# Removing the Costs and Retaining the Benefits of Flash-Based SSD Virtualization with FSDV

Yiying Zhang
University of California, San Diego
yiyingzhang@cs.ucsd.edu

Andrea C. Arpaci-Dusseau
University of Wisconsin-Madison
dusseau@cs.wisc.edu

Remzi H. Arpaci-Dusseau
University of Wisconsin-Madison
remzi@cs.wisc.edu

***Abstract*—We present the design, implementation, and evaluation of the File System De-Virtualizer (*FSDV*), a system that dynamically removes a layer of indirection common in modern storage stacks and decreases indirection space and performance costs. FSDV is a flexible, light-weight tool that de-virtualizes data by changing file system pointers to use device physical addresses. When FSDV is not running, the file system and the device both maintain their virtualization layers and perform normal I/O operations. We implement FSDV with ext3 and an emulated flash-based SSD. Our evaluation results show that FSDV can significantly reduce indirection mapping table space in a dynamic way while preserving the foreground I/O performance. We also demonstrate that FSDV only requires small changes to existing storage systems.**

## I. Introduction

New generations of storage devices often virtualize their storage media to provide the conventional block I/O interfaces and to hide their internal operations [9], [11]. For example, flash-based SSDs maintain a mapping table to map host OS logical block addresses to physical addresses in flash memory using *Flash Translation Layers (FTLs)* [5], [6], [7], [10], [13], [14].

Multiple layers of virtualization can result in *excess* virtualization [4], [20]. For example, running a file system on top of a virtualized device creates two levels of virtualization; a block is first mapped from a file offset to its logical address and then from the logical address to its physical address in the device.

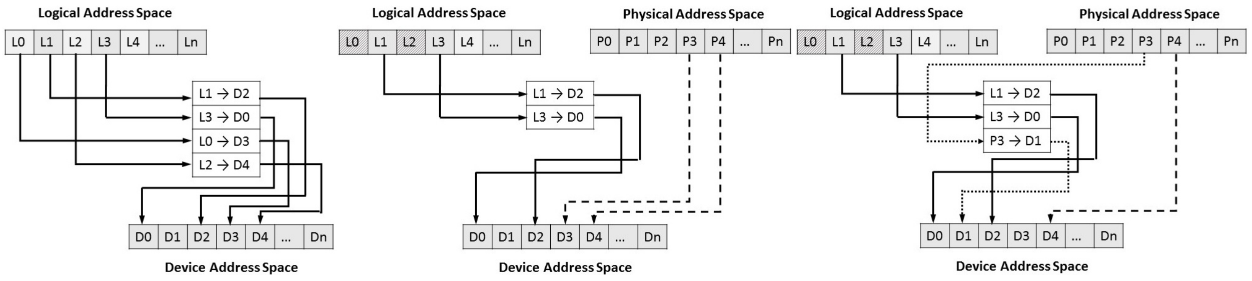
Certain virtualization layers are sometimes unnecessary; moreover, they can cause performance and memory overheads. The redundant level of virtualization in flash-based SSDs are usually maintained in DRAM, imposing performance, monetary, and energy costs. Such costs are of greater concern when SSD size grows or when SSDs are deployed in mobile devices. Even for SSDs that have enough DRAM for a big SSD mapping table [15], a smaller mapping table could still enable more DRAM space for performance optimization usages, such as a read cache or a write buffer. Various approaches [7], [13], [14] have been proposed to reduce the SSD mapping table size, but they usually come with a high performance cost or require fundamental changes to storage systems [12], [18], [19].

One way to remove excess virtualization is to co-design software and hardware layers to completely eliminate the redundant level of virtualization. Nameless writes [20] are one example of using new I/O interfaces to avoid excess virtualization in flash-based SSDs. Approaches like nameless writes remove excess virtualization for all I/Os and do not allow user control of when and how much virtualization is removed. Such approaches also require fundamental changes to the device I/O interface and are thus difficult to integrate into existing systems [16].

In this paper, we propose *de-virtualization*, a different approach to remove excess virtualization by letting existing virtualization layers create their indirection mappings and later removing these mappings. We can view *de-virtualization* as a method to reduce existing excess virtualization, as opposed to approaches that avoid the creation of excess virtualization. Suppose there are two layers of virtualization. The top layer creates a mapping $A \rightarrow B$ (*e.g.*, during file system allocation of logical resource) and the bottom layer creates a mapping $B \rightarrow C$ (*e.g.*, during device allocation of physical resource). A de-virtualizer collapses these mappings by updating the top layer mapping to $A \rightarrow C$ and by removing the bottom layer mapping $B \rightarrow C$.

