Block-level Virtualization: How far can we go?

Michail D. Flouris[†]**, Stergios V. Anastasiadis**[*] **and Angelos Bilas**[‡]
Institute of Computer Science (ICS)
Foundation for Research and Technology - Hellas
P.O.Box 1385, Heraklion, GR 71110, Greece
{ *flouris, stergios, bilas* }*@ics.forth.gr*

Abstract

In this paper, we present our vision on building large-scale storage systems from commodity components in a data center. For reasons of efficiency and flexibility, we advocate maintaining sophisticated data management functions behind a block-based interface. We anticipate that such an interface can be customized to meet the diverse needs of end-users and applications through the extensible storage system architecture that we propose.

1. Introduction

Data storage is becoming increasingly important as larger amounts of data assets need to be stored for archival or online-processing purposes. 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 continuous system operation uninterrupted from device faults, easier storage management, and ultimately lower cost of purchase and maintenance.

Our target is primary a data-center environment, where application servers and user workstations are able to access the storage system through potentially different network paths and protocols. In this environment, although storage consolidation has potential for lower costs and more efficient storage management, its success depends on the ability of the system architecture to face three main challenges:

1. The first significant challenge is the platform, that is how to provide system scalability with minimal hardware costs.
2. The second crucial challenge is the software infrastructure required for (i) virtualizing, (ii) sharing, and (iii) managing the physical storage in a data center. This includes the virtualization capability to mask applications from each other and to facilitate storage management tasks. It also includes the significant objective of lowering administration costs through automated administration. Furthermore, the software infrastructure should provide monitoring and dynamic reconfiguration mechanisms to allow the definition and implementation of high-level quality-of-service policies.
3. Finally, the third challenge is efficient storage sharing between remote data-centers, enabling global-scale storage distribution.

Next we explore each of these challenges in more detail.

1.1. Scalable, Low-cost Platform

Today, commercial storage systems mostly use expensive custom-made hardware components and proprietary protocols. We advocate building scalable low-cost storage systems from storage clusters composed of common off-the-shelf components. Current commodity computers and networks have (or will soon have) significant computing power and network performance, at a fraction of the cost of custom storage-specific hardware (controllers or networks). Using such components, we can build low-cost storage clusters with enough capacity and performance to meet practically any storage needs, and scale it economically. This idea is not new. It has been proposed and predicted a few years ago by several researchers and vendors [8, 9]. We think that the necessary commodity technologies are maturing now with the emergence of new standards, protocols and interconnects, such as Serial ATA, PCI-X/Express/AS and iSCSI. For the rest of this discussion, we assume that the future storage will be based on clusters built of commodity PCs (e.g. x86, SATA disks) and commodity interconnects (e.g. gigabit Ethernet, PCI-Express/AS) accessible through standard protocols (e.g. SCSI).

A commodity PC with PCI-X or PCI-Express can currently accommodate at least 2 TBytes of storage using commodity I/O controllers with 8 or 16 disks SATA disks each. The PCI-X bus can support aggregate throughput in excess of 800 MBytes/sec to the disks, while gigabit Ethernet NICs and switches are an inexpensive means for interconnecting the storage nodes. The nodes can also run an existing high-performance operating system such as Linux or FreeBSD for a very low cost. Similarly, interconnect technology that will allow efficient scaling is currently being developed (e.g. PCI-Express/AS). Thus, we assume that the commodity components to build such storage platform are or will be available at a low cost. We believe that the next missing component for building cost-effective storage systems is addressing challenge 2. This will provide the software infrastructure that

[†] Also a graduate student at the Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3G4, Canada.

[*] Also, with the Department of Electronic and Computer Engineering, Technical University of Crete, Chania, GR 73100, Greece.

[‡] Also, with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece.

will run on top, integrating storage resources into a system that provides flexibility, sharing, and automated management. This is the main motivation for our work.

1.2. Software Infrastructure

A consolidated storage system requires increased flexibility to serve multiple applications and satisfy diverse needs in both storage management and data access. Moreover, as we accumulate massive amounts of storage resources behind a single system image, the automation of the administration tasks becomes crucial for sustaining efficient and cost-effective operation. Continuous unobtrusive measurement of the device efficiency and the user-perceived performance should drive data reorganization and device reconfiguration towards load-balanced system operation. Finally, policy-based data administration should meet preagreed performance objectives for individual applications. These are the challenges for the software infrastructure in a data center. We argue that such requirements can be satisfied through a virtualization infrastructure.

