vDrive: An Efficient and Consistent Virtual I/O System

Mohit Saxena[†], Pin Zhou[*] and David A. Pease[†]
[†]*IBM Research Almaden,* [*]*Datos IO*

Abstract

The most popular methods for managing storage and providing crash consistency are I/O virtualization and journaled filesystems respectively. This popularity is due to their widespread use in production environments. However, both of these methods have evolved separately in different contexts in the past.

This paper presents a first look on providing crash consistency for virtual I/O caches through journaled filesystems. We find that nested filesystem journaling in guest and host operating systems has a significant performance cost. This cost is attributed to the use of traditional disk interfaces for cache flushes and lack of coordination between the two journaling levels. We present *vDrive*, a consistent virtual I/O system architecture, with a new virtual disk interface and semantic journaling mechanism designed to provide high performance.

We have implemented vDrive interface extensions in the KVM/QEMU virtual I/O system and semantic journaling in the Linux ext4 filesystem. We show through experiments that vDrive outperforms nested journaling by up to 142% and correctly recovers to a consistent state after a crash.

1 Introduction

In recent years, virtualization has improved hardware utilization allowing service providers to offer a wide range of application and infrastructure services [3, 16, 24]. I/O virtualization enables efficient and flexible allocation of storage resources across different virtualized workloads.

To achieve high bare-metal performance, I/O intensive workloads require direct access to physical disks or logical volumes. For this purpose, modern hypervisors implement a technique called PCI Passthrough or DirectPath I/O [40]. Passthrough I/O achieves high performance by bypassing the host software on the I/O path. However, it gives up a lot of virtualization flexibility including complicated VM live migration and no storage space overcommit.

As a result, the most popular I/O virtualization technique today is paravirtual I/O[1] [29, 38]. A virtual I/O system consists of a modified driver in the guest operating system, and a virtual disk exported to the guest as a block device but stored as a file on the host filesystem. This enables flexible allocation of storage space and additional management features embedded in VM images [28]. Previous work has pointed to the performance implications of virtual I/O resulting from nested file systems [21], and addressed the overheads for frequent switches between guest and host operating systems [14, 15].

In this paper, we present a first look on the performance implications of crash consistency for virtual I/O. The most popular and widely adopted technique to provide crash consistency in virtualized environments is the use of journaled filesystems in guest and host operating systems. However, the two journal levels in *nested filesystem journaling* interact in ways, which have not been studied earlier, and result in significant performance cost. We find that this performance cost is attributed to two major reasons embedded in the original design of filesystem journaling. First, journaling has been designed to provide crash resilience on bare-metal hardware for data resident in a physical disk cache. Second, journaling uses the traditional storage interfaces for flushing data to disk storage. In contrast, virtual I/O introduces a more complex hierarchy of cache levels, but at the same time provides greater flexibility to rethink the software interface to virtualized storage.

The first source of overhead for nested filesystem journaling arises because the two journaling protocols provide crash resilience for the *same* application data write *independently* across different cache levels in guest and host without any coordination. The guest filesystem

[1] we refer paravirtual I/O as virtual I/O in the rest of the paper

journal provides crash resilience against the virtual disk cache, while the host filesystem journal protects against the physical disk cache. As a result, each write from the guest application waits longer because of multiple stalls and additional writes introduced during journal commits at each level. This significantly degrades the combined system performance for applications running atop nested journal filesystems. We investigate the design of a nested journaling protocol, which is aware of the virtual I/O semantics to minimize these overheads.

The second source of overhead arises due to the use of the traditional storage interface of expensive disk *cache-flushes* for nested filesystem journaling. The disk cache-flush operations used in modern drives impose both ordering and durability guarantees for writes [36, 27]. Recent work on optimistic crash consistency [5] decouples the costs of ordering and durability for disk cache-flushes on a single level of filesystem journal. We investigate the first application of optimistic crash consistency to virtual I/O for reducing the commit cost of nested filesystem journaling. We find that it requires revisiting the software cache-flush interface to virtual disk and major changes to the virtual I/O system. However, these changes simplify the integration of optimistic crash consistency with the guest filesystem than what is required for a single journal level [5]. We exploit this opportunity to reduce the overheads of nested journaling.

We present *vDrive*: an efficient and consistent virtual I/O system. vDrive solves the above problems by using a new technique to provide low overhead crash consistency: *semantic journaling* for virtual I/O. vDrive extends the virtual disk interface by introducing two new primitives: *vorder* and *vflush*, which are exported to the virtual I/O drivers in the guest operating system. The vorder operation only guarantees ordering of preceeding writes, while vflush enforces both order and durability. The guest drivers use these primitives selectively based on the semantics of the data being persisted from the virtual cache hierarchy to the physical disk. With both primitives, vDrive always enforces the order of committed writes so as to enable a consistent state recovered after a crash.

The main contributions of this paper are as follows:

- We present the first experimental analysis of the consistency and performance tradeoffs for virtual I/O. We conduct our experiments across different storage technologies. We identify a key problem with virtualized disk storage: overhead of nested filesystem journaling for different virtual I/O caching modes.
- We design, implement and evaluate vDrive, an efficient and consistent virtual I/O system, which exploits a new virtual disk interface to implement semantic journaling for virtual I/O traffic. vDrive improves virtual I/O performance by up to 142% over nested journaling and correctly recovers to a consistent state after crash.

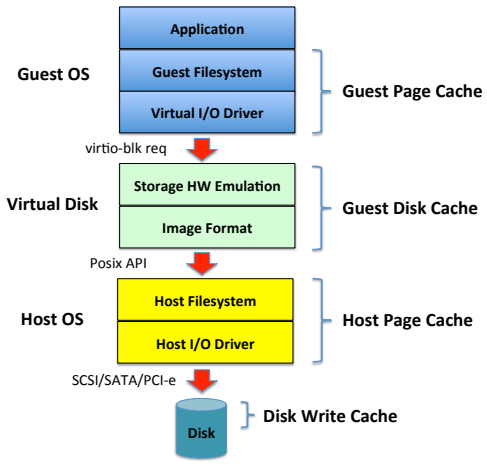

Figure 1: **Virtual I/O Storage Stack:** *The figure shows the different software and caching layers in the virtual I/O storage stack.*

The remainder of the paper is structured as follows. Section 2 and 3 motivate vDrive by describing a qualitative and quantitative analysis of virtual I/O performance tradeoffs respectively. Section 4 and 5 present the detailed description of vDrive design and implementation. Finally, we evaluate vDrive design techniques in Section 6, and finish with related work and conclusions.

