ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices

Zev Weiss*, Sriram Subramanian†, Swaminathan Sundararaman†,
Nisha Talagala†, Andrea C. Arpaci-Dusseau*, Remzi H. Arpaci-Dusseau*
* *University of Wisconsin-Madison,* † *SanDisk, Inc.*

Abstract

We present a new form of storage virtualization based on block-level address remapping. By allowing the host system to manipulate this address map with a set of three simple operations (clone, move, and delete), we enable a variety of useful features and optimizations to be readily implemented, including snapshots, deduplication, and single-write journaling. We present a prototype implementation called Project ANViL and demonstrate its utility with a set of case studies.

1 Introduction

Virtualization has been widely employed as a technique for managing and exploiting the available resources in computing systems, from memory and processors to entire machines [1,2,4,5,9,22]. Virtual memory in particular has enabled numerous features and optimizations, including the `mmap(2)` interface to file I/O, shared libraries, efficient `fork(2)`, zero-copy I/O, and page sharing between virtual machines [3, 29].

Storage virtualization, however, while conceptually similar to memory virtualization, has typically been of limited use to applications, focusing instead on storage management by introducing abstraction between physical storage layout and the logical device as presented to a host or application using it [10, 13, 28]. Features and functionality enabled by storage virtualization, such as deduplication, replication, and thin-provisioning, remain hidden behind the block device interface. While highly useful, the features of existing storage virtualization systems are primarily limited to administrative functionality, such as defining and provisioning LUNs, offering nothing to actual applications beyond standard read and write operations. As others have shown, these limitations in storage virtualization result in sub-optimal application performance and duplication of functionality across different layers in the storage stack [8, 11, 18, 21].

Some of the limits of storage virtualization have been addressed in recent research on Flash Translation Layers (FTLs), with new machinery proposed to support Atomic Writes, Persistent TRIM, and Sparseness [17, 18, 20, 21, 24]. These extensions enable applications to better leverage the power of virtualization already built into the FTL and also extensions enable the removal of redundant functionality across system layers, resulting in better flash endurance and application-level performance [16, 21].

We propose a simple yet powerful set of primitives based on *fine-grained address remapping* at both the block and extent level. As we will show, fine-grained address remapping provides the flexibility needed to benefit applications while still retaining the generality necessary to provide the functionality offered by existing virtualized volume managers. By allowing the host to manipulate the block-level logical-to-physical address map with *clone*, *move*, and *delete* operations, we enable storage virtualization to more closely resemble virtualized memory in its fine-grained flexibility and broad utility, though in a manner adapted to the needs of persistent storage.

We illustrate the utility of our approach by developing the Advanced Nonvolatile-memory Virtualization Layer (ANViL), a prototype implementation of fine-grained address remapping as a stacking block device driver, to efficiently implement both file and volume snapshots, deduplication, and single-write journaling. More specifically, we demonstrate how ANViL can provide high performance volume snapshots, offering as much as a 7× performance improvement over an existing copy-on-write implementation of this feature. We show how ANViL can be used to allow common, conventional file systems to easily add support for file-level snapshots without requiring any radical redesign. We also demonstrate how it can be leveraged to provide a performance boost of up to 50% for transactional commits in a journaling file system.

The remainder of this paper is organized as follows: we begin by describing how ANViL fits in naturally in the context of modern flash devices (§2) and detailing the extended device interface we propose (§3). We then discuss the implementation of ANViL (§4), describe a set of case studies illustrating a variety of useful real-world applications (§5), and conclude (§6).

2 Background

Existing storage virtualization systems focus their feature sets primarily on functionality "behind" the block interface, offering features like replication, thin-provisioning, and volume snapshots geared toward simplified and improved storage administration [10, 28]. They offer little, however, in the way of added functionality to the *consumers* of the block interface: the file systems, databases, and other applications that actually access data from the virtualized storage. Existing storage technologies, particularly those found in increasingly-popular flash devices, offer much of the infrastructure necessary to provide more advanced storage virtualization that could provide a richer interface directly beneficial to applications.

