Virtualize Storage, Not Disks

William Jannen, Chia-Che Tsai, and Donald E. Porter
Stony Brook University
{*wjannen,chitsai,porter*}*@cs.stonybrook.edu*

Abstract

When one uses virtual machines for application compatibility, such as running Windows programs on Linux, the user only wants the API components, yet must emulate a disk drive and execute a second, counterproductive level of media heuristics and I/O scheduling. Systems should have a clean interface between API implementation and media optimization, which would lead to more efficient paravirtualization and facilitate rapid, independent evolution of media optimizations and API features. We describe a design that meets these goals, called Zoochory.

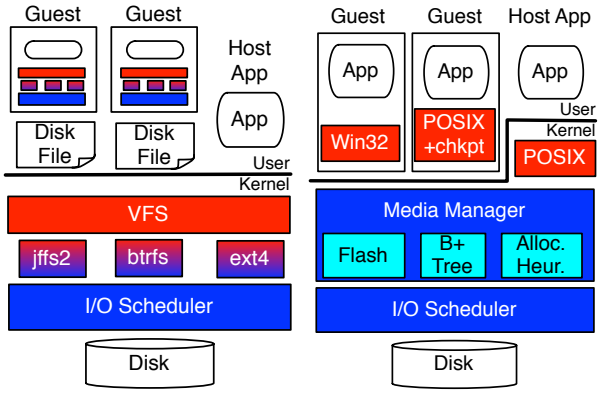
1 Introduction

Modern file systems implement a storage application programming interface (API) and optimize media access. The API implementation is so tightly bound with the media optimizations that virtual machines (VMs) execute fruitless, if not counterproductive, optimizations for hardware that is not actually present. This tight coupling is a design flaw even in the absence of virtualization—virtualization simply exacerbates the problem. This position paper argues for a clean separation of these concerns in the design of the OS storage stack. We propose a storage stack, called Zoochory, that addresses these issues without sacrificing the benefits of a legacy file system.

Using the Linux storage stack as a running example in this paper (Figure 1(a)), the VFS layer (top) partially implements the POSIX API, and the block device layer (bottom) implements simple disk scheduling heuristics, such as the elevator algorithm. Specific file systems in the middle of the stack, such as `ext4`, `jffs2`, and `btrfs`, complete the POSIX API implementation; possibly extend the API with features such as copy-on-write checkpointing, atomic append, or transactions; and optimize media access patterns by judicious selection of on-media data layout, sector allocation heuristics, etc.

As a result of mixing storage concerns, Linux users face an unsavory choice between important hardware optimizations and useful features. For instance, the `btrfs` file system uses B+ trees to optimize rotational disk performance and provide a copy-on-write checkpointing API, but `btrfs` generally performs worse than a flash-optimized file system when benchmarked on flash storage [11]. For users of flash storage, Linux has introduced flash-specific file systems, such as `jffs2` and `f2fs`, that perform and manage wear better than a file system designed for a rotational disk. Flash users must choose between good performance and advanced APIs, such as checkpointing. From an engineering perspective, it is impractical to expect any one file system to implement all combinations of feature sets and media optimizations. As a result, users have to make educated guesses about which file system best suits their applications and hardware. A better storage stack would encapsulate these aspects and allow administrators to compose the right media heuristics and APIs for a given system.

These performance issues are exacerbated when a host OS virtualizes disks. For instance, the Linux anticipatory disk scheduler may hold write requests in (guest) mem-



(a) Linux/KVM storage stack. (b) Zoochory storage stack.

