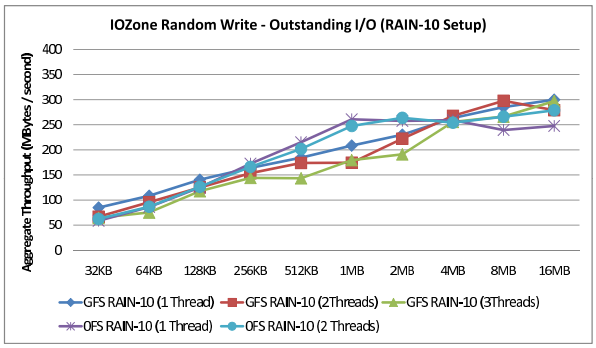


(b) Sequential Write for RAIN-10 Setup

Figure 6.13: IOZone sequential results for 12 clients on RAIN-10 with up to 3 threads per client.



(a) Random Read for RAIN-10 Setup



(b) Random Write for RAIN-10 Setup

Figure 6.14: IOZone random results for 12 clients on RAIN-10 with up to 3 threads per client.

range of 15-20%.

6.8.3 Impact of Outstanding I/Os

To examine the impact of increasing the number of outstanding I/Os in the system we use IOZone with GFS and `OFS` on a RAIN-10 configuration and PostMark with workloads A and B on RAIN-0 and RAIN-10. In these experiments we have used multiple benchmark threads per client node, thus multiplying the load on the system and inducing more concurrent I/Os. IOZone supports multi-threaded mode, however PostMark does not have such an option so we executed multiple PostMarks in parallel on each node.

In Figures 6.13, 6.14 and 6.15 we see that increasing the number of outstanding I/Os of IOZone from one to two, increases throughput for both GFS and `OFS` up to about 30%. We observe, however,

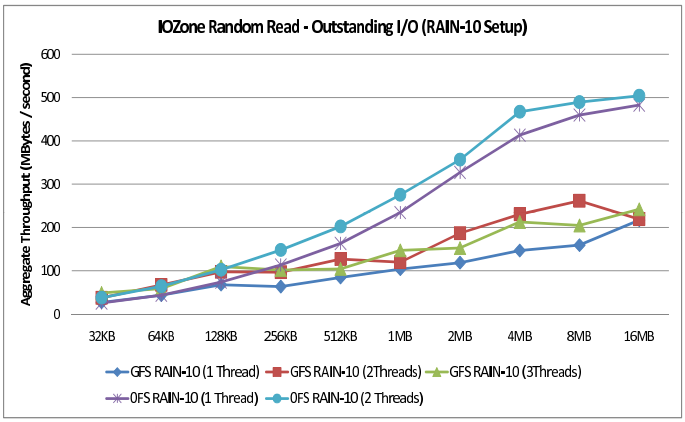
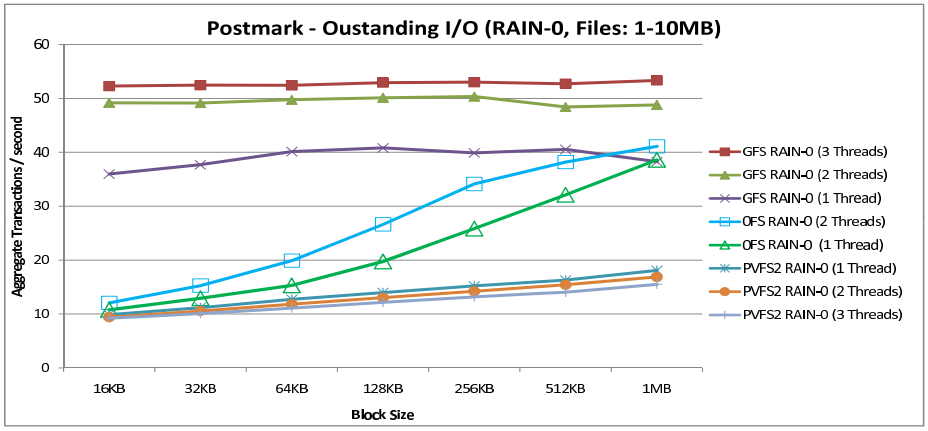
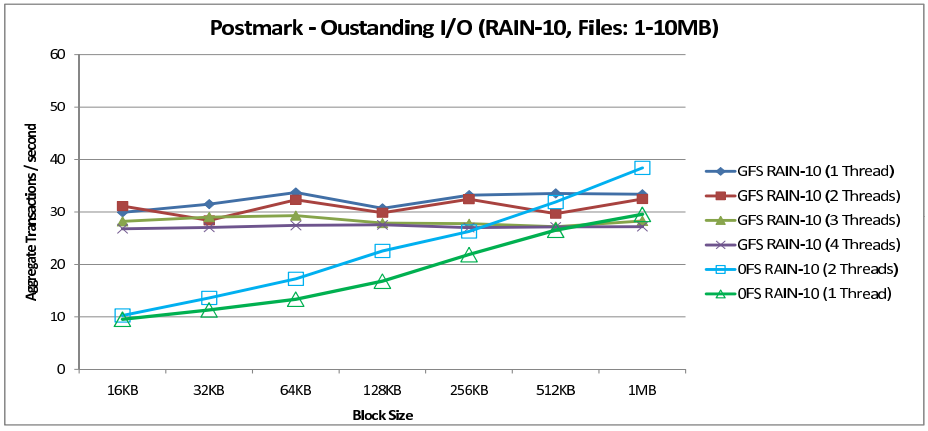


Figure 6.15: IOZone random mixed workload (70% read - 30% write) results for 12 clients on RAIN-10 with up to 3 threads per client.

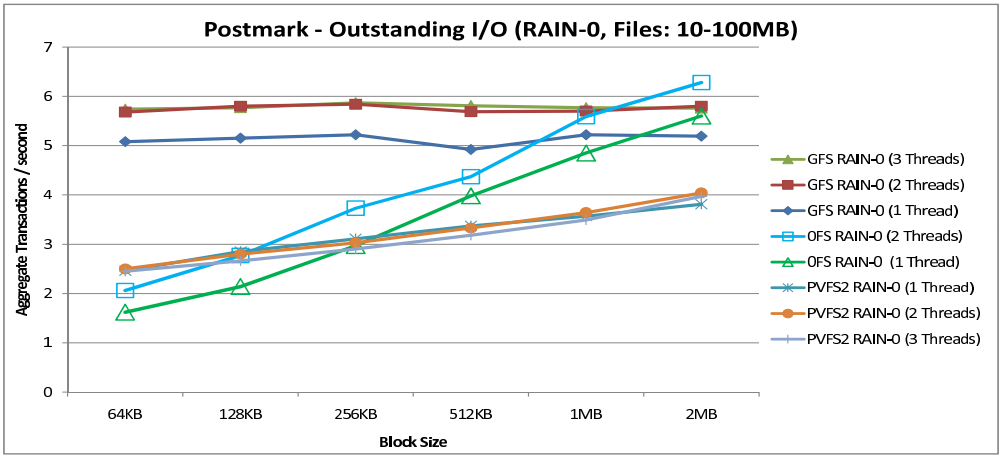


(a) Workload A for RAIN-0 Setup

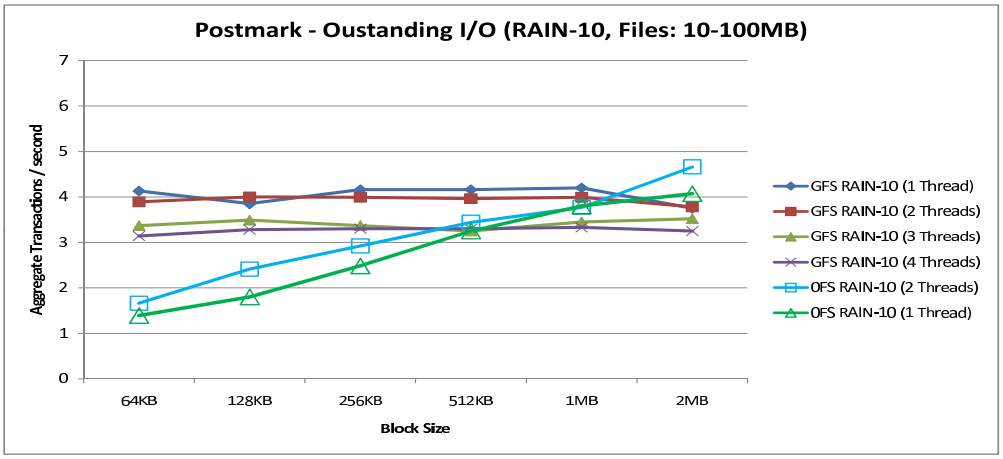


(b) Workload A for RAIN-10 Setup

Figure 6.16: PostMark aggregate transaction rate (transactions/sec) for workload A with up to 3 threads per client.