At its innermost physical level, flash storage does not offer the simple read/write interface of conventional hard disk drives (HDDs), around which existing storage software has been designed. While reads can be performed simply, a write (or *program*) operation must be preceded by a relatively slow and energy-intensive *erase* operation on a larger erase block (often hundreds of kilobytes or larger), before which any live data in the erase block must be copied elsewhere. Flash storage devices typically employ a *flash translation layer* (FTL) to simplify integration of this more complex interface into existing systems by adapting the native flash interface to the simpler HDD-style read/write interface, hiding the complexity of program/erase cycles from other system components and making the flash device appear essentially as a faster HDD. In order to achieve this, FTLs typically employ log-style writing, in which data is never overwritten in-place, but instead appended to the head of a log [23]. The FTL then maintains an internal address-remapping table to track which locations in the physical log correspond to which addresses in the logical block address space provided to higher layers of the storage stack [12, 26].

Such an address map provides most of the machinery that would be necessary to provide more sophisticated storage virtualization, but its existence is not exposed to the host system, preventing its capabilities from being fully exploited. A variety of primitives have been proposed to better expose the internal power of flash translation layers and similar log and remapping style systems, including atomic writes, sparseness (thin provisioning), Persistent TRIM, and cache-friendly garbage collection models [18, 20, 21, 24, 30]. These have been shown to have value for a range of applications from file systems to databases, key-value stores, and caches.

3 Interfaces

Address-remapping structures exist in FTLs and storage engines that provide thin provisioning and other storage virtualization functions today. We propose an extended block interface that enables a new form of storage virtualization by introducing three operations to allow the host system to directly manipulate such an address map.

3.1 Operations

Range Clone: `clone(src, len, dst)`: The range clone operation instantiates new mappings in a given range of logical address space (the *destination* range) that point to the same physical addresses mapped at the corresponding logical addresses in another range (the *source* range); upon completion the two ranges share storage space. A read of an address in one range will return the same data as would be returned by a read of the corresponding address in the other range. This operation can be used to quickly relocate data from one location to another without incurring the time, space, and I/O bandwidth costs of a simplistic read-and-rewrite copy operation.

Range Move: `move(src, len, dst)`: The range move operation is similar to a range clone, but leaves the source logical address range unmapped. This operation has the effect of efficiently transferring data from one location to another, again avoiding the overheads of reading in data and writing it back out to a new location.

Range Delete: `delete(src, len)`: The range delete operation simply unmaps a range of the logical address space, effectively deleting whatever data had been present there. This operation is similar to the `TRIM` or `DISCARD` operation offered by existing SSDs. However, unlike `TRIM` or `DISCARD`, which are merely advisory, the stricter range delete operation guarantees that upon acknowledgment of completion the specified logical address range is persistently unmapped. Range deletion is conceptually similar to the Persistent TRIM operation defined in prior work [15, 20]. Our work extends previous concepts by combining this primitive with the above clone and move operations for additional utility.

Under this model, a given logical address can be either mapped or unmapped. A read of a mapped address returns the data stored at the corresponding physical address. A read of an unmapped address simply returns a block of zeros. A write to a logical address, whether mapped or unmapped, allocates a new location in physical storage for the updated logical address. If the logical address previously shared physical space with one or more additional logical addresses, that mapping will be decoupled, with the affected logical address now pointing to a new physical location while the other logical addresses retain their original mapping.

3.2 Complementary Properties

While giving the host system the ability to manipulate the storage address map is of course the primary aim of our proposed interface, other properties complement our interfaces nicely and make them more useful in practice for real-world storage systems.

Sparseness or Thin Provisioning: In conventional storage devices, the logical space exposed to the host system is mapped one-to-one to the (advertised) physical capacity of the device. However, the existence of the range clone operation implies that the address map must be many-to-one. Thus, in order to retain the ability to utilize the available storage capacity, the logical address space must be expanded – in other words, the device must be *thin-provisioned* or *sparse*. The size of the logical address space, now decoupled from the physical capacity of the device, determines the upper limit on the total number of cloned mappings that may exist for a given block.

Durability: The effects of a range operation must be crash-safe in the same manner that an ordinary data write is: once acknowledged as complete, the alteration to the address map must persist across a crash or power loss. This requirement implies that the metadata modification must be synchronously persisted, and thus that each range operation implies a write to the underlying physical storage media.

Atomicity: Because it provides significant added utility for applications in implementing semantics such as transactional updates, we propose that a vector of range operations may be submitted as a single atomic batch, guaranteeing that after a crash or power loss, the effects of either *all* or *none* of the requested operations will remain persistent upon recovery. Log-structuring (see §4.1) makes this relatively simple to implement.