Figure 1: Linux storage stack and proposed Zoochory stack. API components are red, media optimization components are blue. Linux/KVM duplicates functionality and file systems mix both. The proposed storage architecture cleanly separates these, and features of existing file systems are refactored as plug-ins in only one layer. For instance, the API layer in a guest or host application might provide Win32, POSIX, or POSIX with checkpointing (+chkpt), whereas the media manager might use different media plug-ins such as a log-structure for flash, B+ trees for rotational disks, and allocation heuristics.

ory in anticipation of writes to adjacent sectors, which the guest scheduler might then coalesce into one disk request. Yet adjacent sectors in a virtual disk may not be adjacent on the physical disk—needlessly delaying a write and potentially missing better disk optimizations in the host scheduler. Similarly, file read-ahead is a classic rotational disk optimization, and most OSes tune the degree of read-ahead to local disk access. When the host issues read-ahead based on guest read-ahead, the effect can be amplified to the point that useful data is pushed out of memory to make room for overly-ambitious, speculative reads. Having two, independent I/O schedulers can lead to harmful choices based on inaccurate assumptions about the guest application or underlying hardware.

Although virtual disk images are simple for the host OS to implement (e.g., as a file or partition), virtual disks have serious usability issues. Users struggle to resize file systems, access files from outside of the VM, share files between VMs, and efficiently back up important files without dragging along all of the OS boilerplate.

Finally, as an example of how disk images hide more powerful storage abstractions, consider a journaling file system in both a guest and a host. Journaling is a common technique to prevent metadata corruption when writes span multiple sectors, as a hard disk can only write single sectors atomically. When a virtual disk image is stored on a host's journaled file system, the virtual disk could trivially write multiple blocks atomically. Instead, however, the guest will implement a nested journal, which ultimately holds data in guest memory to order virtual disk writes. Guest performance and overall disk scheduling would likely improve by exposing more information to the host kernel, such as ordering and atomicity constraints.

The storage stack should separate API components from media optimizations. Our proposed design, called Zoochory and illustrated in Figure 1(b), facilitates efficient virtualization. Guests (and the host in a type 2 hypervisor [6]) implement the storage API that applications need, such as POSIX or Win32, but only the host need implement the media layer. The API layers express safety requirements to the media layer, such as write ordering, atomic groupings, or, when data is shared, the expected consistency model (e.g., close-to-open consistency or locking). The media layer independently optimizes device accesses based on high-level requirements, information about physical media, and system usage patterns.

In the tradition of paravirtualization [3], we assume that OS developers will make reasonable changes to improve performance when running as a guest. In practice, both Windows and Linux have adopted paravirtualized device drivers. At the storage layer, paravirtualization primarily streamlines virtual disk emulation. This work aims to define a more expressive paravirtualized storage interface, which can improve system performance and modularity.

2 The Cost of Virtual Disks

This section describes a few simple measurements of the overheads incurred by duplicating storage layers, and summarizes relevant measurements from recent papers.

Experimental setup. We performed all experiments on a 6-core AMD FX-6300 processor. To fairly compare host performance, we restricted the host experiments to 1GB of RAM and set the scheduler affinity to 1 core; guests also had 1 GB of RAM and 1 CPU, but the host was allotted 2GB RAM for guest experiments. The host root file system was on its own 2.5TB WD Caviar hard drive, and experiments were run on 60GB `ext4` partitions on a 128GB OCZ Vertex 4 SSD and a 3TB WD Green Intellipower hard drive, respectively. We created a guest root image on the host root file system, and preallocated a 20 GB raw image on each experiment drive. The host experiment partitions were formatted with `ext4` and mounted with `noatime`, and, for the SSD, `discard`. The guest images were formatted and mounted with similar options, and KVM was configured with write-through caching for safety. We hasten to note that the configuration space for nested storage stacks is substantial; our aim is to measure reasonable options a typical administrator would select.

We used four standard microbenchmarks from the filebench test suite [5] to test random and sequential reads and writes. Both guest and host used Ubuntu 12.10 server, with KVM 1.2.

Results. We found that the nesting of file systems incurred significant cost to throughput, especially on flash. For random reads and writes on an SSD, the measured overheads were 41.9% and 87.2%, respectively. For sequential reads and writes, the overheads were 24.4% and 4.7%, respectively.

Rotational disk overheads were generally lower, as the costs of virtualization were partially masked by high mechanical latency. Sequential read throughput was 1.8% lower than the host, and random read throughput was 5.6% lower. Sequential writes were 9.9% lower and random write throughput was 32.3% lower.