(a) Workload B for RAIN-0 Setup



(b) Workload B for RAIN-10 Setup

Figure 6.17: PostMark aggregate transaction rate (transactions/sec) for workload B with up to 3 threads per client.

that in the case of GFS, more than two outstanding I/Os per client can degrade performance (sequential reads) up to 20%.

In the case of PostMark (Figures 6.16 and 6.17), we also observe that increasing the number of outstanding I/Os also increases throughput up to about 30% in GFS and `0FS`. On the other hand, PVFS2 does not seem to benefit from the increased concurrent I/Os as its performance remains the same as with the single thread. Finally, Figures 6.16 and 6.17 show that GFS does not seem to scale with more than two threads per client. `0FS` results with more than two threads are not available because of stability issues in our prototype.

6.8.4 Impact on System Resources

Now we examine the impact of our protocol on system resources, but looking at CPU utilization, disk utilization and latency.

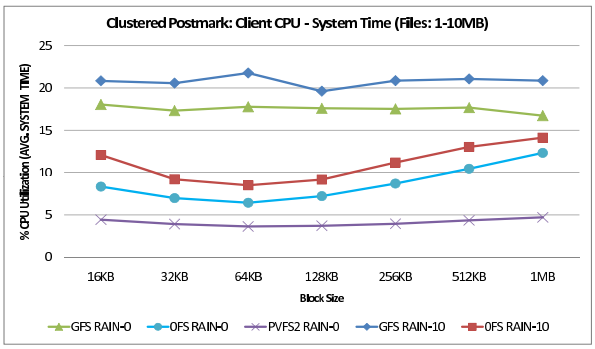
CPU utilization: Figure 6.18 shows the CPU utilization on the server and client sides for both PostMark workloads. We show only system and wait times, as user time is always less than 5% and in most cases less than 2%. First, we observe that overall system CPU utilization remains at low levels and up to 25% in the worst case, while I/O wait times can rise up to 40% both on the server and client side. However, the CPU has not been saturated in any of the experiments. Second, we note that all filesystems have similar system CPU utilizations on the disk server side. Third, we observe that `OFS` has a higher variance in CPU utilization. CPU utilization increases with smaller requests sizes; Request size affects CI size and thus, how frequently RIBD uses the underlying protocols. Small request sizes incur higher CPU utilization on disk servers. Finally, we note that on the client side, GFS incurs significantly higher CPU utilization, due to the use of a client-side cache.

Disk latency and utilization: Figure 6.19 shows the average disk response time for PostMark. Differences in response time reflect longer seeks due to differences in block placement and access locality. `OFS` incurs higher latency than GFS because it does not use a client-side cache and it cannot fence metadata accesses. Thus, data and metadata requests alternate more frequently than in GFS, resulting in longer seek times. Finally, Figure 6.21 shows average disk utilization. We see that RIBD results in up to double disk utilization compared to GFS and PVFS.

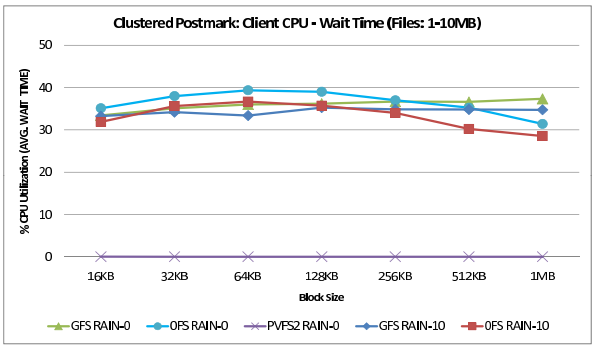
Figure 6.20 shows the average disk throughput for read and write requests. We see that `OFS` reaches higher throughput. This is mainly due to the fact that RIBD is able to send to the disk larger requests than GFS. GFS requests go through the client side cache and are eventually issued at smaller sizes to the disk, resulting in lower throughput.

6.8.5 Impact of Versioning Protocol

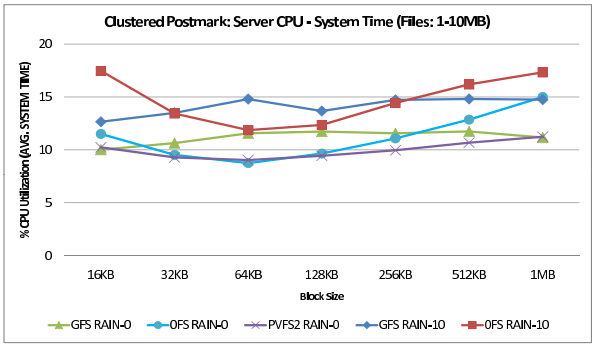
To examine the impact of the versioning protocol in RIBD we repeat the PostMark experiment with `OFS` using workload B on RAIN-10 (shown in Figure 6.11(b)) and four versioning frequencies: 30 seconds, 60 seconds, 180 seconds and never (i.e. no versioning). The garbage collector that deletes versions to free disk space when the disk is close to filling up, is also running during the experiments. The



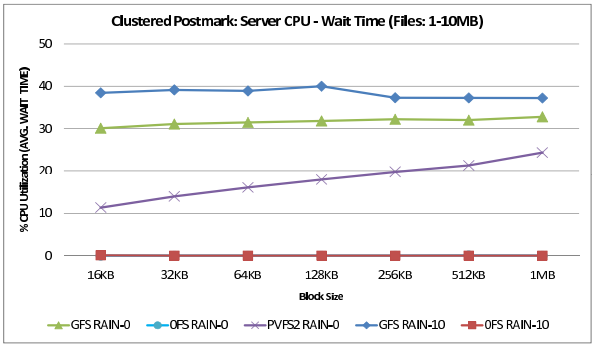
(a) Client CPU System Time for Workload A



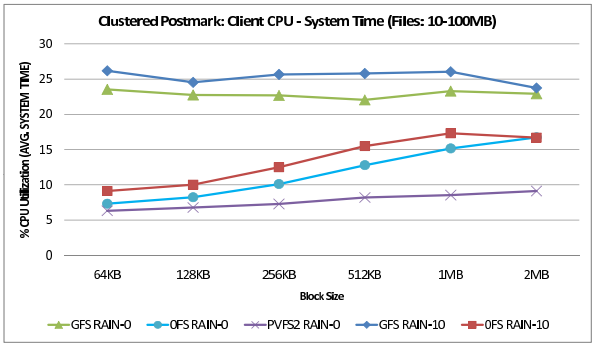
(b) Client CPU Wait Time for Workload A



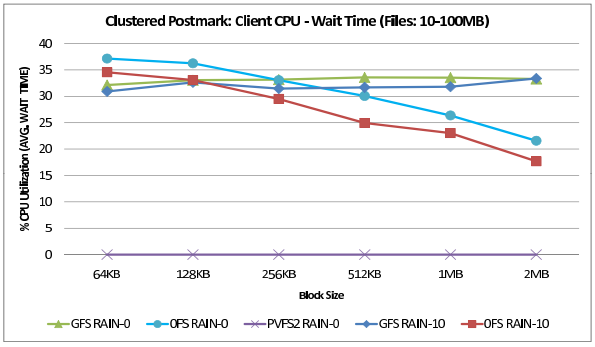
(c) Server CPU System Time for Workload A