2 Motivation

In this section, we describe how the standard virtual I/O model achieves crash consistency through nested filesystem journaling and different caching modes within guest and host operating systems. We demonstrate how these techniques and the use of a traditional disk interface (*i.e.* cache-flush commands) result in negative performance impact for virtual I/O.

Virtual I/O Storage Stack. Figure 1 shows the virtual I/O stack comprised of different software layers and cache levels in the guest and host operating systems. An application I/O request in the guest can be served from the guest OS page cache, otherwise it is forwarded through the frontend guest virtual I/O device driver to the backend virtual disk running in the host user-space. The virtual disk is typically a file on the host fileystem whose layout can vary based on the feature or performance requirements [28].

There are two sets of interfaces for a virtual disk: virtio-blk interface [29] with the guest driver and the posix filesystem interface with the host OS. Apart from

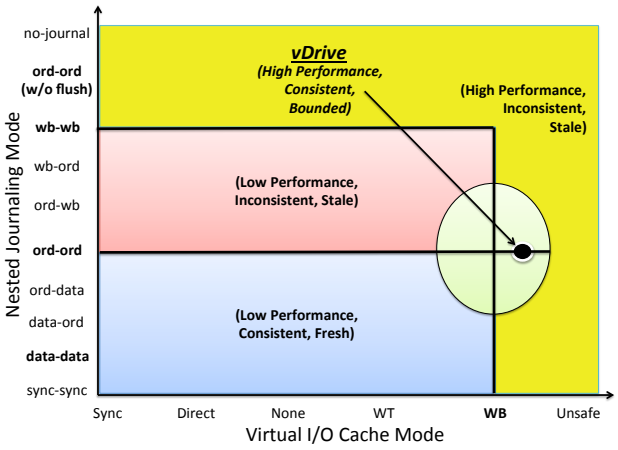

Figure 2: **Virtual I/O Tradeoffs:** *The figure shows the impact of different nested journaling and virtual caching modes on the performance, consistency and freshness of data for virtual I/O. We show the most relevant combinations of filesystem journaling modes in guest and host OS: data, ordered, writeback and ordered without cache-flush. We show the popular cache modes for guest disk cache, host page cache and disk write cache: direct, none, write-through, write-back and unsafe.*

the read and write requests; the guest driver can send *cache-flush* commands to the virtual disk. The virtual disk further translates them into the host filesystem *fsync* system calls. Finally, the host filesystem sends cache-flush commands to flush data from the physical disk write cache.

A guest I/O request can get cached within the host at three levels: virtual disk cache, host page cache or the physical disk cache. Each guest virtual machine can be configured from the host to use one of the five different combinations for host cache modes: *write-back* (all three caches are enabled), *write-through* (guest disk cache is disabled), *none* (host page cache is disabled), *direct* (both guest disk cache and host page cache are disabled) and *unsafe* (all caches are enabled and any cache-flush commands from guest are ignored).

The guest and host filesystems can use journaling to provide write ordering and durability across the virtual and physical disk write caches. There are three major modes for journaling: *data* (both metadata and data are committed into the journal before being written into the main filesystem), *ordered* (data is written to the main filesystem before metadata is committed into the journal), and *write-back* (no ordering is preserved, data may be written into the main filesystem after metadata has been committed into the journal). The filesystem journals send disk cache-flush commands to ensure ordering and durability of writes.

Virtual I/O Tradeoffs. The different virtual caching and nested filesystem journaling modes provide different tradeoffs between performance, consistency and data freshness. Figure 2 shows a classification of such tradeoffs among the four most relevant combinations of cache and journaling modes.

The first lowermost region in blue corresponds to the combination of ordered or data journaling modes in both guest and host, and any of the four cache modes which pass guest cache-flush commands for journal commits through all cache levels. After recovery, this combination provides a consistent and the most recent or fresh state, which existed before a crash. However, these guarantees comes at the performance cost to provide write ordering and immediate durability through expensive disk cache-flushes. The use of synchronous I/O mode in guest and host (shown as the origin of the two axes in Figure 2) provides immediate durability for all writes with a huge performance loss.

The second region in red corresponds to the combination of write-back or ordered journaling modes in both guest and host. The use of write-back journaling mode in either guest or host could result in unordered writes to the physical disk, and an inconsistent and stale virtual disk image after a crash. The use of ordered journaling mode, especially in the guest, results in frequent cache-flush requests. As a result, these journaling mode combinations provide lower performance with the three caching modes: direct, none and write-through; all of which bypass some cache levels at host for unordered writes from journal commits. Apart from the consistency cost, these three cache modes provide different levels of cache exclusiveness and differ in performance due to different cache hit rates (we elaborate this further in Section 3).

The third region in dark green corresponds to no-journaling or combination of ordered journaling modes in guest and host without cache flushes enabled for journal commits. This is equivalent to the unsafe caching mode, which ignores all cache-flush requests from guest regardless of journaling mode. These combinations sacrifice both consistency and freshness of data in the virtual disk for a guest to achieve high virtual I/O performance.

Why vDrive? The fourth circular region in light green corresponds to the space around the combination of ordered journaling modes in both guest and host, and write-back virtual caching mode. This is the default combination in KVM/QEMU, which provides consistent and fresh states but at a high performance cost of cache-flushes in nested ordered journaling mode. We focus within this region because all three spaces described earlier converge here. As a result, moving in different directions within this region result in different tradeoffs for performance, consistency and freshness. Therefore, we aim to find an approach to improve the performance of

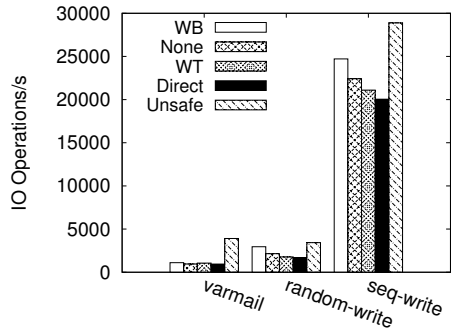

Figure 3: **Caching modes for virtual I/O:** *The figure shows the impact of different caching modes for virtual disk storage on varmail, random write and seq write workloads.*

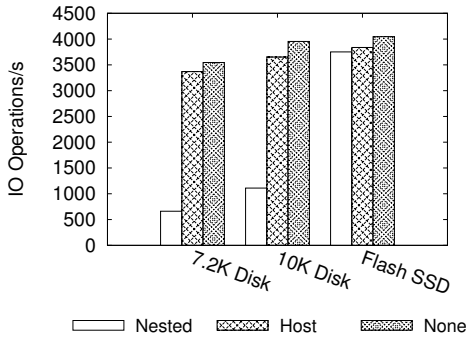

Figure 4: **Filesystem journaling for virtual I/O:** *The figure shows the impact of filesystem journaling on different storage devices (7.2K RPM SATA, 10K RPM SAS Disks and PCI-e Flash SSD). Cache flushes on commit are performed in both guest and host OS (Nested), host OS only (Host), and disabled in both OS (None).*

nested ordered journaling mode with the ability to recover to a consistent state. To achieve this, we use optimistic crash consistency for virtual I/O by decoupling the guarantees of write ordering and durability for nested journal commits. The *vDrive* design provides *write ordering* but *eventual durability* to achieve higher runtime performance, and recover to a consistent virtual disk image, which existed within a bounded time interval before a crash.