Nested file systems. Le et al. measure the cost of nesting file systems on Linux and KVM, compared to the guest accessing a drive directly [12]. For instance, I/O latency is increased by at least 5–15% in the best cases, and latency commonly increases 25%–2$\times$ for their workloads. The main performance benefit observed was in sequential, read-only microbenchmarks, where the guest's read ahead is amplified by read-ahead in the host; in practice, this optimization can harm fairness to other guests, or even push more useful data out of RAM.

Nested schedulers. Boutcher and Chandra also demonstrate that the worst combination of guest and host I/O

schedulers reduces I/O throughput to 40% of the least disruptive combination [4]. They argue for I/O scheduling in the guest, as the guest has more visibility into application patterns; implicit in this argument is the idea that the guest can make educated guesses about the underlying media. Our position is somewhat different: the host should do I/O scheduling, as the host knows media details and understands other guests; but the guest should expose more performance hints to the host.

Thus, duplicate storage layers can substantially harm performance, as well as create a number of configuration pitfalls for the system administrator.

3 Splitting the Storage Stack

This section describes how to separate the storage APIs from media optimizations, beginning with functional requirements of the API layer. The section then describes our prototype design, called Zoochory, which meets these requirements.

3.1 API Layer Requirements

We begin by enumerating the features the API layer requires from the storage layer in any system, with an eye toward generality and hardware-independence.

- **Data storage and naming**, sometimes called a block store—a means to write, identify, and read file data. Most local file systems identify data by physical location. To encapsulate device details, stored data needs a **unique, device-independent identifier**.
  The storage layer interface should specify whether data is stored in fixed- or variable-sized blocks; we expect that both memory and device management will be simpler if the storage layer uses a fixed, common block size, such as 4KB.
- **Extensible metadata storage**. File systems track both common and custom file metadata, such as the last access time or security attributes. Thus, the storage layer should store variable-sized metadata for a file, but its interpretation should be encapsulated in the API layer.
  Explicitly separating data from metadata is an important performance hint and safety constraint. Previous papers have observed that when a host file system does metadata-only journaling, metadata writes in the guest are effectively treated as data writes and may not be written consistently to stable storage, ultimately leading to guest file system corruption if the host crashes [12].
- **Indexing**, or a way to find files. An index can be a hierarchical file system tree, a key/value store, or an index based on file attributes, such as the artist or genre tag recorded in a music file.
- **Write ordering and atomicity**, to ensure that invariants across multiple metadata are upheld on the media.
- In the case of shared storage, a mechanism to **detect concurrent updates** may be useful, although many file systems may prefer to coordinate at the API layer.
- The guest may wish to share additional **performance hints** with the host, such as read-only data, expected access patterns, or data with a short lifespan.

We posit that any media interface meeting these requirements is sufficient to implement any file system API, such as the file system calls in POSIX or Win32. Demonstrating this claim is the goal of ongoing work.

3.2 Zoochory Abstractions

Zoochory is a prototype storage stack that meets the above requirements. Other media abstractions could also meet these requirements; this is an important design space for the community to explore. We think the Zoochory design has some particularly appealing qualities for VMs, such as efficient migration. We are currently developing Zoochory within a Linux host and guest. Key Zoochory abstractions are described below.

Zoochory uses content addressable storage (CAS) for **data storage and naming**. In CAS, data is addressed by a collision-resistant fingerprint function over its contents, such as a `SHA-1` hash, rather than by a fixed offset on a device. CAS is a hardware-independent layer of indirection from data fingerprint to physical location. Our implementation stores data in 4 KB blocks.

We note that CAS has been widely used in distributed and networked file systems, for deduplication [23, 27], minimizing data transfer [19], and ensuring integrity [13]. For many of the same reasons, we are interested in CAS as an alternative block interface for local data storage.