4 Implementation

In this section we describe the implementation of our prototype, the Advanced Nonvolatile-memory Virtualization Layer (ANViL), a Linux kernel module that acts as a generic stacking block device driver. ANViL runs on top of single storage devices as well as RAID arrays of multiple devices and is equally at home on either. It is not a full FTL, but it bears a strong resemblance to one. Though an implementation within the context of an existing FTL would have been a possibility, we chose instead to build ANViL as a separate layer to simplify development.

4.1 Log Structuring

In order to support the operations described earlier (§3), ANViL is implemented as a log-structured block device. Every range operation is represented by a note written to the log specifying the point in the logical ordering of updates at which it was performed. The note also records the alterations to the logical address map that were performed; this simplifies reconstruction of the device's metadata after a crash. Each incoming write is redirected to a new physical location, so updates to a given logical range do not affect other logical ranges which might share physical data. Space on the backing device is managed in large segments (128MB by default); each segment is written sequentially and a log is maintained that links the segments together in temporal order.

4.2 Metadata Persistence

Whenever ANViL receives a write request, before acknowledging completion it must store in non-volatile media not only the data requested to be written, but also any updates to its own internal metadata necessary to guarantee that it will be able to read the block back even after a crash or power loss. The additional metadata is small (24 bytes per write request, independent of size), but due to being a stacked layer of the block IO path, writing an additional 24 bytes would require it to write out another entire block. Done naïvely, the extra blocks would incur an immediate 100% write amplification for a workload consisting of single-block writes, harming both performance and flash device lifespan. However, for a workload with multiple outstanding write requests (a write IO queue depth greater than one), metadata updates for multiple requests can be batched together into a single block write, amortizing the metadata update cost across multiple writes.

ANViL thus uses an adaptive write batching algorithm, which, upon receiving a write request, waits for a small period of time to see if further write requests arrive, increasing the effectiveness of this metadata batching optimization, while balancing the time spent waiting for another write with impact on the latency of the current write.

4.3 Space Management

Space on the backing device is allocated at block granularity for incoming write requests. When a write overwrites a logical address that was already written and thus mapped to an existing backing-device address, the new write is allocated a new address on the backing device and the old mapping for the logical address is deleted and replaced by a mapping to the new backing device address. When no mappings to a given block of the backing device remain, that block becomes "dead" and its space may be reclaimed. However, in order to maintain large regions of space in the backing device so as to allow for sequential writing, freeing individual blocks as they become invalid is not a good approach for ANViL. Instead, the minimum unit of space reclamation is one segment.

A background garbage collector continuously searches

for segments of backing device space that are underutilized (i.e. have a large number of invalid blocks). When such a segment is found, its remaining live blocks are copied into a new segment (appended at the current head of the log as with a normal write), any logical addresses mapped to them are updated to point to the new location they have been written out to, and finally the entire segment is returned to the space allocator for reuse.

5 Case Studies

Here we demonstrate the generality and utility of our range operations by implementing, with relatively little effort, a number of features useful to other components across a broad range of the storage stack, including volume managers (enabling simple and efficient volume snapshots), file systems (easily-integrated file snapshots), and transactional storage systems such as relational databases (allowing transactional updates without the double-write penalty). All experiments were performed on an HP DL380p Gen8 server with two six-core (12-thread) 2.5GHz Intel Xeon processors and a 785GB Fusion-io ioDrive2, running Linux 3.4.

5.1 Snapshots

Snapshots are an important feature of modern storage systems and have been implemented at different layers of the storage stack from file systems to block devices [25]. ANViL easily supports snapshots at multiple layers; here we demonstrate file- and volume-level snapshots.

5.1.1 File Snapshots

File-level snapshots enable applications to checkpoint the state of individual files at arbitrary points in time, but are only supported by a few recent file systems [7]. Many widely-used file systems, such as ext4 [19] and XFS [27], do not offer file-level snapshots, due to the significant design and implementation complexity required.

ANViL enables file systems to support file-level snapshots with minimal implementation effort and no changes to their internal data structures. Snapshotting individual files is simplified with range clones, as the file system has only to allocate logical address space and issue a range operation to clone the address mappings from the existing file into the newly-allocated address space [14].

With just a few hundred lines of code, we have added an `ioctl` to ext4 to allow a zero-copy implementation of the standard `cp` command, providing an efficient (in both space and time) file-snapshot operation. Figure 1 shows, for varying file sizes, the time taken to copy a file using the standard `cp` command on an ext4 file system mounted on an ANViL device in comparison to the time taken to copy the file using our special range-clone `ioctl`. Unsurprisingly, the range-clone based file copy is dramatically faster than the conventional read-and-write approach used by the unmodified `cp`, copying larger files in orders of magnitude less time. Also, unlike standard `cp`, the clone based implementation shares physical space between copies, making it vastly more storage efficient as normal for thinly provisioned snapshots.