To perform de-virtualization, we introduce the *File System De-Virtualizer* (*FSDV*), a tool that dynamically removes the virtualization costs in virtualized storage devices. The basic idea is simple: FSDV walks through file system structures and changes file system pointers from using logical addresses to using physical addresses. The device virtualization layer then removes the logical to physical address mappings.

To dynamically de-virtualize data, we separate blocks into different address spaces. Initially, the file system allocates logical addresses in the traditional way; all blocks are in the *logical address space* and the device uses a mapping table to map them to the *device address space*. FSDV then walks through the file system and de-virtualizes its data. Afterward, the de-virtualized contents are in the *physical address space* and corresponding mappings in the device are removed.



(a) **Original** (b) **After FSDV** (c) **After Device Remap**

Fig. 2: **FSDV Address Spaces.** *Different states of address spaces and address mappings in a storage system with FSDV. Device address space represents the actual physical addresses on the device. The file system sees both the logical and physical address spaces. The physical address space and the device address space have one-to-one mapping (that is, physical address P0 has the same value as device address P0).*

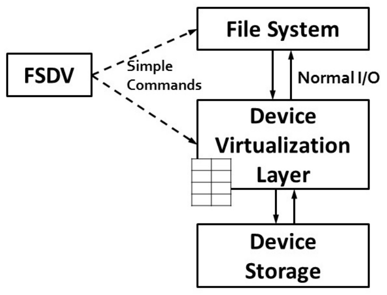


Fig. 1: **FSDV Architecture.** *The file system and the device perform I/Os and accumulates device mappings. FSDV issues simple commands to the file system and the device to perform de-virtualization and reduce the device mapping table size.*

The device then accumulates new mappings with new application writes until FSDV de-virtualizes them again.

FSDV offers two benefits over previous approaches that avoid excess virtualization.

First, FSDV can be invoked dynamically (*e.g.*, when the system is idle or when the system is under memory pressure). Such dynamic de-virtualization can remove excess virtualization both in device hardware where there is hard limit of memory space and in host software [8] where memory space is more elastic.

Second, by letting the virtualization mappings be created and removed later, FSDV preserves existing file system and device virtualization layers, requiring only small changes to them to assist de-virtualization. When FSDV is not running, the file system and the device perform I/O operations in a largely unchanged manner.

We implement FSDV as a user-level tool and modify the ext3 file system and an emulated flash-based SSD for it. Our evaluation results show that FSDV largely reduces the cost of device virtualization while preserving the foreground I/O performance. FSDV reduces device mapping table size by 75% to 96% with only 3% to 5% overhead in foreground I/O throughput, compared to an SSD FTL which has optimal performance but requires a large mapping table that does not fit in common device DRAMs. We also find that by placing most of the functionality in FSDV, only small changes are required in the current storage system.

## II. Design Overview

FSDV largely reduces device-level mapping table space in a dynamic way, while minimizing its performance overhead and its impact on current I/O systems. This section presents an overview of the FSDV system design and how its de-virtualization process works.

FSDV is a user-level tool that runs periodically or on demand to remove the costs of the indirection layer in a virtualized storage device. It works with running file systems by interacting with the file system and the virtualized device with simple commands. Figure 1 presents the architecture of a storage system with FSDV.

Since FSDV dynamically de-virtualizes data, a block can be in different states and have different types of addresses. We use three address spaces to represent different states of a block: the *logical address space* where newly allocated blocks exist, the *physical address space* where de-virtualized blocks sit, and the *device address space* which the device uses to store blocks physically. Figure 2 gives an example of FSDV address spaces and mappings in different states.

When FSDV is not running, the file system and the device perform their own I/O operations and maintain their virtualization layers. The file system allocates data in the *logical address space*. The device allocates *device addresses* and maintains a mapping table from logical to device addresses. Figure 2a represents the state of the storage system when FSDV has not been invoked. All file system addresses are in the logical address space, and the logical addresses L0, L1, L2, and L3 are mapped to the device addresses D3, D2, D4, and D0 through the device mapping table.