3 Quantifying Virtual I/O Tradeoffs

We now quantitatively show the performance impact of different virtual caching and nested filesystem journaling modes. We use the Linux ext4 journaled filesystem in KVM guest and host for our experiments.

Virtual I/O Caches. Figure 3 shows the performance impact of the different virtual caching modes for the default ext4 ordered journaling mode with cache-flushes enabled (at filesystem mount-time) in both guest and host OS. We use the filebench suite [1] of write-intensive workloads (random and sequential writes to isolate the impact of cache hit rates, and varmail: fsync intensive to isolate the cost of crash consistency).

The varmail benchmark simulates a mail server and includes many small synchronous updates, thereby exercising both the journaling and caching mechanisms through frequent cache-flush requests. As a result, all cache modes perform similar and upto 3.5x slower than the unsafe mode. None of these four traditional caching modes can provide better performance for this workload with their existing consistency semantics.

For random write workload, the write-back cache mode is better than other caching modes and comparable to unsafe mode. The is because write-back cache is inclusive of all cache levels and fewer cache-flush commands from the workload result in larger effective cache size. The none mode performs about 20% better than write-through for both random and sequential write modes. Disabling the host page cache in none mode turns all write operations into direct I/O operations to disk write cache, which perform better than synchronous I/O in write-through mode. Direct mode performs worst because it converts all writes into direct synchronous operations each of which result in a disk cache-flush.

Overall, we find that all the virtual caching modes including the default write-back mode always perform slower than unsafe caching mode. The peformance difference is especially significant when the application and the filesystem journal within the guest OS issue frequent cache-flush commands to achieve a consistent virtual disk image.

Nested Filesystem Journaling. We now isolate the impact of nested filesystem journaling in guest and host OS on virtual I/O performance. Figure 4 shows the performance of three different ext4 ordered journal mode configurations: Nested (both guest and host filesystems use ordered journal mode with cache-flushes enabled), Host (unlike Nested mode, cache-flushes are disabled for guest filesystems) and None (cache-flushes are disabled for both guest and host filesystems). We conduct experiments on three different storage devices: a nearline 7.2K RPM SATA disk, an enterprise 10K RPM SAS disk, and a high-end PCI-e Flash SSD. For brevity, we only show the results for the varmail workload and write-back cache mode.

Nested filesystem journaling has significant performance impact on disk storage. For a nearline disk, disabling cache-flushes for guest filesystem journal commits results in nearly a factor of five performance improvement. The enterprise disk has relatively lower seek and data transfer times from disk write cache to disk be-

cause of a faster spindle. Nonetheless, the performance loss is still more than 3x for nested filesystem journaling. Disabling host filesystem journal cache-flushes result in only about 8% performance improvement over no flushes in guest (*Host* configuration in Figure 4). Overall, nested filesystem journaling with cache-flushes enabled on both guest and host has significant negative performance impact on virtual I/O performance for disk storage.

Figure 4 also shows the performance impact of nested journaling on a high-end PCI-e Fusion-io SSD [11]. We also found similar results on an Intel SATA SSD [17]. As expected, flash SSDs have faster random access latencies than disks. Therefore, cache-flushes resulting from nested filesystem journal commits in guest and host operating systems have less than 10% performance impact on both SSDs. One approach to boost nested journaling performance on disk storage is the use of an external journal placed on a flash SSD. However, an external journal on a different device cannot be synchronously updated with filesystem updates, and the result is an inconsistent or read-only filesystem [8].

These findings indicate that providing low overhead crash consistency with the use of traditional cache-flush interface for virtual caches and nested filesystem journals is challenging on virtualized disk storage. To overcome these challenges, we design *vDrive* with the following three goals:

- *Consistent interface* to provide a consistent virtual disk image across a system crash or power failure.
- *Cache management* to improve performance for the virtual cache hierarchy.
- *Filesystem journaling* to reduce the cost of cache-flushes for nested filesystems.

4 System Design

vDrive is a new virtual I/O system designed to improve performance for virtual I/O and provide a consistent virtual disk image after a crash or power failure. This section describes the design of the virtual disk interface and consistency guarantees of vDrive, nested journaling protocol, and cache management. Figure 5 shows the flow of write and cache-flush requests from the guest to vDrive and from vDrive to the host filesystem.

4.1 vDrive Interfaces

The vDrive design provides a consistent disk interface that reflects the need of a virtual disk to: (i) provide in-order guest journal commits to achieve a consistent virtual disk image across a crash, and (ii) provide immediate or eventual durability to trade between performance and freshness of a virtual disk.

The vDrive interfaces, as shown in Table 1, are a small extension to the standard virtio interface [29] between guest OS virtual driver and host user-space QEMU virtual disk driver, and the posix interface between the virtual disk cache manager and the host filesystem.

Name	Function
vflush	flush I/O operations from the virtual disk write cache.
vorder	order I/O operations queued ahead of *vorder*.
aio-notify	notify that all I/O operations queued before *vorder* have been synchronized to physical disk.
preadv/pwritev	gather/scatter a vector of blocks from virtual disk file at a given offset to host memory buffers.

Table 1: **The vDrive Interfaces**

4.1.1 Guest Interfaces

The standard virtio interface between the guest OS and virtual disk is based on a virtio ring buffer implementation. The guest driver enqueues read/write/flush requests into the ring buffer and kicks-off the buffer to the host. The guest disk cache manager implemented within the QEMU emulation block driver dequeues the requests from the ring buffer and then translates them in a separate I/O thread to the underlying system calls to the virtual disk file on the host filesystem.

vDrive provides two new synchronization primitives to decouple ordering and durability guarantees of guest flushes to the virtual disk.