In addition to a CAS data store, Zoochory provides file recipes and namespaces, upon which the API layer can implement metadata and other high-level abstractions. Analogous to a Unix inode, a file recipe stores a list of the block hashes that compose a file, along with a **variable-sized metadata** section. The storage layer does not interpret the metadata section. A recipe's list of hashes is sufficient to reconstruct the file from its constituent blocks.

Flat namespaces are a building block for the API layer to implement many common file system abstractions, including directory trees and checkpointing. A namespace is a key/value store used to **index** files. Keys may be arbitrary byte sequences, such as a hierarchical pathname, or an arbitrary attribute, such as keywords in a text file. Values, however, must be the `SHA-1` hash of a file recipe. Our namespace design is partially inspired by Ventana's file system `views`, in which a VM's file hierarchy is composed of one or more independent file trees [21]. In our design, a guest may similarly compose several names-

paces in order to facilitate sharing, checkpointing, or other advanced API features. Access control is enforced at the granularity of a namespace.

To enforce **atomic updates to disk** and to implement various **consistency models** when a namespace is shared, the namespace API includes minitransactions [1]. Minitransactions are essentially an arbitrarily large, atomic compare-and-swap operation on a set of keys. When a minitransaction commits, all writes are guaranteed to be on disk (atomic and durable). For instance, a user may use a minitransaction to ensure that a set of updates are written to disk atomically for a private namespace. On the other hand, close-to-open consistency could be implemented for a shared namespace by retrieving the hash of a file's recipe when it is opened, and inserting a modified hash upon file close. Stronger consistency models or a transactional file system can detect concurrent modification by asserting a key/hash pair.

Example. A guest could implement a POSIX-compliant file system on Zoochory by mapping complete paths to file recipes in the namespace. The Zoochory recipe includes a variable-sized metadata block, which could store fields typically placed in an inode, as well as an ordered list of block hashes. Listing a directory, such as `/tmp`, would require listing all namespace keys prefixed by `/tmp` but without a subsequent `/`. The metadata block could also store information such as a symbolic link target. In order to implement an atomic update to metadata, such as `rename`, the guest may use a minitransaction to atomically change a the name to recipe hash mappings.

This design works similarly to a standard Unix file system, except that the guest is not aware of any data placement, and expresses only high-level consistency constraints. The host is free to place file data on any device, including a network storage system. Moreover, the use of content addressing facilitates deduplication and detection of corrupt data.

Migration. Zoochory facilitates VM migration among hosts, as it cleanly divorces *where* data is stored from the API layers. The relatively small indices and recipes provide enough information to recreate a file system view on a new host. Hash based indexing also minimizes unnecessary data transfers [19].

Cooperative media management. In some cases, such as a database, it may make sense for a guest to manage its own storage cooperatively with the host. For instance, a guest may wish to use a media optimization plug-in the host does not trust. For most applications, we expect performance hints will be sufficient.

For the near future, the host will issue the final device I/O schedule, similar to the Exokernel file system [10]. Safe, direct access to storage hardware by a guest is a challenging research problem, as existing disk controllers cannot limit a guest to a subset of a storage device, and a disk arm must be carefully scheduled to ensure fairness.

4 Independent Evolution of Layers

A key reason to separate the storage stack design is to permit the API and media layers to evolve independently and rapidly. In other words, system developers should be free to identify useful API extensions, such as atomic append, and independently experiment with different heuristics as new classes of media evolve. Because existing systems entangle these facets, both evolve more slowly.

4.1 API Evolution

Entangling media and API features in the same code has slowed API evolution. As a result, applications have resorted to implementing custom file systems within a single file [7], which introduces substantial noise into system performance analysis and frustrates performance optimization. Modern file systems now face a choice: they must introspect to understand increasingly strange file access patterns, or they must provide a storage interface that better captures applications' data relationships and safety constraints. We believe that a more expressive interface will yield a cleaner design and unlock more performance optimizations in the long run.

4.2 Media Evolution

In this subsection, we consider a few common classes of media optimizations in modern file systems and argue that Zoochory can implement similar optimizations.