When the mapping table space pressure is high or when the system is idle, FSDV can be invoked to perform de-virtualization to reduce mapping space. FSDV de-virtualizes a block by changing the file system pointer that points to it (*i.e.*, the metadata) to use its device address. FSDV queries the device about the device address of a block and moves the block from the logical address space to the physical address space. This physical address is the same as the device address of the block. The device then deletes the corresponding

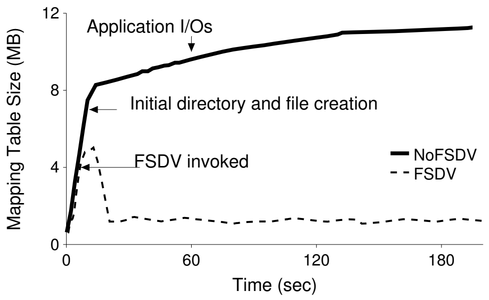


Fig. 3: **Mapping Table Space with and without FSDV.** *The mapping table space change over time running a FileServer workload without and with FSDV.*

mapping from the logical address to the device address. For future reads, the device checks if there is a mapping entry for the block. If there is, the device serves the reads after remapping. Otherwise, the device reads directly from the device address.

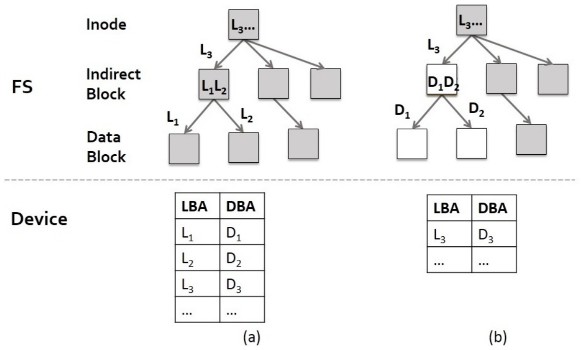
Figure 2b represents the state after FSDV de-virtualizes blocks L0 and L2 and moves them to the physical address space. These blocks now use physical addresses P3 and P4 which directly represent the device addresses D3 and D4. The device mappings from L0 and L2 to D3 and D4 are removed accordingly.

Apart from mappings created during application writes, device operations can also create or change address mappings. Flash-based SSDs perform garbage collection and wear leveling operations, both involving physical-block migration. When a directly mapped block is migrated to a new device address, the FTL adds a new mapping from its old device address to its current device address. FSDV also removes these mappings created by the device. Figure 2c represents the state after the device migrates a block from device address D3 to D1 (*e.g.*, during a wear leveling operation) and adds a mapping from physical address P3 to device address D1.

Figure 3 gives a concrete example of mapping table space change over time. In this example, a FileServer workload from the FileBench suite (F3 in Table I) is used. In the initial phase, the workload allocates all directories and files, causing the mapping table size to increase quickly. The workload then performs I/O operations (*e.g.*, appends and overwrites), causing a slow mapping table size increase. When FSDV runs, it reduces the mapping table size significantly and is able to keep the table size small throughout the workload.

## III. The FSDV Tool

We implement FSDV as a user-level tool that works with the file system and the device to perform de-virtualization. This section describes our implementation of FSDV, two optimizations FSDV uses for better performance, and how FSDV handles reliability issues.



| LBA | DBA |
|---|---|
| $L_1$ | $D_1$ |
| $L_2$ | $D_2$ |
| $L_3$ | $D_3$ |
| ... | ... |

| LBA | DBA |
|---|---|
| $L_3$ | $D_3$ |
| ... | ... |

Fig. 4: **FSDV Processing a File Tree.** *The left part of the graph (a) represents the state of a file in the file system and the device mapping table before FSDV runs. The right part (b) represents a state in the middle of a FSDV run. L1 and L2 have been devirtualized to D1 and D2. The indirect block containing these pointers has also been rewritten. The mappings from L1 to D1 and L2 to D2 in the device have been removed as well.*

### A. De-Virtualization Process

FSDV de-virtualizes a file system by processing one file at a time. For most file systems like ext2, ext3, and ext4, a file can be viewed as a tree structure with the inode of the file at the tree root, indirect blocks (or extent blocks) in the middle of the tree, and data blocks at the leaf level. FSDV de-virtualizes a file by walking through the file tree structure and processing metadata blocks from bottom up. Figure 4 gives an example of FSDV processing a file tree.

For each pointer in a metadata block, FSDV sends the address that the pointer uses (either logical or physical address) to the device and queries its current device address. If the device returns a device address, then FSDV updates the pointer to this address. After FSDV has processed all the pointers in the metadata block, it writes the updated metadata block to the device and informs the device to remove the corresponding mappings. FSDV uses the bottom-up fashion, because when FSDV processes an upper-level metadata block, all its children have already been processed. FSDV can update this metadata block with the final device addresses of all its children.

We do not de-virtualize inodes, directory blocks, or the metadata in the file system journal, since they only account for a small part of typical file systems [3].