Write Flush. The *vflush* primitive is similar to the traditional flush interface used for ATA/SAS disks, also termed as "cache-flush" or "synchronize cache", which flushes all buffered writes from the disk write cache. This primitive provides both immediate durability and correct write ordering. It only returns when the buffered writes have been acknowledged to be flushed from all three host cache levels including the guest disk write cache, host page cache and the physical disk write cache. As a result, the cost of the *vflush* primitive is larger for a virtual disk than the traditional cache-flush for physical disk write caches.

Write Order. The *vorder* primitive provides ordering for all writes buffered within the three host cache levels. When the operation returns, this request has been only submitted to the I/O queue in the host OS. Immediate durability of the preceeding writes is not guaranteed upon its return. However, all preceeding writes complete in the order as they were submitted by the guest driver to the host emulation framework. As a result, new writes issued after *vorder* will always be durable after the writes preceeding *vorder*. The cost of using this primitive is al-

ways less than that of *vflush* as it does not acknowledge for durability.

By default, all *cache-flush* requests from the guest filesystem are converted to *vorder* requests from the guest virtual I/O driver. The application and guest operating system also has an option of using the *vflush* primitive to force flush through all cache levels in the host. We describe the selection mechanisms and policies for the use of these primitives in more detail in our discussion on semantic journaling in Section 4.2.

4.1.2 Host Interfaces

The vDrive cache manager is implemented within the QEMU user-space virtual disk driver and controls when blocks are flushed from the three host caches: guest disk cache, host page cache and physical disk cache. It extends the existing interface used between the traditional QEMU user-space virtual disk driver and the host filesystem.

Flush Notification. The vDrive block driver can issue new writes after *vorder*, which reach disk after all preceeding writes issued before *vorder*. However, the *vorder* primitive does not wait to acknowledge the durability of preceeding writes. The vDrive cache manager instead uses the *aio-notify* interface to asynchronously acknowledge the completion of all preceeding writes. The *aio-notify* is a new interface implemented within the vDrive cache manager as a signal handler. Specifically, this primitive is used to receive a notification when all writes buffered within the host caches prior to a *vorder* operation have been flushed to the disk. The vDrive cache manager also updates additional information such as the number of pending write operations and resets a timer within the *aio-notify* call. As we will discuss later in Section 4.2, this information is useful to bound the freshness of the virtual disk image recovered after a crash.

Scatter/Gather I/O. The vDrive cache manager uses the existing *scatter-gather* vectored interface implemented within the QEMU block driver to issue read-/write system calls to the host filesystem. In addition, it updates the number of write operations pending to be acknowledged through *aio-notify* or *vflush* on each *pwritev* operation.

4.2 Semantic Journaling

The new vDrive interfaces enable the design of semantic journaling to reduce the cost of crash consistency for I/O virtualization. To understand semantic journaling better, we first examine the interaction between the guest and host filesystems using Linux ext3/ext4 ordered journaling mode protocols. Much of our discussion is also applicable to other journaling filesystems such as IBM JFS, SGI XFS and Windows NTFS.

Journaling Protocol. Figure 5 shows the flow of application writes to a data block (*D*) through the guest filesystem, virtual I/O driver, guest cache manager within the host user-space QEMU block driver and the host filesystem. When a guest application updates the filesystem state, either the filesystem metadata (*M*), user data (*D*), or ofen both need to be updated in an ordered manner. The atomic update of the filesystem metadata including the inode and allocation bitmap to the journal is referred to as a transaction. The filesystem must first write data blocks (*D*) and log the metadata updates (J_M) to the journal (filesystem write W_1), then finally write a commit block (J_C) to the journal to mark transaction commit (filesystem write W_2). Finally, the metadata (*M*) can be written in place to reflect the change (filesystem write W_3). Therefore, the journaling protocol is: *D* and J_M before J_C before *M*, or more simply: $D \mid J_M \rightarrow J_C \rightarrow M$. The data (*D*) and the journal metadata entries (J_M) can represent multiple disk blocks within a transaction, whereas the commit record (J_C) is always a single sector. In summary, for each application write to data (*D*), there are three logical filesystem write operations: W_1, W_2 and W_3 (see Figure 5). A logical write can be composed of multiple physical disk writes to discontiguous blocks. There is no ordering required within a logical write itself for ordered journal mode [27].

The guest filesystem also issues *cache-flush* commands (shown as F_1 and F_2 in Figure 5), wherever order ($\rightarrow$) is required between the different writes in this protocol. This is because the protocol has been originally designed to use a physical disk cache-flush interface, which provides both write ordering and immediate durability. Recent work on optimistic crash consistency proposes to decouple the cache-flush ordering and durability semantics with modifications to a single level of filesystem journal and the physical disk interface [5]. As we will show next, implementing optimistic crash consistency for vDrive through semantic journaling mainly requires changes to the virtual disk interface and the virtual I/O system, and minimal changes to the guest filesystem.

The traditional KVM/QEMU virtual I/O system (guest driver and host QEMU block driver) further translates each cache-flush command into a *fsync* system call to the virtual disk file on the host filesystem. As shown in Figure 5, the host filesystem uses another level of journaling protocol ($d \mid j_m \rightarrow j_c \rightarrow m$) separately for each of the three logical writes (d being W_1, W_2 and W_3 respectively). The nesting of the two journaling protocols has two negative performance impacts. First, it results in *write amplification*: a guest application write can be converted into upto nine writes to the physical disk. Second, it results in *write stalls*: the first guest *cache-flush* (F_1) only required for guest write *ordering* stalls the virtual