Spatial locality. A classic file system heuristic allocates blocks for the same file from adjacent disk sectors. Similarly, a file system may try to co-locate files of the same directory, or files that were created at the same time. These heuristics attempt to optimize future rotational disk accesses based on write patterns.

The Zoochory media interface conveys sufficient information to make similar locality inferences. A recipe identifies data in the same file, similarity of keys in a namespace suggests likelihood of joint access, and the media layer can observe temporal locality of write requests. Although CAS naturally deduplicates stored data, nothing prevents the media layer from storing multiple instances of the same block to optimize future reads (an observation applied in deduplication systems [25]). Alternatively, a Zoochory system running on flash media can eliminate these optimizations and favor deduplication, since it is not subject to expensive seek times.

Moreover, delaying disk block allocation can *improve* spatial locality on disk, and therefore performance. As

a file grows in the current Linux VFS design, blocks must be allocated on disk immediately. In contrast, TxOS deferred block allocation until transaction commit, and then allocated all blocks at once, yielding substantial performance improvement for certain write-intensive workloads [22]. The Zoochory design gives the storage layer similar flexibility to buffer and group pending writes, improving spatial locality on the physical media.

Finally, we observe that many spatial locality heuristics are not resilient to sector remapping and other techniques for masking failures in the storage media. In other words, these heuristics incorrectly assume that sectors with adjacent numbers are also close on the media. For instance, if sector 10 is failing on the drive, it may be transparently remapped to sector 6000, frustrating efforts to ensure spatial locality on a rotational disk. Any file system could solve this by adding another layer that moves data from a remapped sector back to a closer sector (say 15) and updates the indexing metadata. Because every file system will require this facility for sustained performance, and because detecting and correcting for sector remapping can be tricky, it makes more sense to encapsulate and reuse this code.

Optimized indexing data structures. Several recent file systems optimize data look-up with specialized data structures. For instance, `ext4` transitioned from storing directories as lists to trees [16], and B+ trees are a cornerstone of the `btrfs` design [17]. Similarly, `ext4` also optimized the internal representation of large, contiguous disk block allocations by switching from a block list to an extent list. These data structure shifts follow changes in expected common-case storage patterns—e.g., expected average directory size.

Our Zoochory media design can similarly incorporate optimized data structures, but, unlike a typical file system stack, these optimizations are completely encapsulated in the media layer. This separation facilitates the dynamic adjustment of on-media structures to actual workload characteristics, such as switching from a list to a tree as an index grows. Dynamic tuning is difficult when assumptions are baked into the file system code.

Write grouping for flash. File systems such as `jffs2` optimize flash performance by adopting a log-structured design. Arguably, the simplest implementation of Zoochory is as a log-structured file system. Speaking more broadly, nothing in our API mandates a given on-disk format. For instance, one may write logs of the namespaces in whatever units make the most sense for the media.

We recognize that a well-specified media layout is critical, especially for removable media. In our design, any media plug-in should specify its media layout. Our aim is to decouple API features from media layouts, facilitating more rapid system evolution.

5 Related Work

Deduplicating Storage Layers. Flash media commonly includes a flash translation layer (FTL) for backwards compatibility, which internally allocates blocks; the file system in turn allocates blocks exposed by the FTL for its files. The storage community has explored ways to balance this duplication of roles with the benefits of a layered design, as each layer has limited visibility into the other, and little means to express intent. Current approaches include co-designing a file system with the FTL [15], delegating block allocation to the FTL [9, 29], and adapting file system layout policies to inferred FTL behavior [8]. This paper argues for an approach that encapsulates these media-specific heuristics from API components, while exposing sufficient information to make good optimizations in the media layer.

Optimizing Virtual Disk Image Access. Many systems have been designed to optimize access to virtual disk images [24, 26] and storage of virtual disk images [2,18,20,28]. Especially when disk images are stored on a remote server, these designs may employ CAS as a building block to minimize data transfer and transparently deduplicate image boilerplate [14] . Efficient virtual disk access is an important, practical problem for legacy file systems; our work aims to identify a more efficient, paravirtual alternative to virtual disks.