### B. Optimization Techniques

We use two optimization techniques to reduce the overhead of FSDV. First, FSDV does not need to process files that have not been updated since the last run of FSDV. To further reduce the run time of FSDV, FSDV can choose not to process hot data, since they will soon be overwritten after FSDV de-virtualizes them. FSDV uses a hot file threshold, $Thresh_{HotFile}$, to only process files that are not accessed recently. For example, if we set $Thresh_{HotFile}$ to be 1/10 of the time window between two FSDV runs, the latter run will ignore the files that are most recently updated within 1/10 of this time window.

### C. Reliability Issues

Several reliability and consistency issues can occur during the de-virtualization process of FSDV. For example, the FSDV tool can crash before it completes its de-virtualization operations of a file, leaving the metadata of the file inconsistent.

We solve the reliability-related problems using the following techniques. First, we make sure that the device never deletes an old metadata block until the new version of it has been committed. When the new metadata block is written, its old version is invalidated at the same time. This operation is an overwrite and most current devices already invalidate old blocks atomically with overwrites. Second, FSDV logs the old addresses a metadata block points to before FSDV processes the metadata block. Doing so ensures that if FSDV crashes after writing the new metadata but before the device removes the old address mappings, the device can remove these mappings on recovery. Finally, we set a timeout in the file system to detect a dead or unresponsive FSDV tool.

## IV. Device Support

This section describes the changes in the device virtualization layer to support FSDV. We change a flash-based SSD emulator [20] to support FSDV. The emulator FTL uses log-structured allocation and page-level mapping. This FTL offers better performance than FTLs that maintain coarser-grained mappings, but it would require huge DRAM to store its mapping table without FSDV.

Most part of the emulated SSD and its FTL are not changed. The FTL still performs device address allocation for writes and maintains its mapping table. For reads, the FTL looks up its mapping table and either reads directly from the device address (if there is no corresponding mapping entry) or from the mapped device address.

We implement the FTL mapping table with a simple hash table that chains entries that fall into the same hash bucket. The key to the hash table is the block address that the file system sends to the device in an I/O request, the value in each hash table entry is the current device address of this block.

FSDV interacts with the device FTL using simple commands. When FSDV queries the device for the device address of a block, the FTL looks up its mapping and returns the mapped address or a no-mapping-found status to FSDV. After processing and writing the new metadata block, FSDV informs the device to remove the corresponding mapping entries. The device also stores FSDV operation logs for reliability issues.

One challenge specific to flash-based SSDs and other devices that migrate physical data is that new mappings are added for the migrated data. FSDV also removes these mappings caused by device data migration. A simple way to handle these mappings is to scan and de-virtualize the whole file system; these mappings will eventually be removed in this process. However, the performance cost of a whole-file-system scan is high, especially for large file systems. In order for FSDV to know what metadata points to the device-mapped data, we associate each block to the file to which it belongs. Specifically, we let the file system send the inode number together with a block write. The device records the inode number in the Out-Of-Band (OOB) area adjacent to the flash page that the device assigns the write. Notice that the device only needs to store the inode number in the OOB area and not in the device DRAM, thus it does not increase DRAM consumption. During page migration, the FTL records the files that contain migrated data. FSDV later queries the device about these device-migrated files.

Overall, the changes required in the device to support FSDV are small and do not affect the major functionality of the device. The interfaces between FSDV and the device are all simple commands initiated by FSDV. Sending inode numbers with writes is the only change we make to the I/O interfaces, and it is only required for devices that migrate physical data. These interface changes are easier to adopt into current device interfaces than previous solutions [16] that require fundamental interface changes (*e.g.*, calling from the device into the file system).

## V. File System Support

This section describes our file system changes to support FSDV. Specifically, we port the ext3 file system to FSDV. Most of the file system functionality is unmodified. We make the following changes to ext3 to support FSDV.

First, we add a bit to distinguish logical addresses from physical ones and change the device size boundary check to accommodate physical addresses.

Next, we modify the way the file system tracks address spaces to support FSDV. Ext3 uses logical address bitmaps to track allocated logical addresses and a free space counter to track the total amount of allocated addresses. Moving blocks between the logical address space and the physical address space results in bitmap changes, but we do not change the free space counter. Doing so ensures that ext3 has the correct information about the actual amount of free space on the device.

For devices that migrate physical data, the file system sends the inode number to the device during a write, so that the device can let FSDV know what files include device-migrated data.