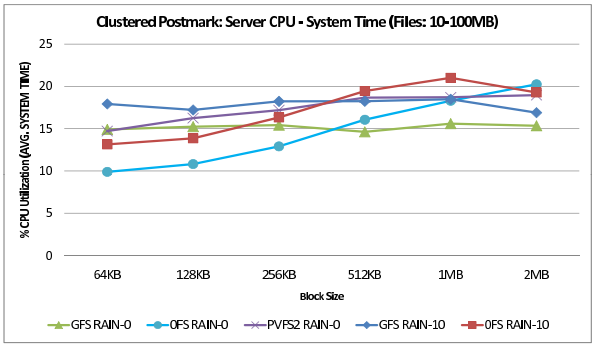
(d) Server CPU Wait Time for Workload A



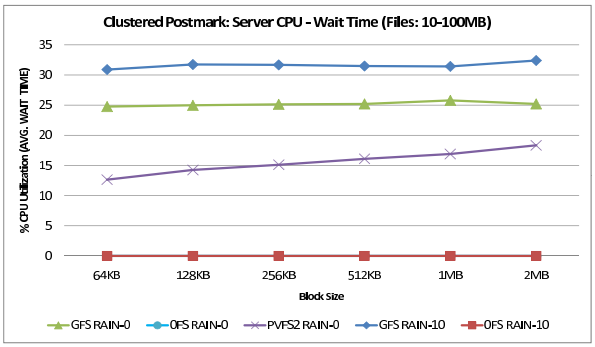
(e) Client CPU System Time for Workload B



(f) Client CPU Wait Time for Workload B

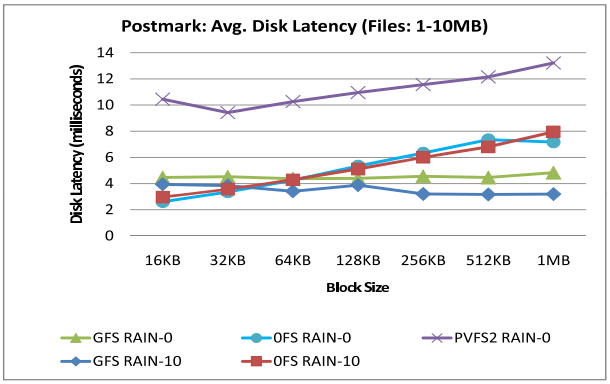


(g) Server CPU System Time for Workload B

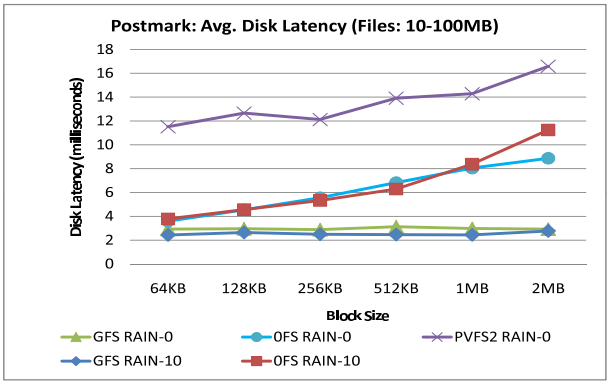


(h) Server CPU Wait Time for Workload B

Figure 6.18: PostMark average CPU utilization (%) on the servers and clients for workloads A,B.

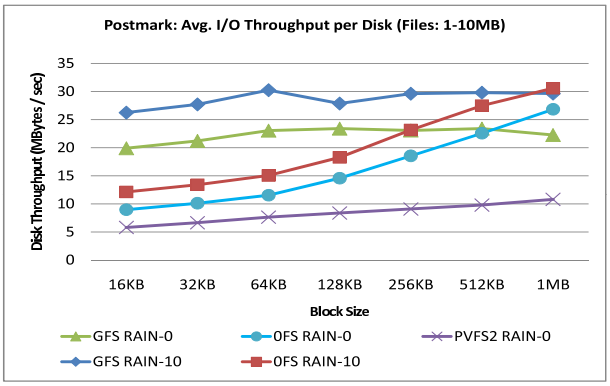


(a) Workload A (Files 1-10MB)

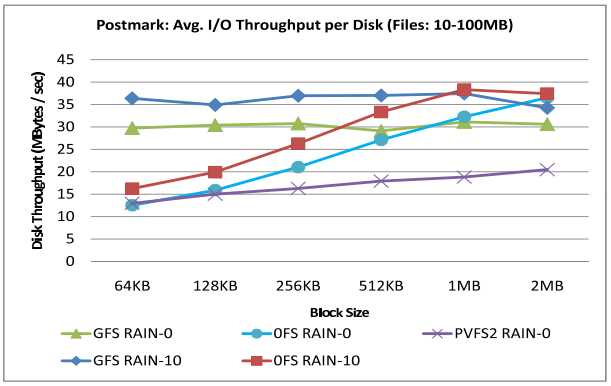


(b) Workload B (Files 10-100MB)

Figure 6.19: PostMark average (read/write) disk latency.



(a) Workload A (Files 1-10MB)



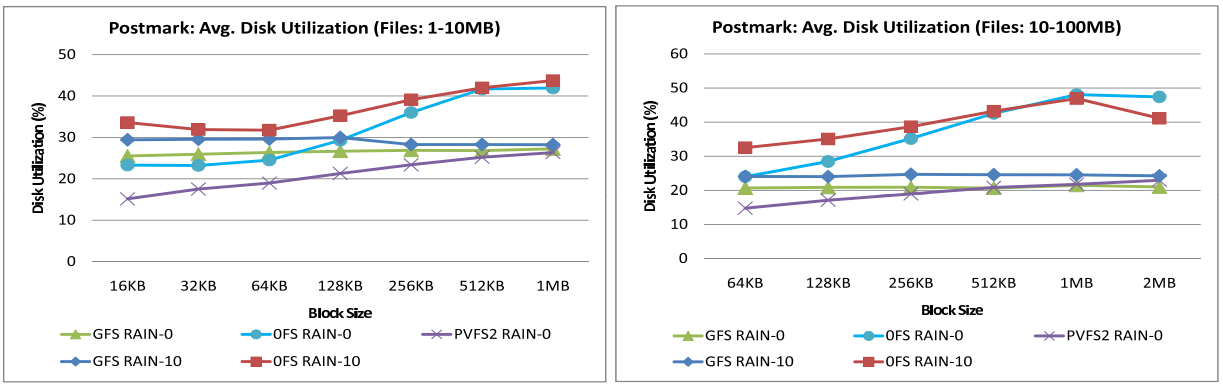
(b) Workload B (Files 10-100MB)

Figure 6.20: PostMark average total (read and write) throughput per disk.

results are shown in Figure 6.22. As shown, the maximum overhead of this protocol on the PostMark transaction rate is 21.7% for the 30 second version case with a block size of 64 KBytes. As the block size is increased to 1 MByte, the overhead of the protocol becomes less that 2%. We also find that lowering the versioning frequency improves performance slightly, since the versioning agreement protocol and the garbage collector run less frequently.

6.8.6 Summary

Overall, we find that although RIBD's protocols for maintaining consistency affect system behavior and consume resources, performance and scalability, especially for larger requests, remains comparable to GFS and PVFS2. We also find that performance in small requests and metadata-intensive workloads is greatly enhanced by a client-side cache. Finally, we conclude that RIBD's approach is better than



(a) Workload A (Files 1-10MB) (b) Workload B (Files 10-100MB)

Figure 6.21: PostMark average disk utilization across all disks.