The term "virtualization" has been used to describe mainly two concepts: indirection and sharing. The first refers to the addition of an indirection layer between the physical resources and the logical devices accessed by the applications. Such a layer allows administrators to create and applications to access various types of virtual volumes that are mapped to physical devices, while offering higher-level semantics through multiple layers of mappings. This kind of virtualization has several advantages: (i) It enables a lot of useful management and reconfiguration operations, such as non-disruptive data reorganization and migration during the system operation. (ii) 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. (iii) It allows implementation of useful functionality, such as fault-tolerance with RAID levels, backup with volume snapshots or encryption with encryption data filters. (iv) It allows extensibility, 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 most storage systems and products [11, 13, 14] mostly support a set of predefined functions and leave freedom to the administrator in the way of combining them. That is, they provide *configuration flexibility*, but very little *functional flexibility*. For example predefined virtualization semantics include virtual volumes mapped to an aggregation of disks or RAID levels. In this category belong both research prototypes of volume managers [2, 5, 7, 12, 19] as well as commercial products [3, 4, 10, 21, 22]. 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 by providing new functionality, such as snapshots, encryption, compression, virus-scanning, application-specific data placement or content hash computation. Moreover, it is not possible to combine such new functions in a flexible manner.

The second notion of virtualization refers to the ability of sharing the storage resources across many nodes and multiple applications without compromising the performance of each application in any significant way. Even though some vendors claim that they can support such sharing, efficient virtualization remains an elusive target in large-scale systems, because of the software complexity. Maintaining consistency is complex and usually introduces significant overheads in the system.

1.3. Global-scale Sharing

In this area issues include interoperability, hiding network latencies and achieving high throughput in networks with large delay-bandwidth products. We consider this area out of the scope of our work, but we believe that the virtualization infrastructure we propose facilitates global storage connectivity and interoperability between data centers.

The rest of this paper is organized as follows. Section 2 presents our position on virtualization issues, while Section 3 discusses our approach to block-level virtualization. Finally, Section 4 discusses related work and Section 5 draws our conclusions.

2. Our Approach

Our goal in this work is to address the software infrastructure challenges, by exploring storage virtualization solutions. The term virtualization is used here to signify both an indirection layer between physical resources and virtual volume, and resource sharing. Next we describe the directions we have taken in our approach.

2.1. Block-level functionality

We advocate block-level virtualization (we use the indirection notion here), as opposed to file-level virtualization for the following reasons. First, certain functionality, such as compression or encryption, may be simpler and more efficient to provide on unstructured fixed data blocks rather than variable-size files. Second, storage hardware at the back-end has evolved significantly from simple disks and fixed controllers to powerful storage nodes [1, 8, 9] that offer block-level storage to multiple applications over a storage area network [16, 17]. Block-level virtualization modules can exploit the processing capacity of these storage nodes, where filesystems (running mainly on the application servers and the clients) cannot. Third, 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. 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.

With the traditional file vs. block abstractions, intelligent control functions remain at higher-level abstractions such as filesystems or databases running at the front-end to support application servers and client workstations. In this manner, all the processing power of the storage hardware has to be hidden behind virtual disks for transparency reasons. This imbalance that favors the front-end control leaves underutilized the hardware capabilities of the back-end. On the other hand, offloading functionality to the back-end could free up the processor at the front-end to actually run application code instead of filesystem functions. The concept is similar to using DMA and smart hardware to relieve the main processor from performing memory transfers between the main memory and the hardware on the bus (e.g. a RAID controller on

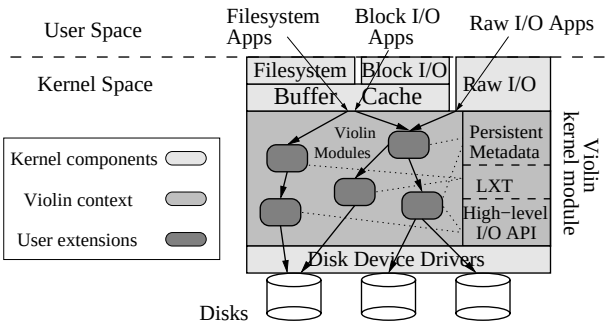

Figure 1. Violin in the OS.

the PCI). Similarly, we could offload processing and data transfers to the storage back-end. For example, instead of the client filesystem moving data over the network in order to copy a file, it could instruct the back-end to perform the copy locally at the level of data blocks. Thus, we could improve the performance of individual applications and the entire system.