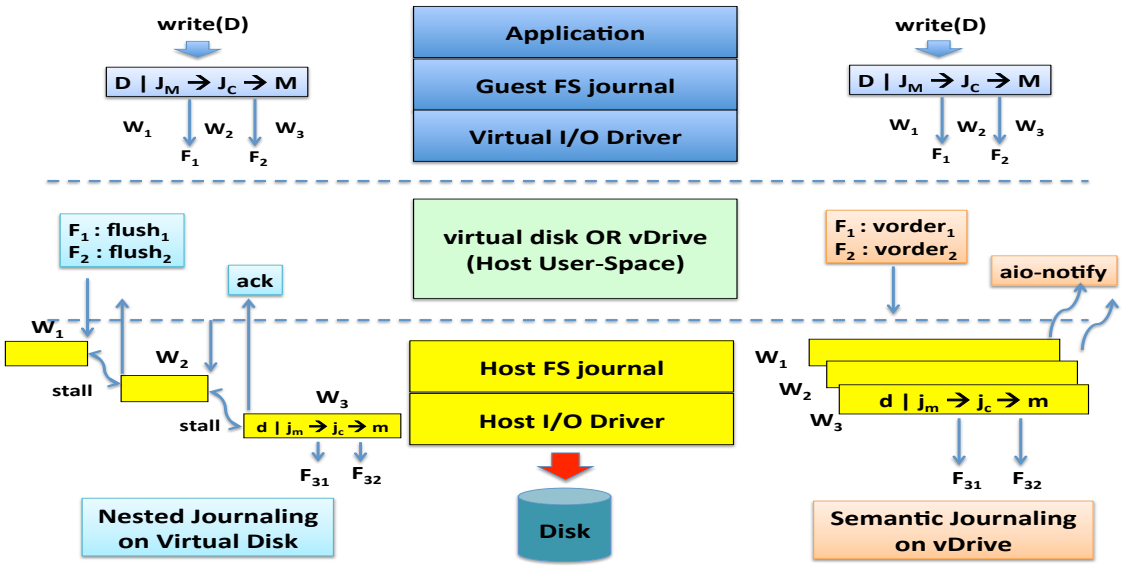

Figure 5: **Nested and Semantic Journaling:** *This figure shows the interaction between guest and host filesystem journaling for nested journaling on traditional virtual disk and semantic journaling on vDrive. The vDrive design uses* vorder *and* aio-notify *interfaces.*

I/O system until the first group of host filesystem updates ($d \mid j_m \rightarrow j_c \rightarrow m$ corresponding to W_1) have been acknowledged to be *flushed* to disk. The second group of updates corresponding to W_2 is only issued after receiving the acknowledgement for F_1. Similarly, F_2 results in another stall between the group of writes corresponding to W_2 and W_3.

In contrast, we observe that the cache manager and virtual disk are emulated entirely within the host software. As a result, the vDrive design has the flexibility to address the cost of nested journaling by using its new synchronization interfaces for *cache-flush* requests. We classify the cache-flush requests from the guest filesystem based on the semantics of the data being persisted. For simplicity, we pass the semantic information used for classifying the cache-flush requests by annotating them within the guest filesystem and virtual memory subsystems. With the use of more sophisticated approaches proposed in the past [35, 37], this classification can be also done by rather discovering the semantic information within the virtual I/O system.

The vDrive classifies the cache-flush command based on its semantic requirements for ordering and durability to provide a consistent virtual disk image after a crash. There are broadly four major classifications based on when the *cache-flush* is issued: journal transaction commit, virtual memory page write-backs, checkpoints for journal truncation, and flushing I/O queues after a disk write failure.

Journal Commits. vDrive converts all guest filesystem cache-flush commands after the journal commit record (J_C) into *vorder* request to the virtual disk. This ensures correct write ordering both within and across different guest filesystem transactions without incurring the cost of immediate durability for each journal commit. The vDrive also keeps track of the time elapsed since the last *vorder* completion has been notified through the *aio-notify* handler. If this time interval exceeds the freshness threshold for vDrive and there are pending write operations, it issues a *vflush* to the host filesystem. This ensures that the virtual disk image recovered after a crash is always consistent and has all updates older than the freshness threshold before the crash. If the workload is read-intensive, there are fewer pending write operations and the cost of a *vflush* operation is not incurred by vDrive.

VM Page Write-backs. In addition to the journal commits, the guest virtual memory subsystem also writes-back pages when the dirty to clean page ratio exceeds a configured threshold. As these write-backs happen in the background and do not require immediate durability, vDrive only requires correct ordering for them with other writes. As a result, vDrive uses the *vorder* primitive for such VM page write-backs.

Journal Truncation. When the guest journal gets full, a cleanup is required for the journal tail to re-use space in memory and disk pre-allocated for the journal. The journal metadata checkpoint (M) and all transactions corresponding to the re-used journal space are flushed to the virtual disk before the cleanup starts. vDrive issues a *vflush* request for all such cache-flush requests to en-

force immediate durability and avoid any transactions or checkpoints lost due to cleanup.

Write Failure. In addition to the journal truncation, vDrive also classifies a cache-flush request issued when a new write fails because of a stopped guest I/O scheduler queue. The guest I/O queue is stopped when the device driver signals it cannot service further requests because of full virtual/hardware disk queue. The vDrive issues a *vflush* request to flush all enqueued requests with immediate durability guarantee and only then allows the queuing of the new write request.

4.3 Crash Behavior and Recovery

With the use of default write-back virtual cache and ordered nested filesystem journaling modes, the filesystem recovers by replaying the journal upto the last fully committed transaction. However, a partially committed transaction could happen if a crash happens after the data block (*D*) or the J_M, but before the commit record has been written out to the journal. If the data write is to a new allocated block, it would result in garbage data after recovery. Otherwise, if it was an overwrite to an already allocated block, it would result in partially updated data after recovery. There is no data corruption possible in ordered journaling mode because metadata of the filesystem always points to valid data that existed before the crash. As a result, both the filesystems in guest and host will always recover to a consistent state that existed before the crash.

The vDrive design provides similar crash recovery guarantees: the filesystem always recovers to a consistent state that existed before the crash. The host filesystem still uses the ordered filesystem, and provides similar guarantees as before. With the use of *vorder* for journal commits and *vflush* operations for journal truncation, the vDrive journaling mechanism has the following ordering invariants for the guest filesystem writes:

1. Data block (*D*) and journal metadata entries (J_M) within a transaction always reach disk before the journal commit record (J_C).
2. Two different transactions are always committed in-order.
3. A transaction is never released for journal re-use before all previous checkpointed blocks (*M*) are flushed to disk.

Upon recovery, a partially committed guest transaction could still result in garbage data or partially updated data as before. However, these vDrive invariants guarantee that the metadata in the filesystem never points to invalid data, which did not exist. However, within this consistency semantics, we relax the guarantee about freshness of data to trade for higher performance. Journal replay brings the guest filesystem to a consistent state, which existed before the crash, but not necessarily the most fresh state. The vDrive architecture bounds the consistent state to be no older than the freshness threshold of the virtual disk.

5 Implementation

The implementation of vDrive entails three components: the semantic classifier, the vDrive virtual disk interface and the cache manager. The first two are implemented inside the Linux 3.2 kernel as modifications to the jbd2 journaling module for ext4 filesystem and virtio_blk virtual I/O block driver respectively. The cache manager is implemented within the QEMU block driver for raw posix I/O.