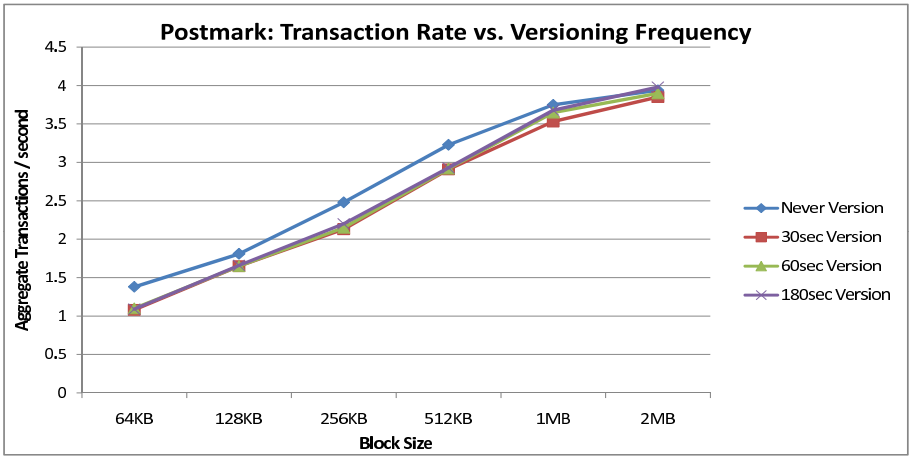


Figure 6.22: PostMark aggregate transaction rate (transactions/sec) for workload B and four versioning frequencies.

existing solutions, since it offers stronger consistency guarantees with similar performance.

6.9 Limitations

A limitation of our RIBD and `OFS` prototypes is that they do not support a client-side cache, mainly because storing state on the file-servers (clients) would complicate versioning and roll-back recovery functionality. A client-side cache is expected to offer performance benefits in certain workloads and reduce network and disk I/O traffic, as shown in our results. We believe, however, that for scaling to large numbers of clients, client state should not affect system state required for recovery purposes. Thus, existing approaches for client-side caching may need to be re-thought, especially given the availability

of high-throughput low-latency interconnects. Furthermore, as shown in our results (Section 6.8), the use of a client-side cache does not always improve, but may also degrade performance, for example when the workload is not file system metadata-intensive (i.e. consists of few large files and directories as our IOzone experiments). Such workloads are typical to many parallel applications. We believe that the design of a client-side cache involves tradeoffs between network latencies, consistency and recovery protocol overheads and scalability. We consider this outside the scope of this thesis and we intend to work on it in the future.

Another limitation of RIBD is that, after a global failure, the most recent recovery point is the most recent global version captured. Data written after this event will be discarded. This limitation is inherent to any asynchronous consistency scheme, either versioning or asynchronous logging, and can be solved using synchronous metadata updates. Increasing the versioning or log flush frequency can mitigate the problem, which is essentially a performance tradeoff.

In this work we have not implemented and validated the roll-back and recovery procedures for disk server failures or global failures (described in Sections 6.5.1, 6.5.5 and 6.5.6) in real failure scenarios. However, our belief that they are correct is based on the fact that versioning does not overwrite data, but keeps many previous versions available. Based on that assumption we are able to recover because our technique relies on the fact that we keep all data modification on the disk. Since our focus is examining scalability and protocol overheads in the failure-free path, not the recovery, we do not implement and evaluate the recovery procedures.

Furthermore, as discussed in Section 6.5.5, our resynchronization process of a failed disk server is simple and inefficient. Using the state in the versioning metadata maps it should be possible to avoid copying all the data from the replica server, resulting in much shorter rebuild time. This process, however, is beyond the scope of this work and is left for future work.

Finally, in our evaluation platform we have used a Gigabit Ethernet interconnect, which especially with TCP/IP exhibits high latencies and problems such as the Incast [Phanishayee et al., 2008]. Since RIBD aims at center environment, it would be more appropriate to use higher throughput and lower latency interconnects, such as 10-Gbit Ethernet, Infiniband [IBTA], or Myrinet [Myrinet]. We believe that providing support for additional networking platforms is straightforward due to the modular design of our framework and well-defined APIs.

6.10 Conclusions

Cluster-based storage with commodity components is a promising alternative to scaling capacity and performance of future storage systems in a cost-effective manner. However, their decentralized nature poses important challenges, especially in terms of reliability and availability. Current solutions to this problem focus mostly at the file level and result in complex systems that are difficult to design, scale, and tune.

In this Chapter we discuss how consistency issues can be addressed at the block level providing a simple abstraction. Our approach, RIBD, uses CIs, a lightweight transactional mechanism, agreement, versioning, and explicit locking, to address consistency of both replicas and metadata. We discuss in detail associated protocols and we implement RIBD on a real prototype, in the Linux kernel. We evaluate our approach using a setup of 24 nodes (12 disk and 12 file servers). Overall, our contributions in this Chapter are: (a) a taxonomy of alternative approaches and mechanisms for dealing with replica and metadata consistency, (b) the design and implementation of a real prototype, (c) the evaluation of RIBD in comparison with two popular filesystems, which shows that RIBD performs comparably to systems with weaker consistency guarantees.

Chapter 7

Conclusions and Future Work

Reality is the murder of a beautiful theory by a gang of ugly facts.

—Robert L. Glass, CACM, 39(11), 1996.

7.1 Conclusions

Our hypothesis in this dissertation is that *storage virtualization at the block-level with support for metadata management can be used to build scalable, reliable, available and customizable storage systems with advanced features*.

We have proved the hypothesis by addressing several problems in the emerging architecture of decentralized storage clusters. First, we have explored the requirements and merits of advanced block-level virtualization functions, by providing transparent, low-overhead versioning at the block level. Second, we have dealt with the issues of supporting extensions with advanced, metadata-intensive functionality at the block-level, and the flexible creation of customized application-specific storage volumes. Third, we have dealt with distributed storage resource sharing and management and provided mechanisms for concurrent, scalable sharing of block-level storage. Finally, we have addressed the issues of providing availability, replica consistency and recovery after failures in a decentralized storage cluster.

Overall, our contributions in this thesis are:

- We explore the usefulness, the mechanisms and the overheads associated with metadata-intensive functions at the block level, with our work on *block-level versioning*. We find that block-level versioning has low overhead and offers advantages over filesystem-based approaches, such as

transparency and simplicity.

- We design, implement and evaluate a *block-level storage virtualization framework* with support for metadata-intensive functions. We show that our framework, Violin: (i) significantly reduces the effort to introduce new functionality in the block I/O stack of a commodity storage node, and (ii) provides the ability to combine simple virtualization functions into hierarchies with semantics that can satisfy diverse application needs.

- We examine how extensible, block-level I/O paths can be supported over decentralized storage clusters. We extend our single-node virtualization infrastructure, Violin, to Orchestra. Our extended framework allows for virtual hierarchies to be distributed almost arbitrarily over application and storage nodes, providing a lot of flexibility in the mapping of system functionality to available resources.

- We design and implement locking and allocation mechanisms for concurrent, scalable sharing of block-level storage. Our framework supports shared dynamic metadata at the storage node side, but not at the application servers.

- We address consistency issues at the block level, providing simple abstractions and necessary protocols. Our approach, RIBD, uses *consistency intervals (CIs)*, a lightweight transactional mechanism, agreement, block-level versioning, and explicit locking, to address consistency of both replicas and metadata. We implement RIBD on a real prototype, in the Linux kernel. In the same area, we provide a taxonomy of alternative approaches and mechanisms for dealing with replica and metadata consistency. Finally, we evaluate RIBD in comparison with two popular filesystems, showing that RIBD performs comparably to systems with weaker consistency guarantees.