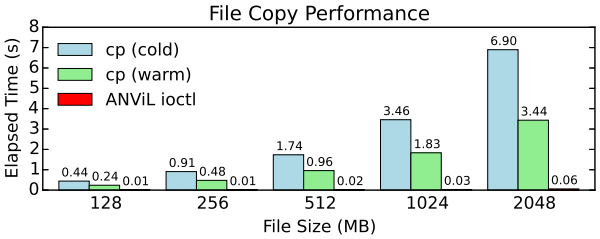

Figure 1: Time to copy files of various sizes via standard `cp` with both a cold and a warm page cache, and using a special ANViL `ioctl` in our modified version of ext4.

5.1.2 Volume Snapshots

Volume snapshots are similar to file snapshots, but even simpler to implement. We merely identify the range of blocks that represent a volume and clone it into a new range of logical address space, which a volume manager can then provide access to as an independent volume.

Volume snapshots via range-clones offer much better performance than the snapshot facilities offered by some existing systems, such as Linux's built-in volume manager, LVM. LVM snapshots are (somewhat notoriously) slow, because they operate via copy-on-write of large extents of data (2MB by default) for each extent that is written to in the volume of which the snapshot was taken. To quantify this, we measure the performance of random writes at varying queue depths on an LVM volume and on ANViL, both with and without a recently-activated snapshot. In Figure 2, we see that while the LVM volume suffers a dramatic performance hit when a snapshot is active, ANViL sees little change in performance, since it instead uses its innate redirect-on-write mechanism.

5.2 Deduplication

Data deduplication is often employed to eliminate data redundancy and better utilize storage capacity by identifying pieces of identical data and collapsing them together to share the same physical space. Deduplication can of course be implemented easily using a range clone operation. As with snapshots, deduplication can be performed at different layers of the storage stack. Here we show how block-level deduplication can be easily supported by a file system running on top of an ANViL device.

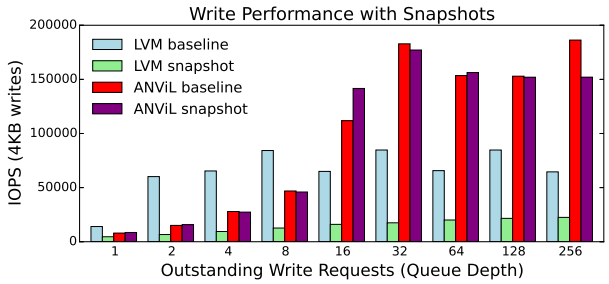

Figure 2: Random write IOPS on ANViL and LVM, both in isolation and with a recently-activated snapshot. The baseline bars illustrate ANViL's raw I/O performance. Its relatively low performance at small queue depths is due to the overhead incurred by its metadata updates.

Extending the same `ioctl` used to implement file snapshots (§5.1.1), we added an optional flag to specify that the file system should, as a single atomic operation, read the two indicated file ranges and then conditionally perform a range clone if and only if they contain identical data. This operation provides a base primitive that can be used as the underlying mechanism for a userspace deduplication tool, with the atomicity necessary to allow it to operate safely in the presence of possible concurrent file modifications. Without this locking it would risk losing data written to files in a time-of-check-to-time-of-use race between the deduplicator detecting that two block ranges are identical (the check) and performing the range-copy operation (the use). While the simplistic proof-of-concept deduplication system we have is unable to detect previously-deduplicated blocks and avoid re-processing them, the underlying mechanism could be employed by a more sophisticated offline deduplicator without this drawback (or even, with appropriate plumbing, an online one).

5.3 Single-Write Journaling

Journaling is widely used to provide atomicity to multi-block updates and thus ensure metadata (and sometimes data) consistency in systems such as databases and file systems. Such techniques are required because storage devices typically do not provide any atomicity guarantees beyond a single block write. Unfortunately, journaling causes each journaled update to be performed twice: once to the journal region and then to the final location of the data. In case of failure, updates that have been committed to the journal are replayed at recovery time, and uncommitted updates are discarded. ANViL, however, can leverage its redirect-on-write nature and internal metadata management to support a multi-block atomic write operation. With this capability, we can avoid the double-write penalty of journaling and thus improve both performance and the lifespan of the flash device.