We base the semantic classifier on the jbd2 journaling layer and the virtual memory subsystem in Linux, which enables to classify the different *cache-flush* requests sent to the block layer. This makes our modifications modular and does not require changes to the ext4 filesystem structures. At the block layer, we sought to leave as much code as possible unmodified. Thus, we augment cache-flush requests from the higher layers with an additional field, effectively adding our new interface commands as sub-types of the existing cache-flush command. The write to the commit record in the journal transaction commit adds the *vorder* sub-type. Similarly, the writes from the virtual memory writeback thread also adds the *vorder* sub-type to the associated flush command. However, the writes during the journal cleanup for checkpointing add the *vflush* sub-type to ensure that journal space is not re-used for a transaction before it is committed and its metadata is checkpointed. The I/O queue restart code-path also uses the *vflush* sub-type to flush all the preceeding requests in the queue on a write failure. The block layers pass the sub-type field down to the next layer. We modified the I/O scheduler to consider both sub-types as a traditional cache-flush command while servicing them. This enables us to achieve the same ordering guarantees for *vorder* as for a traditional cache-flush request in the guest operating system.

We modify the guest virtio_blk driver, which implements the interface to the virtual block device in the host emulated using the QEMU block driver. It inserts the I/O requests in the virtio_ring buffer and triggers the QEMU block driver to service those requests using a work queue of threads. We modified the virtio_blk driver to insert a different *vorder* and *vflush* request into the virtio_ring buffer based on the sub-type of the command received from the block request queue in guest. The QEMU emulation framework pops off the request from the virtio_ring and hands it to the QEMU block driver.

We base our cache manager implementation on the block driver for raw posix I/O in QEMU. The raw

posix driver provides best performance than other drivers supporting different image formats and implementing additional features [28]. We modify the code path for handling cache-flush requests in the block driver. The QEMU block driver issues a *fdatasync* system call for each cache-flush command received from the guest driver. We modify it to issue an *aio_fsync* system call with D_SYNC flag for a corresponding *vorder* command received from guest. On vDrive initialization, we setup an asynchronous I/O control block, signal event handler for aio_fsync implementing the *aio-notify* notification, and a freshness threshold timer. We reset the freshness timer on each *aio-notify* notification or *vflush* completion. Similarly, we update the number of write operations issued so far on each *pwritev* call to the host filesystem. The cache manager implementation has the flexibility to use alternative notification interfaces with different semantics such as posix inotify, asynchronous durability notifications [5], and callbacks implemented on PCI-e based storage systems [41, 33]. The cache manager also forces a *vflush* operation if there are pending write operations and the freshness timer exceeds the freshness threshold of vDrive. We configure the freshness threshold to match the average latency of a single cache-flush request: 50 ms. We find through experiments that this provides good runtime performance and still tightly bounds the virtual disk freshness within the cache-flush request latency.

6 Evaluation

We evaluate vDrive's design components against traditional virtual disk storage on two axes: performance and reliability.

6.1 Methods

Experiments were performed on a machine equipped with two sockets and 8 cores-per-socket Intel Xeon E5530 CPU running at 2.4 GHz with 24 GB of memory, 146 GB 10K RPM IBM 42D0421 SAS drive, and running Linux 3.2 and KVM/QEMU 1.5.3. We use an Intel S3700 SATA SSD [17] and PCI-e Fusion-io SSD [11] for experiments reported in Section 3. The guest virtual machine is configured with 4 GB memory, 4 core processor and a virtual disk of 60 GB.

We compare the vDrive virtual I/O system using semantic journaling and new virtual disk interfaces against the *baseline* system, which uses the unmodified KVM/QEMU virtual disk. All systems use Linux ext4 ordered mode filesystems in both guest and host operating systems. The *baseline* system is configured in three different modes by using different barrier options to configure cache-flushes when mounting the filesystems: *nested* journal with disk cache-flushes enabled in both guest and host, only the *guest* journal enabled with cache-flushes, *no* journal enabled with cache-flushes. These three modes present different performance and consistency tradeoffs for the *baseline* system. We use raw posix I/O image and write-back guest disk cache mode as they both provide best performance across different virtual disk image configurations.

We first measure performance of vDrive under a number of micro- and macro-benchmarks (Section 6.2) running in the guest. We use random and sequential write workloads with 200 K writes with an fsync every 1K writes over a 10 GB file. These micro-benchmarks are used to exercise the journaling code paths and are different from the micro-benchmarks used in Section 3 to isolate the performance difference between different cache modes. We run the filebench varmail (1:1:1 reads/writes/fsyncs), fileserver (1:2 reads/writes), and webserver (10:1 reads/writes) workloads [1] and MySQL benchmark (200K OLTP transactions on a table of 1M rows) from sysbench [2]. These macro-benchmarks exercise all the read, write and fsync code paths of the virtual storage stack.

We also perform two case studies (Section 6.3) to demonstrate how vDrive provides crash consistency and recovery using atomic writes used within a text editor (gedit) and log management within a database (SQLite) running within the guest. We write a tool to capture block-level I/O traces at the host virtual disk layer and analyze the crash behavior by applying a subset of writes from the trace to the original disk image to reconstruct the image after crash.

6.2 System Comparison

Figure 6 shows the performance of the vDrive system and three different journaling mode combinations (see Section 6.1) normalized to nested journaling in the baseline system. The first four workloads: varmail, MySQL, rand-write, and seq-write are write/fsync intensive, thus exercise the journaling disk cache-flush interface. The fileserver benchmark is write-intensive and is bound by the overhead of frequent metadata updates for append/delete system calls to the guest filesystem. Similarly, the webserver benchmark is read-intensive and is bound by the random read performance of the virtual disk. As a result, these two workloads get most of their requests hit in host caches and have minimal performance impact of 8%-11% because of journaling or cache-flushes.

For the four write/fsync-intensive workloads, disabling cache-flushes for host journal (*guest* system) improves performance by 15%-56%, but disabling both guest and host (*no* system) cache-flushes has a much larger impact. The guest journal impacts performance more because it flushes out guest disk cache and host

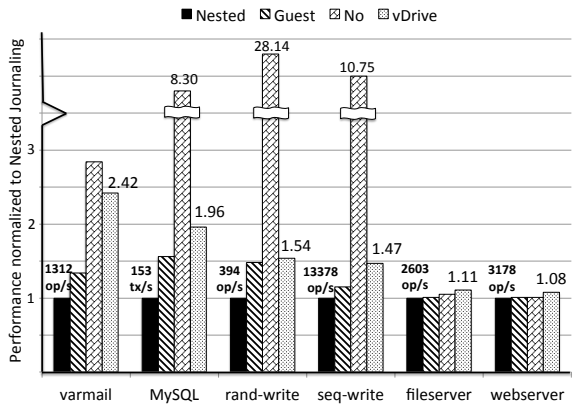

Figure 6: **Application Performance:** The performance of *guest*-only journaling, *no* journaling, and *vDrive*'s semantic journaling normalized to *nested* journaling. The absolute performance of *nested* journaling and improvement of *vDrive*'s semantic journaling are shown as labels in the figure.

page cache on a journal commit. This corrobates with the vDrive design, which exploits the guest journal semantics to reduce its overhead.