By using block-level virtualization we have to restrict our system to the traditional block-level interface. A higher API, such as objects, may be better in this case and we are considering it for the future. However, we currently aim to explore the limits of the block-level API before dealing with higher abstractions.

3. Block-level Virtualization

Our work addresses block level virtualization, both for indirection and for sharing purposes. In this section we present our approach for both notions in more detail.

3.1. Indirection Virtualization

3.1.1. Single-storage-node Virtualization As a building block towards a distributed storage system with flexible and efficient virtualization support, we have designed and implemented Violin [6]. Violin (Virtual I/O Layer INtegrator), is a kernel-level framework for (i) building and (ii) combining block-level virtualization functions. Violin is 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 virtual storage hierarchy with new mechanisms and (ii) flexible combination of these mechanisms to create modular hierarchies with rich semantics. Figure 1 illustrates a high-level view of Violin in the operating system context.

The main contributions of Violin are: (i) it significantly reduces the effort of introducing new functionality in the block I/O stack of a single storage node and (ii) provides ample configuration flexibility to combine simple virtualization operators into hierarchies with semantics that can satisfy diverse application needs. To achieve configuration 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. Figure 2 shows such a device graph, where mappings between higher and lower devices are represented by arrows. Every virtual device's functionality is provided by independent virtualization modules that are linked to the main Violin framework. A linked device graph is called a hierarchy and represents practically a virtualization layer stack. 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.

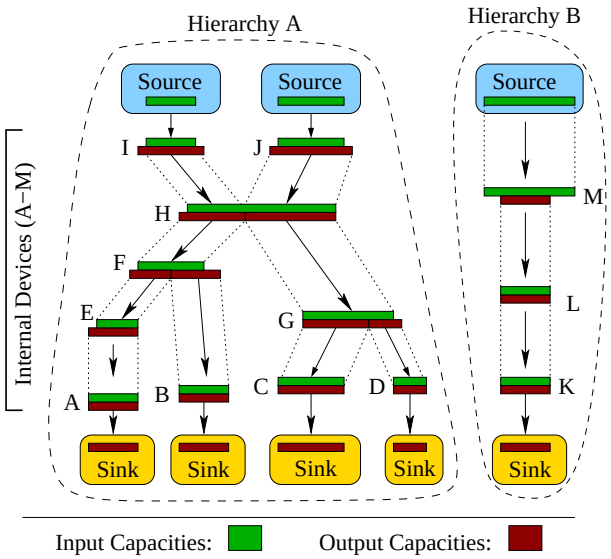

Figure 2. The virtual device graph in Violin.

Violin also 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. Additionally, Violin deals with metadata persistence of the full storage hierarchy, offloading the related complexity from individual virtual devices. It supports persistent objects for storing virtual device metadata. Violin automatically synchronizes persistent objects to stable storage, resulting in much less effort for the module developer.

Systems such as Violin can be combined with standard storage access protocols, such as iSCSI to build large-scale distributed volumes. Figure 3 shows a system with multiple storage nodes that provide a common view of the physical storage in a cluster. We believe that future, large-scale storage systems will be built in this manner to satisfy application needs at a cost-effective manner. 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 implemented various virtual modules as dynamically loadable kernel devices that bind to Violin's API. We also provide 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.

In [6], we evaluate the effectiveness of our approach 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, storage virus-scanning modules, online migration and transparent disk layout

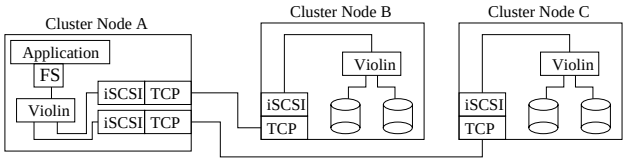

Figure 3. Violin in a distributed environment.

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 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.

3.1.2. Multi-storage-node Virtualization Our step towards a scalable virtualization platform is to enhance the Violin framework with the necessary primitives in order to allow multi-node hierarchies. We find that the primitives required to achieve this are two: (i) locking of layer metadata, and (ii) control messaging through the distributed I/O stack.