7.2 Future Work

Through our work we have identified interesting areas for future work on storage virtualization and metadata-intensive off-loading. We present these topics next.

7.2.1 Data and Control Flow from Lower to Higher Layers

Traditionally, control and data in the I/O stack flows from higher to lower layers, for example from the application to the filesystem, then to the block-layer stack and to the hardware device drivers. We

believe that in some cases, it makes sense to initiate "upcalls" from a lower layer to a higher one, such as from a lower layer in the block I/O stack to a higher block layer or even to the filesystem on top. We believe that support for upcalls would enable interesting functionality in the block-level stack (e.g. client-side consistent block-level caching), and would be an interesting topic for future research.

7.2.2 Block-level Caching and Consistent Client-side Caches

Data and metadata caching has been traditionally placed at the filesystem level. The various caches, such as the buffer cache, inode cache and directory entry caches, have been designed for the purposes of a filesystem and are normally accessible only by it. This cache placement, in combination with block-level virtualization, creates problems with data and metadata caching for the storage node side, since data no longer need to pass through the filesystem layers and have thus no access to the built-on OS caching mechanisms. We believe that exploring such issues is an interesting topic for future work, especially since we expect block-level caching to significantly boost the performance of virtualized block-level storage systems. Finally, we believe that caching at the client (application server) side is worthwhile to explore, however this is a more complex topic and probably requires support for "upcalls" in order to provide consistency.

7.2.3 Support for High-Availability of Client-side Metadata

The systems we have explored in this thesis, provide support for redundant paths and resources between each client and the storage node side. However, we have not explored redundancy for providing high-availability at the client side, that is to support active replication of requests between two clients, so that in case of a single client failure the system continues to be available. We believe that this is an interesting topic for future work.

7.2.4 Security for Virtualized Block-level Storage

The topic of security mechanisms for decentralized block-level storage was not within the scope of this thesis, however we believe it is an interesting topic for future work. Significant work on back-end storage security support has been presented in the NASD [Gibson et al., 1998] with object-level capabilities. We believe, however, that it is worthwhile to explore whether the same concepts would be applicable to the finer granularity of block-level storage.

7.2.5 New methods for ensuring metadata consistency and persistence

Block-level metadata consistency and persistence are critical to preventing data loss or corruption, however, providing such guarantees typically incurs high performance overheads. The problem is currently solved using synchronous metadata consistency schemes (logging or versioning). To mitigate this impact, high-performance non-volatile memory (e.g. NVRAM or battery-backed DRAM) is used in high-end, expensive systems. We believe that new approaches to addressing this problem are an important topic for future work.

7.3 Lessons Learned

During the course of this thesis we have been involved in the design, implementation and evaluation of several storage system prototypes. In our experience decentralized storage architectures have a lot of potential for scaling storage capacity and performance in a cost-effective manner. However, the lack of central controllers poses important dependability challenges. For this reason few existing products are currently based on this architecture [IBM Corp., 2008b; Hoffman, 2007; HYDRAstor, 2008].

Through our work we have realized that *metadata at the block level are key enablers of advanced features* and we believe that *they will be increasingly used in future systems*. There are currently emerging systems that use block level metadata. Examples range from flash-based SSDs that need large amounts of dynamic metadata to remap blocks in order to achieve performance and wear-leveling, to the use of block-level metadata to support features, such as thin provisioning [Compellent, 2008; IBM Corp., 2008c], snapshots [IBM Corp., 2008d], volume management [EVMS; GEOM; Teigland and Mauelshagen, 2001], data deduplication [Quinlan and Dorward, 2002], block remapping [English and Alexander, 1992; Wang et al., 1999; Wilkes et al., 1996; Sivathanu et al., 2003], and logging [Wilkes et al., 1996; Stodolsky et al., 1994].

Metadata at the block level is a fundamental change with major repercussions for the traditional I/O stack. The need for consistency, coherency, and caching mechanisms for block-level metadata will result in duplicating file-system-level mechanisms at the block level. If the file system and block-level storage evolve separately at each side of the block API, the complexity of the storage hierarchy will increase significantly. In this case, the different mechanisms, assumptions and optimizations in the two separate layers (file and block), will lead to limited efficiency, dependability, and behavior that is hard to understand. Instead we believe that the layers in the I/O stack, either at the file or block level, will

need to operate in harmony, being aware of each other's behavior, optimizations and assumptions. For example, if the block layer performs block remapping without being aware of the assumptions of the file layer, the latter's data placement optimizations will become irrelevant with significant performance issues, as we have seen also in this thesis.

An alternative approach is to eliminate block-level virtualization and perform all virtualization at the file system level, as has been done with ZFS [Bonwick and Moore, 2008] and GPFS [Schmuck and Haskin, 2002]. However, this approach leads to monolithic systems that will become complex and unmanageable as they continue to incorporate more features. Furthermore, such systems will not work effectively on virtualized block storage, because some of their optimizations (e.g. data placement) will become irrelevant. Finally, our experience tells us that one-for-all solutions rarely work as expected, especially when they are controlled be a single vendor.

In a different direction, semantically-smart disk systems, have explored the possibility of the block-level subsystem to infer essential information about the higher layers, such as I/O access patterns, block liveness data or file system structure and block relationships [Arpaci-Dusseau et al., 2006]. Such information would mitigate the problem and enable off-loading of many useful tasks to the block level, e.g. secure deletion, data placement optimizations, and garbage collection. However, this gray-box approach works well only in cases where hints are adequate and there is no need for precise information. For example, making assumptions about file system block liveness at the block level incurs a risk of false inference that may lead to data loss.

Our view is that maintaining the traditional block API will continue to hinder off-loading virtualization functions at the block-level. Virtual block devices can certainly perform several optimizations by observing the file layer's behavior, as has been demonstrated in the semantically-smart disk approach. However, in other cases, such as block liveness, there is need for exact, accurate information from the file layer that would enable off-loading of significant functionality, such as secure block deletion and disk space savings in the case of versioning and thin provisioning. Thus, in order to facilitate off-loading and better integration of the file and block layers, we believe that there's a need for a new, redesigned storage stack standard with richer APIs between I/O layers.

Finally, we believe that this is a period of change for personal storage, as people rely more and more on stored digital information. As individuals (and small businesses) do not have the resources to efficiently manage data storage, there is a question whether storage will become mostly a (commodity) utility provided by storage providers, or due to social (e.g. concerns for the privacy of personal data)

and/or technical (e.g. network performance and availability) reasons will continue the current trend of storing data at the owner's physical space. How this question will be answered depends both on technology and on social implications and it will be interesting to see how issues from both sides will affect the evolution of storage.

Bibliography

Michael Abd-El-Malek, II William V. Courtright, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa Minor: Versatile Cluster-Based Storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technology (FAST '05)*, Berkeley, CA, USA, December 2005. USENIX Association.

Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 81–91, San Jose, California, October 3–7, 1998. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

Marcos K. Aguilera, Susan Spence, and Alistair Veitch. Olive: Distributed point-in-time branching storage for real systems. In *Proceedings of the 3^{rd} Symposium on Networked System Design and Implementation (NSDI '06)*, 2006.

Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proc. of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, pages 159–174, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5.

M. Ajtai, R. Burns, R. Fagin, D. Long, and L. Stockmeyer. Compactly Encoding Unstructured Inputs with Differential Compression. *Journal of the ACM*, 39(3), 2002.

Cuneyt Akinlar and Sarit Mukherjee. A Scalable Distributed Multimedia File System Using Network Attached Autonomous Disks. In *IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 180–, 2000.

K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, USA, June 2000. USENIX Association.