Virtual Disk Alternatives. VAFS proposes an alternative paravirtualized interface for storage, but with a focus on features for efficient sharing, versioning, and external VM management [21]. For instance, VAFS facilitates efficiently patching vulnerabilities in system code across many VMs. VAFS, however, relies on centralized servers to manage the state of each object, whereas namespaces provide the means for individual users or guest OSes to manage and customize their own file system state.

6 Conclusion

This paper proposes a local storage stack which cleanly separates the concerns of an API implementation from media optimizations. Zoochory implementation is ongoing. We expect that this design retains the benefits of current storage systems, but also creates new opportunities.

7 Acknowledgements

We thank Mike Ferdman and the anonymous reviewers for insightful comments on earlier versions of this paper. This research was supported in part by NSF CAREER grant CNS-1149229, NSF CNS-1161541, and the Office of the Vice President for Research at Stony Brook University.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 159–174, 2007. §3.2

[2] G. Ammons, V. Bala, T. Mummert, D. Reimer, and X. Zhang. Virtual machine images as structured data: the mirage image library. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, pages 22–22, 2011. §5

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003. §1

[4] D. Boutcher and A. Chandra. Does virtualization make disk scheduling passé? *ACM Operating Systems Review*, 44(1):20–24, Mar. 2010. §2

[5] Filebench. `http://sourceforge.net/projects/filebench/`. §2

[6] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, 1973. §1

[7] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 71–83, 2011. §4.1

[8] J.-Y. Hwang. F2FS (flash-friendly file system). Presented at the Embedded Linux Conference, 2013. §5

[9] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. Dfs: A file system for virtualized flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010. §5

[10] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 52–65, 1997. §3.2

[11] M. Larabel. F2fs file-system shows hope, runs against btrfs & ext4. `http://www.phoronix.com/scan.php?page=article&item=linux_f2fs_benchmarks&num=1`, 2013. §1

[12] D. Le, H. Huang, and H. Wang. Understanding performance implications of nested file systems in a virtualized environment. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2012. §2, §3.1

[13] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, 2004. §3.2

[14] A. Liguori and E. V. Hensbergen. Experiences with content addressable storage and virtual disks. In *Proceedings of the Workshop on I/O Virtualization*, pages 5–5, 2008. §5

[15] Y. Lu, J. Shu, and W. Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2013. §5

[16] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*, 2007. §4.2

[17] A. McPherson. A conversation with Chris Mason on btrfs: the next generation file system for Linux. `http://www.linuxfoundation.org/news-media/blogs/browse/2009/06/conversation-chris-mason-btrfs-next-generation-file-system-linux`. §4.2

[18] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: Virtual disks for virtual machines. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 41–54, 2008. §5

[19] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 174–187, 2001. §3.2, §3.2

[20] F. Oliveira, G. Guardiola, J. Patel, and E. Hensbergen. Blutopia: Stackable storage for cluster management. In *IEEE International Conference on Cluster Computing*, pages 293–302, 2007. §5

[21] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 353–366, 2006. §3.2, §5

[22] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 161–176, 2009. §4.2

[23] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, 2002. §3.2

[24] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield. Capo: Recapitulating storage for virtual desktops. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 3–3, 2011. §5

[25] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: latency-aware, inline data deduplication for primary storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 299–312, 2012. §4.2

[26] K. Suzaki, T. Yagi, K. Iijima, N. A. Quynh, and Y. Wantanabe. Effect of readahead and file system block reallocation for LBCAS. In *Linux Symposium*, pages 275–286, 2009. §5

[27] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 225–238, 2010. §3.2

[28] VMware. VMware vStorage VMFS product datasheet. `http://www.vmware.com/products/datacenter-virtualization/vsphere/vmfs.html`. §5

[29] Y. Zhang, L. P. Arulraj, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2012. §5