The single-node Violin I/O stack allows the creation of multi-layered storage volumes. These are exported to other storage nodes through block-level network protocols, such as iSCSI. The nodes can further build I/O stacks on top of the remote storage volumes, creating higher-level volumes as needed. This organization is depicted in Figure 3. We find that data I/O requests are correctly forwarded by the layers through this distributed hierarchy 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. A remaining issue with layer distribution is related to the consistency of layer metadata: since a layer can be distributed to two or more nodes (i.e. every node runs a separate instance of the layer on the same volume), in order to maintain the layer metadata consistent between nodes, each layer has to synchronize its metadata to the storage volume in every metadata operation. Since this can be very expensive in terms of performance, the system is able to minimize the overhead by supporting metadata locking for layers. Thus, the metadata can be locked and updated asynchronously.

A second issue with distributing the virtualization hierarchy is effectively passing control messages to layers both downstream (from a high layer to a lower one) and upstream (from a lower layer to a high one). Storage network protocols such as iSCSI offer some support for transferring control commands between remote volumes through SCBs (or SCSI command blocks), but they are always towards the downstream direction. We propose more generic and flexible support for control message transfers through the distributed I/O stack. To maintain compatibility with current standards (e.g. iSCSI), we advocate using the common data I/O mechanisms combined with virtualization layers to handle control messages. One approach we are considering for the downstream direction is defining special virtual block addresses in each volume, where the usual I/O operations would be translated to control messages. For the upstream direction we consider other options, such as ticket callbacks from higher to lower layers. The upstream direction, however, is currently considered future work.

3.2. Sharing Virtualization

The second notion of virtualization concerns storage sharing. Sharing is required at two levels, either at the block level where storage volumes are shared, or at the file level, where applications share files.

3.2.1. Shared Block-level Access Many block-level applications, such as databases, web caches or video servers using raw disk access, must share concurrent access to volumes in a consolidated storage system. Thus, we need to support concurrent volume sharing. We find that a single primitive is required for this purpose: locking support of data block address ranges on the volumes. Using such locks, applications can synchronize accesses to shared blocks on the volume.

We have designed and we are currently implementing this feature in Violin, as a virtualization module. The "locking" layer captures lock commands and generates distributed control commands in Violin. These commands are forwarded through the virtualization hierarchy to special "lock server" modules, which serialize and enforce the locks. Lock commands are extensions to the block API but are compatible to the ioctl device interface of most operating systems. Also the SCSI interface supports custom "command blocks" that can transfer such commands.

3.2.2. Shared File Access Violin achieves shared block-level access and layer distribution within the storage cluster. However the most common API of accessing storage is through the file interface, which we can offer with a filesystem on top of a Violin shared block volume. Our goal for the filesystem implementation is that the filesystem itself should be as simple as possible, while independent functionality should be moved at the storage back-end. Our approach to the filesystem design is similar to Frangipani [20], where many independent filesystem instances run on a single shared block volume without direct communication among the instances themselves. The filesystem instances share the blocks of the volume as any other block-level application, using locks on the volume's blocks to maintain consistency for the filesystem metadata. This organization initially simplifies the filesystem design. However, in contrast to previous work, our goal is to simplify filesystem design and implementation by reconsidering the division of functionality between the filesystem and the block-level layer.

We find that the block-level layer can be enhanced with three types of support: (a) block allocation, (b) block-range locking, and (c) metadata consistency. The free block allocation and the lock service greatly reduce the filesystem complexity in comparison to other distributed filesystems. The filesystem code is only aware of a large shared volume and is not aware of other instances running on it. It simply locks the blocks it needs for atomic metadata updates, and uses the free block allocation commands to al-

locate or free data blocks on the shared volume. These extensions allow us to build a simple pass-through filesystem that provides only naming and file-level access rights.

4. Related Work

Storage Tank (ST) is a scalable distributed file and storage management system running on heterogeneous machines over storage area networks [14]. It offers virtualization and centralized policy-based management of storage resources. It separates data storage from metadata management and enables clients to access data directly for improved scalable performance. Although we share several goals with ST, the main difference of our system is its design approach that aims at pushing significant storage system functionality behind the block-based interface. A second difference is the versatility of our system to serve diverse application needs through our extensible design. Another difference is the architecture of our distributed filesystem. Our FS is based on independent instances operating on a shared block volume, similar to Frangipani [20]. On the other hand, the ST filesystem has metadata servers, uses many disk volumes and each FS instance is aware of the others and directly cooperates with them.

Object-based storage advocates that the traditional block-based interface of storage devices should be changed to directly support management of objects [15]. Although blocks offer fast scalable access to shared data, a file server is currently needed to authorize the I/O and maintain the metadata at the cost of limited security and data sharing. Support for data objects essentially moves to the storage devices issues of internal space management, quality-of-service guarantees and access authorization. Although decentralized storage management is one of our goals, we aim at supporting metadata management on top of commodity devices rather than modifying their hardware interface. Additionally, we strive to hide storage management functions behind the block-based interface whenever possible.