Khalil Amiri, Garth A. Gibson, and Richard Golding. Highly Concurent Shared Storage. In *Proceedings of* 20^{th} *International Conference on Distributed Computing Systems (ICDCS '00)*, pages 298–307, Taipe, Taiwan, R.O.C, April 2000. IEEE Computer Society.

Eric Anderson. Results of the 1995 SANS Survey. *;login, The Usenix Association Newsletter*, 20(5), October 1995.

Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)*, pages 175–188, Berkeley, CA, January 28–30 2002. USENIX Association.

Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14 (1):41–79, February 1996.

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Lakshmi N. Bairavasundaram, Timothy E. Denehy, Florentina I. Popovici, Vijayan Prabhakaran, and Muthian Sivathanu. Semantically-Smart Disk Systems: Past, Present, and Future. *Sigmetrics Performance Evaluation Review (PER)*, 33(4): 29–35, March 2006.

C. Attanasio, M. Butrico, C. Polyzois, S. Smith, and J. Peterson. Design and implementation of a recoverable virtual shared disk. Technical Report IBM Research Report RC 19843, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1994.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, and Rolf Neugebauer. Xen and the art of virtualization. In *Proc. of the nineteenth ACM symposium on Operating systems principles (SOSP19)*, pages 164–177, October 2003.

Richard Barker and Paul Massiglia. *Understanding Storage Area Networks*. John Wiley & Sons, Inc., 2002.

Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 117–128, November 2000.

Stephen J. Bigelow. Application-aware storage promises intelligence and automation. http://searchstorage.techtarget.com/ generic/ 0,295582,sid5_ gci1267371,00.html, August 2007.

Vasken Bohossian, Chenggong C. Fan, Paul S. LeMahieu, Marc D. Riedel, Jehoshua Bruck, and Lihao Xu. Computing in the RAIN: A Reliable Array of Independent Nodes. *IEEE Trans. Parallel Distrib. Syst.*, 12(2):99–114, 2001.

Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/ os/ community/ zfs/ docs/ zfs_last.pdf, 2008.

M. R. Brown, K. N. Kolling, and E. A. Taft. The Alpine File System. *ACM Transactions in Computing Systems*, 3(4):261–293, 1985. ISSN 0734-2071.

M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Technical Report SRC-RR-124, HP Labs, 1994.

Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proc. of the 4th Annual Linux Showcase and Conference*, 2000.

S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode File System. In *Proc. of the USENIX Winter 1992 Technical Conference*, pages 43–60, San Fransisco, CA, USA, 1992.

Russell Coker. Bonnie++ I/O Benchmark. http:// www.coker.com.au/ bonnie++.

Compellent. Storage Center Data Sheet. http://www.compellent.com/∼/media/ com/ Files/ Datasheets/DS_FT_021908.ashx, 2008.

W. V. CourtrightII, G. A. Gibson, M. Holland, and J. Zelenka. RAIDframe: Rapid Prototyping for Disk Arrays. *Proceedings of the 1996 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 24(1):268–269, May 1996.

Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making Backup Cheap and Easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, Berkeley, CA, December 9–11 2002. The USENIX Association.

Flaviu Cristian and Christof Fetzer. The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

Miguel de Icaza, Ingo Molnar, and Gadi Oxman. The linux raid-1,-4,-5 code. In *LinuxExpo*, April 1997.

Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proc. of 14th SOSP*, pages 15–28, 1993.

T. Denehy, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 177–190, June 2002.

Murthy Devarakonda, Bill Kish, and Ajay Mohindra. Recovery in the Calypso file system. *ACM Transactions in Computing Systems*, 14(3):287–310, 1996. ISSN 0734-2071.

EMC Enginuity. EMC. Enginuity: The Storage Platform Operating Environment (White Paper). http://www.emc.com/pdf/techlib/c1033.pdf.

EMC Symmetrix. EMC: Introducing RAID 5 on Symmetrix DMX. http://www.emc.com/ products/ systems/ enginuity/ pdf/ H1114_Intro_ raid5_ DMX_ldv.pdf.

Robert English and Stephnov Alexander. Loge: A Self-Organizing Disk Controller. In *Proceedings of the Winter 1992 USENIX Conference*, Berkeley, CA, 1992. The USENIX Association.

Sadik C. Esener, Mark H. Kryder, William D. Doyle, Marvin Keshner, Masud Mansuripur, and David A. Thompson. WTEC Panel Report on The Future of Data Storage Technologies. International Technology Research Institute. World Technology (WTEC) Division, June 1999.

EVMS. Enterprise Volume Management System. http://evms.sourceforge.net.

Fibre Channel. Fibre Channel Industry Association. Executive Summary. http://www.fibrechannel.org/ OVERVIEW/ index.html.

Michail D. Flouris and Angelos Bilas. Clotho: Transparent Data Versioning at the Block I/O Level. In *12th NASA Goddard & 21st IEEE Conference on Mass Storage Systems and Technologies (MSST2004)*, April 2004.

Michail D. Flouris and Angelos Bilas. Violin: A Framework for Extensible Block-level Storage. In *Proceedings of 13th IEEE/NASA Goddard (MSST2005) Conference on Mass Storage Systems and Technologies*, Monterey, CA, April 11–14 2005.

Michail D. Flouris, Stergios V. Anastasiadis, and Angelos Bilas. Block-level Virtualization: How far can we go? In *Proceedings of Second IEEE-CS International Symposium on Global Data Interoperability - Challenges and Technologies*, Sardinia, Italy, June 20–24 2005.

Michail D. Flouris, Renaud Lachaize, and Angelos Bilas. Using Lightweight Transactions and Snapshots for Fault-Tolerant Services Based on Shared Storage Bricks. In *Proc. of the International Workshop on High Performance I/O Techniques and Deployment of Very Large Scale I/O Systems (HiperIO '06)*, September 2006.

Michail D. Flouris, Renaud Lachaize, and Angelos Bilas. Orchestra: Extensible Block-level Support for Resource and Data Sharing in Networked Storage Systems. In *Proc. of the 14th IEEE International Conference on Parallel and Distributed Systems (ICPADS'08)*, Melbourne, Victoria, Australia, December 2008.

Gartner Group. Total Cost of Storage Ownership – A User-oriented Approach, September 2000.

GEOM. FreeBSD: GEOM Modular Disk I/O Request Transformation Framework. http://kerneltrap.org/node/view/454.

Garth A. Gibson and John Wilkes. Self-managing network-attached storage. *ACM Computing Surveys*, 28(4es):209–209, December 1996. ISSN 0360-0300.

Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, California, October 3–7, 1998. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient Consistency for Erasure-coded Data via Versioning Servers. Technical Report CMU-CS-03-127, Carnegie Mellon University, April 2003.

Jim Gray. Greetings from a Filesystem User. Keynote Address at the 4^{th} USENIX FAST Conference, December 2005.

Jim Gray. What Next? A Few Remaining Problems in Information Technology (Turing Lecture). In *ACM Federated Computer Research Conferences (FCRC)*, May 1999.

Jim Gray. Storage Bricks Have Arrived. Invited Talk at the First USENIX Conference on File And Storage Technologies (FAST '02), 2002.

R. Grimm, W.C. Hsieh, M.F. Kaashoek, and W. de Jonge. Atomic Recovery Units: Failure Atomicity For Logical Disks. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 26–37. IEEE Computer Society, 1996.

R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. of the 11th ACM Symposium on Operating Systems Principles (SOSP'87)*, pages 155–162, New York, NY, USA, 1987. ACM. ISBN 0-89791-242-X.

J. H. Hartman, I. Murdock, and T. Spalink. The Swarm Scalable Storage System. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*. IEEE Computer Society, June 1999.

John H. Hartman and John K. Ousterhout. Zebra: A Striped Network File System. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1992.

J.S. Heidemann and G.J. Popek. File System Development with Stackable Layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

Andy Hisgen, Andrew Birrell, Charles Jerian, Timothy Mann, and Garret Swart. New-value logging in the Echo replicated file system. Technical Report 104, Xerox, Palo Alto CA (USA), 1993.

Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, 17–21 1994. ISBN 1-880446-58-8.

Patrick Hoffman. NEC Storage Platform Reduces Cost, Complexity. http://www.eweek.com/ article2/ 0,1759,2103110,00.asp, March 2007.

Sam Hopkins and Brantley Coile. AoE (ATA over Ethernet). http://www.coraid.com/ documents/ AoEr10.txt.

HP OpenView. Hewlett-Packard: OpenView Storage Area Manager. http://h18006. www1.hp.com/ products/ storage/ software/ sam/ index.html.

Chuang hue Moh and Barbara Liskov. Timeline: A high performance archive for a distributed object store. In *In* 1^{st} *Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. Physical File System Backup. In *Proc. of the 3rd USENIX Symposium on Operating Systems Design and Impl. (OSDI99)*, February 1999.

HYDRAstor. NEC Corp. http://www.necam.com/HYDRAstor, 2008.

IBM Corp. Collective Intelligent Bricks (CIB) Project. http://www.almaden.ibm.com/ StorageSystems/ autonomic_storage/ clockwork/ index.shtml, 2008a.

IBM Corp. XIV Storage. http://www.xivstorage.com, 2008b.

IBM Corp. Delivering the Thin Provisioning Advantage with XIV's Nextra Architecture White Paper. http://www.xivstorage.com/ materials/ white_papers/ nextra_thin_provisioning_white_paper.pdf, 2008c.

IBM Corp. XIV Nextra Snapshot Implementation White Paper. http://www.xivstorage.com/ materials/ white_papers/ nextra_snapshot_white_paper.pdf, 2008d.

IBTA. InfiniBand Trade Association. About InfiniBand. http://www.infinibandta.org/ about/ about_infiniband/.

IDC. International Data Corporation. The Diverse and Exploding Digital Universe - An Updated Forecast of Worldwide Information Growth Through 2011. White Paper - sponsored by EMC, http://www.emc.com/ leadership/ digital-universe/ expanding-digital-universe.htm, March 2008.

Iometer. The I/O Performance Analysis Tool. http://www.iometer.org.

iSCSI. IETF IP Storage Working Group. iSCSI Internet Draft. http://www.ietf.org/ internet-drafts/ draft-ietf-ips-iscsi-20.txt.

James E. Johnson and William A. Laing. Overview of the Spiralog File System. *Digital Technical Journal*, 8(2):5–14, 1996. ISSN 0898-901X.

C. Jurgens. Fibre Channel: A Connection to the Future. *IEEE Computer*, 28(8):88–90, 1995.

M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector Briceno, Russel Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Janotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, 1997.

Jeffrey Katcher. PostMark: A New File System Benchmark. http://www.netapp.com/tech_library/ 3022.html.

Kimberly Keeton and John Wilkes. Automatic design of dependable data storage systems. In *Proc. of Workshop on Algorithms and Architectures for Self-managing Systems*, pages 7–12, San Diego, CA, June 2003.

Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A Case for Intelligent Disks (IDISKs). *SIGMOD Record*, 27(3):42–52, 1998.

Deepak R. Kenchammana-Hosekotea, Richard A. Golding, Claudio Fleiner, and Omer A. Zaki. The Design and Evaluation of Network RAID protocols. Research report RJ 10316, IBM Almaden Research Center, March 2004.

Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

Orran Krieger and Michael Stumm. HFS: a performance-oriented flexible file system based on building-block compositions. *ACM Transactions in Computing Systems*, 15(3):286–321, 1997. ISSN 0734-2071.

Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. Vaxcluster: a closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2), 1986.

John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *Proceedings of ACM ASPLOS*, November 2000.

Neal Leavitt. Application Awareness Makes Storage More Useful. *IEEE Computer*, 41(7):11–13, 2008. ISSN 0018-9162.

Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, *Computer Architecture News*, volume 24:Special, pages 84–93. ACM SIGARCH/SIGOPS/SIGPLAN, October 1996.

Greg Lehey. The Vinum Volume Manager. In *Proceedings of the FREENIX Track (FREENIX-99)*, pages 57–68, Berkeley, CA, June 6–11 1999. USENIX Association.

Michael Lesk. How Much Information Is There In the World? http://www.lesk.com/ mlesk/ ksg97/ ksg.html, 1997.

Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Liuba Shrira. Replication in the Harp File System. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 226–238, New York, NY, USA, 1991. ACM. ISBN 0-89791-447-3.

C. R. Lumb, R. Golding, and G. R. Ganger. D-SPTF: Decentralized Request Distribution in Brick-Based Storage Systems. In *Proceedings of the 11th ACM ASPLOS Conference*, Boston, MA, USA, October 2004.

Peter Lyman and Hal R. Varian. How Much Information. http://www.sims.berkeley.edu/ how-much-info-2003, 2003.

John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation (OSDI-04)*. USENIX, December 6–8 2004.

J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM Storage Tank - A heterogeneous scalable SAN file system. *IBM Systems Journal*, 42(2):250–267, 2003.

M. Mesnier, G. R. Ganger, and E. Riedel. Object-Based Storage. *IEEE Communications Magazine*, 41 (8):84–90, August 2003.

James G. Mitchell and Jeremy Dion. A comparison of two network-based file servers. *Communications of the ACM*, 25(4):233–245, 1982. ISSN 0001-0782.

Joe Moran, Bob Lyon, and Legato Systems Incorporated. The Restore-o-Mounter: The File Motel Revisited. In *Proc. of USENIX '93 Summer Technical Conference*, June 1993.

David Mosberger and Larry L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proc. of the 2nd USENIX Symposium on Operating Systems Design and Impl. (OSDI96)*, October 28–31 1996.

S. J. Mullender and A. S. Tanenbaum. Immediate files. *Software Practice and Experience*, 14(4): 365–368, 1984.

Myrinet. Myricom, Inc., Myrinet Overview. http://www.myri.com/ myrinet/ overview/.

Nils Nieuwejaar and David Kotz. The Galley Parallel File System. In *Proc. of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, 1996. ACM Press.

William D. Norcott and Don Capps. IOzone Filesystem Benchmark. http://www.iozone.org.

Michael A. Olson. The design and implementation of the inversion file system. In *Proc. of USENIX '93 Winter Technical Conference*, January 1993.

S. W. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

Walter Oney. Programming the Microsoft Windows Driver Model, Second Edition. http://www.microsoft.com/ mspress/ books/ 6262.asp.

D. Patterson. The UC Berkeley ISTORE Project: bringing availability, maintainability, and evolutionary growth to storage-based clusters. http://roc.cs.berkeley.edu, January 2000.

R. Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)*, pages 117–130, Berkeley, CA, January 28–30 2002. USENIX Association.

Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In *Proc. of the 6th USENIX Conference on File and Storage Technologies (FAST08)*, pages 175–188, 2008.

Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from bell labs. In *Proc. of the Summer UKUUG Conference*, 1990.

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

Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, April 2005a.

Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005b.

K. W. Preslan, A. Barry, J. Brassow, M. Declerk, A. J. Lewis, A. Manthei, B. Marzinski, E. Nygaard, S. Van Oort, D. Teigland, M. Tilstra, S. Whitehouse, and M. O'Keefe. Scalability and Recovery in a Linux Cluster File System. In *Proceedings of the 4th Annual Linux Showcase and Conference*, College Park, Maryland, USA, October 2000.

Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigl, and Matthew T. O'keefe. A 64-bit, shared disk file system for Linux. In *16th IEEE Conference on Mass Storage Systems and Technologies (MSST '99)*, March 1999.

PVFS2. PVFS2 Project Home Page. http://www.pvfs.org.

Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Data Storage. In *Proceedings of FAST '02*, pages 89–102. USENIX, January 28–30 2002.