The vDrive system with semantic journaling outperforms nested journaling by 47%-142%. This improvement is mainly due to the use of the *vorder* interface, which allows for lower write stall times and more parallel writes to the host filesystem while still maintaining the correct order for writes to disk. In contrast, the *no* cache-flush system does not provide any ordering or durability for writes and result in substantial performance improvement over nested journaling, especially for the write-only random and sequential workloads (10x-28x improvements over nested journaling). This is because all writes in these workloads complete out-of-order and get acknowledged as soon as they hit any of the in-memory cache levels within the host. The vDrive system is 22%-95% slower than the *no* cache-flush system, which is attributed to the cost it pays for correct write ordering using the *vorder* and bounded freshness using the *vflush* interface for guest journal commits.

6.3 Case Studies

We now evaluate how vDrive's crash behavior compares against nested journaling. We use two applications: an atomic write to a file within a popular text editor (gedit) and log management within the SQLite database.

Table 2 shows the crash behavior for *nested* journaling on the *baseline* system and semantic journaling on vDrive respectively. We capture block-level traces at the host virtual disk layer when executing the application in guest. We then simulate crashpoints by applying subsets

	Gedit		SQLite	
System	Nested	vDrive	Nested	vDrive
Total Crashpoints	10	10	30	30
Old State	3	5	12	21
New State	7	5	18	9

Table 2: **Crash Behavior.** The number of crashpoints simulated for two applications, and the resulting application states after remounting the filesystem in the guest on the virtual disk image obtained after crash. Old and new states correspond to before and after application execution.

of writes from the trace to the original virtual disk image to reconstruct the crashpoint virtual disk image. We then restart the guest virtual machine, remount the filesystem on the reconstructed virtual disk image, and run recovery to test for crash behavior.

Atomic Write. The atomic write to a file within gedit creates a new version of the file under a temporary name, issues a fsync to flush the atomic write to disk, and renames the temporary file to the original file name. As a result, an atomic write ensures that either the old contents before the write (*old* state) or new contents after the write (*new* state) are available in the file in entirety, both of which are consistent states.

As shown in Table 2, nested journaling uses the cache-flush interface and points to either the old or new contents of the file after recovery. Semantic journaling preserves the same crash behavior as nested journaling by always recovering to a consistent state. However, it recovers to the old state more frequently than nested journaling. This is because journal commits use the *vorder* interface in vDrive, which delays durability for writes. Neither of the two modes result in an inconsistent state where the filesystem was not able to recover or the recovered file did not have old or new contents or have a mix of contents.

Log Management. The SQLite database uses a two fsync sequence for committing its transactions: first after committing its writes to a log file, and second after making in-place updates to the database. We simulate a SQLite transaction transferring data across a set of tables. After the filesystem recovery, the SQLite database performs its own recovery to reach to a consistent state, which reflects either pre-transaction (*old* state) or post-transaction (*new* state) changes.

As earlier, recovery on vDrive results in the old pre-transaction state for about 70% of the crashpoints. The vDrive design trades off the freshness of the virtual disk image for application runtime performance. For the nested journal, we find that the database recovers to the post-transaction state more often. This is because there

is a higher chance that either the filesystem or the SQLite will recover to the new state. If the first fsync is not committed, the filesystem recovers to an old state and the database recovery performs a roll-back from the log file. If the first fsync is committed, the filesystem recovers to an old state but the database recovers to the new state by replaying the log changes. If both the fsyncs are committed, both the filesystem and database recover to the new state. In all cases, both vDrive and baseline systems always recover to either the old or new state, both of which are consistent.

We also tested the *no* cache-flush journaling mode from Section 6.2, which provides performance higher than semantic journaling, but causes delayed as well as re-ordered writes. In this mode, the crashpoints resulted in different inconsistent images with unmountable or read-only filesystems, orphaned filesystem inodes, and data corruption with mixed contents from the old and new states.

7 Related Work

Our work builds on past work on virtual I/O, new storage interfaces, and consistent storage systems.

Virtual I/O Performance. Recent work has sought to improve the performance of I/O virtualization. ELVIS [15] and ELI [14] improve virtual I/O performance for network and read/write I/O in KVM/QEMU. They propose the use of exit-less interrupts and polling to reduce the number of switches between guest and host. Further, they scale virtual I/O performance by using x86 hardware support for posted interrupts. Past work [31, 25] has also investigated guest scheduling mechanisms to improve virtual network and I/O performance in the context of Xen [4]. Similarly, the performance overheads for nested filesystems have been analyzed with advice implied for new filesystem designs [21]. In contrast, vDrive is the first system implementation to account for the performance and consistency tradeoffs for journaling in nested filesystems. vDrive provides high performance for crash consistency with the use of journaled filesystems, and does not require any costly additional data copies as in other consistency techniques such as local snapshots and distributed replication [28, 39].

New Storage Interfaces. Recent proposals have investigated storage interface extensions for specific use cases such as flash caching [32, 10, 23], new filesystems [41, 19, 5], flash databases [26], and key-value stores [12]. Most of these works have benefitted from the internal mechanisms within a flash SSD such as logging, garbage collection and sparse address spaces. However, such a tight integration mostly requires offloading software functionality to hardware with changes to the standard SATA/SCSI protocols [33, 41, 5, 30] or custom communication channels [12, 22] on the PCI-e bus. In contrast, this work specifically targets disk storage and virtual environments where the benefits of providing high performance data consistency are significantly higher than flash storage. In addition, Xen and virtio support paravirtual I/O barriers for SCSI block devices to enforce write ordering only limited for write-through disk caching [30, 29, 4]. vDrive provides significantly better control than these operations by using a combination of *vflush*, *vorder* and flush notifications that work for all caching modes in guest and host systems.