Cluster filesystems such as GPFS offer shared access to distributed homogeneous resources [18]. Metadata management is distributed across multiple file server nodes, and shared data can be accessed in parallel from different storage devices. Even though cluster filesystems successfully manage storage space in data centers, their design is monolithic and non-extensible. Furthermore, such systems offer little flexibility in reorganizing data or reconfiguring devices, making administration tasks tedious.

Stackable filesystems facilitate file system development through incremental refinement of features and extensible interfaces [23]. In the present paper, we advocate our approach on using software modularity and extensibility in building sophisticated storage system functionality while preserving the block-based interface. In essence, our approach is complementary to the stackable filesystem design since we target different layers of the data storage hierarchy.

5. Conclusions

In this paper we identify two main challenges for consolidated storage in a data center: (i) the scalable, low-cost platform issue and (ii) the software infrastructure issue. Our goal is to address the latter. We propose a virtualization infrastructure to tackle the storage management, flexibility and reconfiguration issues. We have designed and implemented Violin, a virtualization framework for (i) building and (ii) combining block-level virtualization functions. We have also designed and are currently implementing extensions that will allow distributed virtualization stacks and sharing at both the block and the file level.

REFERENCES

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proc. of the 8th ASPLOS*, pages 81–91, San Jose, California, Oct. 3–7, 1998.

[2] M. de Icaza, I. Molnar, and G. Oxman. The linux raid-1,-4,-5 code. In *LinuxExpo*, Apr. 1997.

[3] EMC. Enginuity(TM): The Storage Platform Operating Environment (White Paper). http://www.emc.com/pdf/techlib/c1033.pdf.

[4] EMC. Introducing RAID 5 on Symmetrix DMX. http://www.emc.com/ products/ systems/ enginuity/ pdf/ H1114_Intro_ raid5_DMX_ldv.pdf.

[5] Enterprise Volume Management System. evms.sourceforge.net.

[6] M. D. Flouris and A. 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, Apr. 11–14 2005.

[7] FreeBSD: GEOM Modular Disk I/O Request Transformation Framework. http://kerneltrap.org/node/view/454.

[8] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proc. of the 8th ASPLOS*, pages 92–103, San Jose, California, Oct. 3–7, 1998.

[9] J. Gray. Storage Bricks Have Arrived. Invited Talk at the 1st USENIX Conf. on File And Storage Techn. (FAST '02), 2002.

[10] HP. OpenView Storage Area Manager. http://h18006. www1.hp.com/ products/ storage/ software/ sam/ index.html.

[11] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proc. of ASPLOS VII*, volume 24:Special, pages 84–93, Oct. 1996.

[12] G. Lehey. The Vinum Volume Manager. In *Proc. of the FREENIX Track (FREENIX-99)*, pages 57–68, Berkeley, CA, June 6–11 1999. USENIX Association.

[13] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *Proc. of the 6th Symposium on Operating Systems Design and Implementation (OSDI-04)*. USENIX, Dec. 6–8 2004.

[14] 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.

[15] M. Mesnier, G. R. Ganger, and E. Reidel. Object-based storage. *IEEE Communications Magazine*, (8):84–90, Aug. 2003.

[16] D. Patterson. The UC Berkeley ISTORE Project: bringing availability, maintainability, and evolutionary growth to storage-based clusters. http://roc.cs.berkeley.edu, January 2000.

[17] B. Phillips. Industry Trends: Have Storage Area Networks Come of Age? *Computer*, 31(7):10–12, July 1998.

[18] F. Schmuck and R. Haskin. GPFS: A Shared-disk File System for Large Computing Centers. In *USENIX Conference on File and Storage Technologies*, pages 231–244, Monterey, CA, Jan. 2002.

[19] D. Teigland and H. Mauelshagen. Volume managers in linux. In *Proc. of USENIX 2001 Technical Conference*, June 2001.

[20] C. A. Thekkath, T. Mann, and E. 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. ACM Press, Oct. 5–8 1997.

[21] Veritas. Storage Foundation(TM). http://www.veritas.com/ Products/ www?c=product&refId=203.

[22] Veritas. Volume Manager(TM). http://www.veritas.com/vmguided.

[23] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the 2000 USENIX Annual Technical Conference*, pages 55–70. USENIX Association, June 18–23 2000.