ReiserFS. Namesys. http://www.namesys.com.

W. D. Roome. 3DFS: A Time-Oriented File Server. In *Proceedings of USENIX '92 Winter Technical Conference*, January 1992.

Yasushi Saito, Svend Frølund, Alistair C. Veitch, Arif Merchant, and Susan Spence. FAB: building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pages 48–58. ACM Press, 2004.

Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding When to Forget in the Elephant File System. In *Proceedings of 17th SOSP*, December 1999.

SATA. Serial ATA Working Group. Serial ATA Specifications. http://www.serialata.org.

Paul W. Schermerhorn, Robert J. Minerick, Peter W. Rijks, and Vincent W. Freeh. User-level Extensibility in the Mona File System. In *Proc. of Freenix 2001*, pages 173–184, June 2001.

Frank Schmuck and Roger Haskin. GPFS: A Shared-disk File System for Large Computing Centers. In *USENIX Conference on File and Storage Technologies*, pages 231–244, Monterey, CA, January 2002.

Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX '00)*, pages 71–84, Berkeley, CA, June 18–23 2000. USENIX Association.

R. A. Shillner and E. W. Felten. Simplifying Distributed File Systems Using a Shared Logical Disk. Technical Report TR-524-96, Princeton University, Princeton, NJ, USA, October 1996.

Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Evolving RPC for Active Storage. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 264–276, San Jose, CA, October 2002.

Muthian Sivathanu, Lakshmi Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or Death at Block-Level. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, December 2004.

Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A logic of file systems. In *Proceedings of the 4^{th} USENIX Conference on File and Storage Technologies (FAST'05)*, Berkeley, CA, USA, 2005a. USENIX Association.

Muthian Sivathanu, Lakshmi Bairavasundaram, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Database-aware semantically-smart storage. In *Proceedings of the 4^{th} Conference on USENIX Conference on File and Storage Technologies (FAST'05)*, pages 18–18, Berkeley, CA, USA, 2005b. USENIX Association.

Muthian Sivathanu, Vijayan Prabhakaran, Florentina Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the FAST '03 Conference on File and Storage Technologies (FAST-03)*. USENIX Association, April 2003.

Dale Skeen. Nonblocking commit protocols. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142, New York, NY, USA, 1981. ACM. ISBN 0-89791-040-0.

Glenn C. Skinner and Thomas K. Wong. Stacking/ vnodes: A progress report. In *Proc. of the USENIX Summer 1993 Technical Conference*, pages 161–174, Berkeley, CA, USA, June 1993. USENIX Association.

EMC SnapView. Snapview data sheet. http://www.emc.com/pdf/ products/ clariion/SnapView2_DS.pdf.

SNIA. Storage Networking Industry Association End User Council Survey - Top Ten Pain Points Report. http://www.snia.org/ collateral/ TopTenPainPoints_Report_Final_040823.pdf.

Craig A.N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata Efficiency in Versioning File Systems. In *Proceedings of the FAST '03 Conference on File and Storage Technologies (FAST-03)*, Berkeley, CA, April 2003. The USENIX Association.

Daniel Stodolsky, Mark Holland, II William V. Courtright, and Garth A. Gibson. Parity-logging disk arrays. *ACM Trans. Comput. Syst.*, 12(3):206–235, 1994. ISSN 0734-2071.

Storactive. Delivering Real-Time Data Protection & Easy Disaster Recovery for Windows Workstations. http://www.storactive.com/files/ Storactive_Whitepaper.doc, January 2002.

John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 165–180, Berkeley, CA, October 23–25 2000. The USENIX Association.

Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proc. of USENIX 2001 Annual Technical Conference*, June 2001.

Sun Microsystems. Instant Image White Paper. http://www.sun.com/ storage/ white-papers/ ii_soft_arch.pdf, 2008.

Sun Microsystems. Lustre File System: High-Performance Storage Architecture and Scalable Cluster File System. http://www.sun.com/ software/ products/ lustre/ docs/ lustrefilesystem_ wp.pdf, December 2007.

Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proc. of the 1996 USENIX Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

David Teigland and Heinz Mauelshagen. Volume managers in linux. In *Proceedings of USENIX 2001 Technical Conference*, June 2001.

Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *Proc. of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 224–237, New York, October 5–8 1997. ACM Press.

Stephen Tweedie. Ext3, journaling filesystem. In *Presentation at Ottawa Linux Symposium*, Ottawa Congress Centre, Canada, July 2000.

Mustafa Uysal, Anurag Acharya, and Joel Saltz. Evaluation of Active Disks for Decision Support Databases. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 337–348, Toulouse, France, January 8–12, 2000. IEEE Computer Society TCCA.

Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A Framework for Protocol Composition in Horus. In *Symposium on Principles of Distributed Computing*, pages 80–89, 1995.

Alistair C. Veitch, Eric Riedel, Simon J. Towers, and John Wilkes. Towards Global Storage Management and Data Placement. In *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII). May 20–23, 2001, Schloss Elmau, Germany*, pages 184–184, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2001. IEEE Computer Society Press. ISBN 0-7695-1040-X.

Veritas. Flashsnap. http://eval.veritas.com/ downloads/ pro/ fsnap_ guide_ wp.pdf, a.

Veritas. Storage Foundation(TM). http://www.veritas.com/ Products/ www?c=product&refId=203, b.

Veritas. Volume Manager(TM). http://www.veritas.com/vmguided, c.

Werner Vogels, Dan Dumitriu, Ashutosh Agrawal, Teck Chia, and Katherine Guo. Scalability of the Microsoft Cluster Service. In *WINSYM'98: Proceedings of the 2nd conference on USENIX Windows NT Symposium*, Seattle, WA, USA, August 1998. USENIX Association.

Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, pages 29–43, 1999.

Ralph O. Weber (Editor). Information Technology – SCSI Object-Based Storage Device Commands (OSD), Revision 10. Technical Council Proposal Document T10/1355-D, Technical Committee T10, July 2004.

Ralph O. Weber (Editor). Information Technology – SCSI Object-Based Storage Device Commands-2 (OSD-2), Revision 2. T10 Working Draft T10/1729-D, Technical Committee T10, July 2007.

Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.

Brian S. White, Michael Walker, Marty Humphrey, and Andrew S. Grimshaw. LegionFS: a secure and scalable file system supporting cross-domain high-performance applications. In *Proc. of the 2001 ACM/IEEE conference on Supercomputing (Supercomputing'01)*, pages 59–59, New York, NY, USA, 2001. ACM. ISBN 1-58113-293-X.

John Wilkes. Traveling to Rome: QoS Specifications for Automated Storage System Management. In *Proceedings of the International Workshop on QoS (IWQoS'2001). Karlsruhe, Germany*, June 2001.

John Wilkes, Richard A. Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

Jake Wires and Michael J. Feeley. Secure file system versioning at the block level. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 203–215, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3.

Theodore M. Wong, Richard A. Golding, Joseph S. Glider, Elizabeth Borowsky, Ralph A. Becker-Szendy, Claudio Fleiner, Deepak R. Kenchammana-Hosekote, and Omer A. Zaki. Kybos: Self-management for distributed brick-based storage. Technical Report RJ10356, IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, CA, August 2005.

Xdd. Version 6.3, I/O Performance Inc. http://www.ioperformance.com.

Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 273–288, December 6-8, 2004, San Francisco, California, USA. USENIX Association, 2004.

H. Yokota. Performance and Reliability of Secondary Storage Systems. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI2000), Invited Paper*, pages 668–673, July 2000.

Erez Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 55–70, Berkeley, CA, June 18–23 2000. USENIX Association.

Seth Zirin and Intel Corp. Joe Bennett, Principal Engineers. Advanced Switching Overview. http://download.intel.com/ netcomms/ as/ devcon_as_overview.pdf.

J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23:337–343, May 1977.