Consistent Storage. A number of approaches to building higher performing and crash-consistent filesystems have been investigated in the past [13, 9, 6]. However, most of these proposals either increase filesystem complexity significantly [13, 6] or provide very generalized frameworks to order filesystem updates [9]. Recent works on filesystem [5], databases [7] and storage protocols [30] propose the use of optimistic crash consistency and closely relate to vDrive's design principles. In this work, we explore the first use of optimistic crash consistency to address the performance overheads of nested journaling. We find that it requires major changes to the virtual I/O system and rethinking the software interface between the guest and host operating systems. Recently, new optimizations have also been proposed to minimize the overheads for nested journaling arising from SQLite log and ext4 filesystem journal in the Android I/O stack on flash storage [18, 20, 34]. However, those optimizations focus primarily on cheap and slow flash storage popularly used in smartphones or tablets, and are not directly applicable to virtual disk I/O stack.

8 Conclusions

Virtual I/O has been widely adopted in production environments for application and infrastructure services in the recent past. We present a first look on the crash consistency and performance tradeoffs for virtual I/O. Our findings reveal deep insights into the performance costs and interaction of nested filesystem journals. We propose a new virtual disk interface and semantic journaling technique to address the performance overheads associated with nested journaling. As new non-volatile memory technologies become available, such as phase-change and storage-class memory, it will be important to revisit the interface to further address the system call overheads for virtual I/O.

Acknowledgements

We thank our shepherd Nitin Agrawal and the anonymous reviewers for their helpful comments and useful feedback.

References

[1] Filebench. http://sourceforge.net/projects/filebench.

[2] Sysbench benchmark. https://launchpad.net/sysbench.

[3] AMAZON. Elastic Compute Cloud2. http://aws.amazon.com/ec2.

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP* (2003).

[5] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *SOSP* (2013).

[6] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *FAST* (2012).

[7] CIPAR, J., GANGER, G., KEETON, K., III, C. B. M., SOULES, C. A., AND VEITCH, A. Lazybase: Trading freshness for performance in a scalable database. In *EuroSys* (2012).

[8] EXT4 FILESYSTEM. external journal caveats. http://www.raid6.com.au/posts/fs_ext4_external_journal_caveats.

[9] FROST, C., MAMMARELLA, M., KOHLER, E., DE LOS REYES, A., HOVSEPIAN, S., MATSUOKA, A., AND ZHANG, L. Generalized filesystem dependencies. In *SOSP* (2007).

[10] FUSION-IO INC. directCache. http://www.fusionio.com/data-sheets/directcache.

[11] FUSION-IO INC. ioDrive SSD Datasheet. http://www.fusionio.com/products/iodrive.

[12] FUSION-IO INC. ioMemory Application SDK. http://www.fusionio.com/products/iomemorysdk.

[13] GANGER, G. R., AND PATT, Y. N. Metadata update performance in filesystems. In *OSDI* (1994).

[14] GORDON, A., AMIT, N., HAR'EL, N., AND BEN-YEHUDA, M. Eli: bare-metal performance for i/o virtualization. In *ASPLOS* (2012).

[15] HAR'EL, N., GORDON, A., LANDAU, A., BEN-YEHUDA, M., TRAEGER, A., AND LADELSKY, R. Efficient and scalable paravirtual i/o system. In *Usenix ATC* (2013).

[16] IBM. Cloud Computing. www.ibm.com/cloud-computing.

[17] INTEL INC. S3700 Series 200GB SSD. http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-dc-s3700-series.html.

[18] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/o stack optimizations for smartphones. In *Usenix ATC* (2013).

[19] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. DFS: a file system for virtualized flash storage. In *FAST* (2010).

[20] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving journaling of journal anomaly in android i/o: Multi-version b-tree with lazy split. In *FAST* (2014).

[21] LE, D., HUANG, H., AND WANG, H. Understanding performance implications of nested file systems in a virtualized environment. In *FAST* (2012).

[22] MARVELL CORP. Dragonfly platform family. http://www.marvell.com/storage/dragonfly/, 2012.

[23] MESNIER, M., AKERS, J. B., CHEN, F., AND LUO, T. Differentiated storage services. In *SOSP* (2011).

[24] MICROSOFT. Windows Azure. http://www.windowsazure.com.

[25] ONGARO, D., COX, A. L., AND RIXNER, S. Scheduling i/o in virtual machine monitors. In *VEE* (2008).

[26] OUYANG, X., NELLANS, D., WIPFEL, R., FLYNN, D., AND D.K.PANDA. Beyond block i/o: Rethinking traditional storage primitives. In *HPCA* (feb. 2011), pp. 301 –311.

[27] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling filesystems. In *Usenix ATC* (2005).

[28] QEMU. Virtual Disk Image Formats. http://en.wikibooks.org/wiki/QEMU/Images.

[29] RUSSELL, R. virtio: Towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating System Review 42*, 5 (July 2008).

[30] RUSSELL, R. Virtio pci card specification, May 2012. http://ozlabs.org/~rusty/virtio-spec/virtio-0.9.5.pdf.

[31] SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G., AND PRATT, I. Bridging the gap between software and hardware techniques for i/o virtualization. In *Usenix ATC* (2008).

[32] SAXENA, M., SWIFT, M. M., AND ZHANG, Y. FlashTier: A lightweight, consistent and durable storage cache. In *EuroSys* (2012).

[33] SAXENA, M., ZHANG, Y., SWIFT, M. M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Getting real: Lessons in transitioning research simulations into hardware systems. In *FAST* (2013).

[34] SHEN, K., PARK, S., ZHU, M., SHEN, K., PARK, S., AND ZHU, M. Journaling of journal is (almost) free. In *FAST* (2014).

[35] SIVANTHU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems. In *FAST* (2003).

[36] STEIGERWALD, M. Imposing order. In *Linux Magazine* (2007).

[37] TARASOV, V., JAIN, D., HILDEBRAND, D., TEWARI, R., KUENNING, G., AND ZADOK, E. Improving i/o performance using virtual disk introspection. In *HotStorage* (2013).

[38] VMWARE. Performance evaluation of vmxnet3 virtual network device. In *Technical Report* (2009).

[39] VMWARE INC. VMWare Virtual Machine File System: Technical Overview and Best Practices. http://www.vmware.com/pdf/vmfs-best-practices-wp.pdf.

[40] YASSOUR, B.-A., BEN-YEHUDA, M., AND WASSERMAN, O. Direct device assignment for untrusted fully-virtualized virtual machines. In *IBM Research Report H-0263* (2008).

[41] ZHANG, Y., ARULRAJ, L. P., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. De-indirection for flash-based ssds with nameless writes. In *FAST* (2012).