By making a relatively small modification to a journaling file system, we can use a vectored atomic range move operation to achieve this optimization. When the file system would write the commit block for a journal transaction, it instead issues a single vector of range moves to atomically relocate all metadata (and/or data) blocks in the journal transaction to their "home" locations in the main file system. Figure 3 illustrates an atomic commit operation via range moves. This approach is similar to Choi et al.'s JFTL [6], though unlike JFTL the much more general framework provided by ANViL is not tailored specifically to journaling file systems.

Using range moves in this way obviates the need for a second write to copy each block to its primary location, since the range move has already put them there, eliminating the double-write penalty inherent to conventional journaling. This technique is equally applicable to metadata journaling and full data journaling; with the latter this means that a file system can achieve the stronger consistency properties offered by data journaling without paying the penalty of the doubling of write traffic incurred by journaling without range moves. By halving the amount of data written, flash device lifespan is also increased.

Commit-via-range-move also obviates the need for any journal recovery at mount time, since any transaction that has committed will need no further processing or IO, and any transaction in the journal that has not completed should not be replayed anyway (for consistency reasons). This simplification would allow the elimination of over 700 lines of (relatively intricate) recovery code from the jbd2 codebase.

In effect, this approach to atomicity simply exposes to the application (the file system, in this case) the internal operations necessary to stitch together a vectored atomic write operation from more primitive operations: the application writes its buffers to a region of scratch space (the journal), and then, once all of the writes have completed, issues a single vectored atomic range move to put each block in its desired location.

We have implemented single-write journaling in ext4's jbd2 journaling layer; it took approximately 100 lines of new code and allowed the removal of over 900 lines of existing commit and recovery code. Figure 4 shows the performance results for write throughput in data journaling mode of a process writing to a file in varying chunk sizes and calling `fdatasync` after each write. In all cases ext4a (our modified, ANViL-optimized version of ext4) achieves substantially higher throughput than the baseline ext4 file system. At small write sizes the relative gain of ext4a is larger, because in addition to eliminating the double-write of file data, the recovery-less

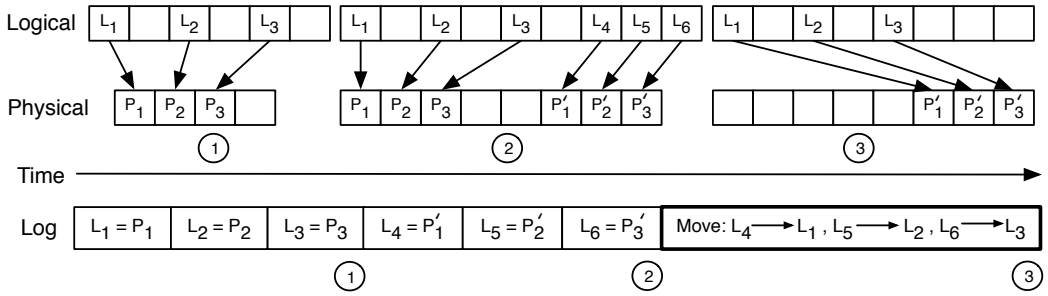

Figure 3: **Transactions via address remapping** *By using an application-managed scratch area, atomic transactional updates can be implemented using range operations. At ① the system is in its consistent pre-transaction state, with logical blocks L_1, L_2, and L_3 each mapped to blocks containing the initial versions of the relevant data. Between ① and ②, new versions of these blocks are written out and mapped to logical addresses in a temporary scratch area (L_4, L_5, and L_6). Note that this portion is not required to proceed atomically. Once the temporary locations have all been populated, an atomic range-move operation remaps the new blocks at L_4, L_5, and L_6 to L_1, L_2, and L_3, respectively, leading to ③, at which point the transaction is fully committed.*

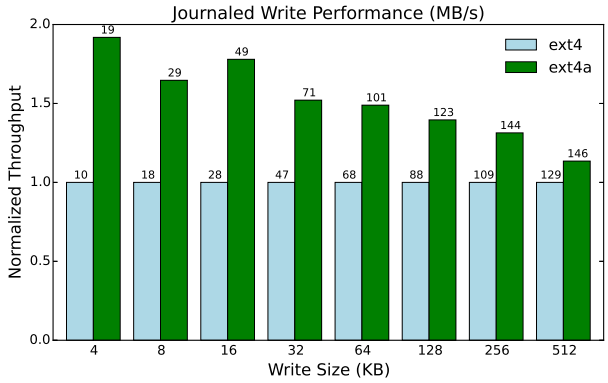

Figure 4: Data journaling write throughput with ANViL-optimized ext4a compared to unmodified ext4. Each bar is labeled with absolute write bandwidth.

nature of single-write journaling also obviates the need for writing the start and commit blocks of each journal transaction; for small transactions the savings from this are proportionally larger. At larger write sizes, the reason that the performance gain is less than the doubling that might be expected (due to halving the amount of data written) is that despite consisting purely of synchronous file writes, the workload is actually insufficiently IO-bound. The raw performance of the storage device is high enough that CPU activity in the file system consumes approximately 50% of the workload's execution time; jbd2's `kjournald` thread (which performs all journal writes) is incapable of keeping the device utilized, and its single-threadedness means that adding additional userspace IO threads to the workload does little to increase device IO bandwidth utilization. Adding a second thread to the 512KB write workload increases throughput from 132 MB/s to 140 MB/s; four threads actually *decreases* throughput to 128 MB/s.

The mechanism underlying single-write journaling could be more generally applied to most forms of write-ahead logging, such as that employed by relational database management systems [21].

6 Conclusions

The above case studies show that with a simple but powerful remapping mechanism, a single log structured storage layer can provide upstream software with both high performance and a flexible storage substrate.

Virtualization is an integral part of modern systems, and with the advent of flash it has become important to consider storage virtualization beyond volume management in order to uncover the true potential of the technology. In this paper we have proposed advanced storage virtualization with a set of interfaces giving applications fine-grained control over storage address remapping. Their implementation is a natural extension of common mechanisms present in log-structured datastores such as FTLs, and we demonstrated, with a set of practical case studies with our ANViL prototype, the utility and generality of this interface. Our work to date shows that the proposed interfaces have enough flexibility to provide a great deal of added utility to applications while remaining relatively simple to integrate.

7 Acknowledgments

We thank the anonymous reviewers, Tyler Harter, and our shepherd Jiri Schindler for their feedback, the rest of the ANViL development team, and Vijay Chidambaram for his assistance with our modifications to ext4.

References

[1] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, Washington, March 2008.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Pacific Grove, California, October 1991.

[3] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Ottawa Linux Symposium*, 2009.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.80 edition, May 2014.

[5] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.

[6] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *ACM Transactions on Storage (TOS)*, 4(4), February 2009.

[7] Chris Mason. Btrfs Design. `http://oss.oracle.com/projects/btrfs/dist/documentation/btrfs-design.html`, 2011.

[8] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 177–190, Monterey, California, June 2002.

[9] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[10] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. FlexVol: Flexible, Efficient File Volume Virtualization in WAFL. In *Proceedings of the USENIX Annual Technical Conference (USENIX '08)*, Boston, Massachusetts, June 2008.

[11] Gregory R. Ganger. Blurring the Line Between OSes and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[12] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, pages 229–240, Washington, DC, March 2009.

[13] Red Hat. LVM2 Resource Page. `http://www.sourceware.org/lvm2/`.

[14] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[15] Nikolaus Jeremic, Gero Mühl, Anselm Busse, and Jan Richling. Enabling TRIM Support in SSD RAIDs. Technical report, Technical Report Informatik Preprint CS-05-11, Department of Computer Science, University of Rostock, 2011.

[16] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.

[17] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, Massachusetts, October 1996.

[18] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan.

NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association.

[19] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[20] David Nellans, Michael Zappe, Jens Axboe, and David Flynn. ptrim() + exists(): Exposing New FTL Primitives to Applications. In *Proceedings of the Non-Volatile Memory Workshop*, NVMW '11, 2011.

[21] Xiangyong Ouyang, David W. Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *HPCA*, pages 301–311. IEEE Computer Society, 2011.

[22] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 31–39, Palo Alto, California, 1991.

[23] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[24] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, New York, NY, USA, 2012. ACM.

[25] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a Flash with ioSnap. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[26] Kyoungmoon Sun, Seungjae Baek, Jongmoo Choi, Donghee Lee, Sam H. Noh, and Sang Lyul Min. LTFTL: Lightweight Time-shift Flash Translation Layer for Flash Memory Based Embedded Storage. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, 2008.

[27] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[28] Veritas. Features of VERITAS Volume Manager for Unix and VERITAS File System. `http://www.veritas.com/us/products/volumemanager/whitepaper-02.html`, July 2005.

[29] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[30] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. HEC: Improving Endurance of High Performance Flash-based Cache Devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, 2013.