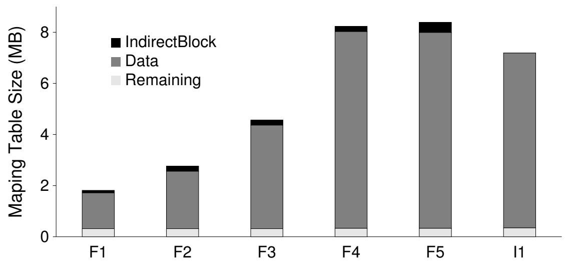


Fig. 5: **Mapping Table Space Reduction.** *The mapping table space reduction of indirect blocks and data blocks.*

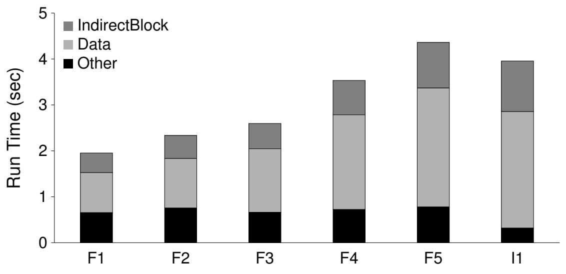


Fig. 6: **FSDV Run Time.** *The total run time of FSDV spent on de-virtualizing indirect blocks, data blocks, and the rest.*

| Workloads | Total Size | Files | File Size |
|---|---|---|---|
| F1 | 512 MB | 1000 | 512 KB |
| F2 | 1 GB | 2000 | 512 KB |
| F3 | 2 GB | 2000 | 1 MB |
| F4 | 4 GB | 2000 | 2 MB |
| F5 | 4 GB | 4000 | 1 MB |
| I1 | 3.6 GB | 3000 | 1.2 MB |

TABLE I: **Workloads Description** *The workload properties including the total workload size, the number of files, and the average file size. Workloads F1 to F5 represent different FileServer workloads from the FileBench suite. Workload I1 represents the file system image generated using Impressions.*

The file system works with FSDV to guarantee data consistency. When FSDV sends the request to process a block, the file system flushes it from the page cache. The file system also blocks I/Os to the block by stalling them until FSDV finishes its processing of this block.

Finally, for FSDV performance optimizations, the file system records the files that have changed since the last run of FSDV and their update time.

Overall, FSDV requires only small changes to a file system and does not affect its major functionalities.

## VI. Evaluation

This section presents our experimental evaluation of FSDV. We implement a user-level FSDV tool and change an emulated SSD device and the ext3 file system to support FSDV. The total lines of code change in the file system is 201 and is 423 in the device. We implement the FSDV tool using the *fsck* code base [1].

**Experimental environment:** All experiments were conducted on a 64-bit Linux 2.6.33 server that uses a 3.3 GHz Intel i5-2500K processor and 16 GB of RAM. The emulated SSD used in our experiments has 5 GB total size, 10 parallel flash planes, 4 KB flash pages, and 256 KB erase blocks. The flash page read and write operations take 25 and 200 $\mu s$. The erase operation takes 1.5 $ms$.

**Workloads:** We use the Impressions tool [2] to mimic typical file system images. For more controlled workloads, we use the FileServer macro-benchmark in the FileBench suite [17] with various numbers of directories and average file sizes. Table I summarizes the settings used in these workloads.

### A. Mapping Table Reduction

Reducing device virtualization costs is a major goal of FSDV. We first evaluate the mapping table space FSDV reduces. Figure 5 presents the amount of mapping table space reduction with different FileServer workloads and the Impressions file system image. We further break down the mapping table space reduction into the reduction due to data blocks, indirect blocks, and the amount of remaining mappings.

Overall, FSDV reduces the device mapping table size by 75% to 96%. As expected, most of the mapping table reduction is with data blocks, since typical file system images mostly consist of data blocks [3]. We also find that larger files result in bigger data block and indirect block mapping table reduction. The Impressions workload has less indirect block reduction as compared to the FileServer workloads, since it has smaller file sizes. Finally, there is a small part of mappings that FSDV does not remove. These remaining mappings are for inodes blocks, directory blocks, and other global file system metadata.

### B. FSDV Performance

It is important for FSDV to have short run time and low impact on foreground I/Os. Figure 6 shows the time taken to run FSDV with the FileServer workloads and the Impressions file system image.

Overall, the run time of FSDV is small (from 2 to 5 seconds for 512 MB to 4 GB data). We further break down the run time into the time spent on processing mappings of data blocks, indirect blocks, and other time (*e.g.*, time spent on reading block group description blocks). Most of the FSDV time is spent on processing data and indirect blocks. This time increases with larger file size and larger file system size. On the other hand, processing data and indirect blocks also contribute to most of the mapping table size reduction.

We also evaluate the performance overhead of FSDV on normal I/Os using the FileServer workloads. FSDV has only a small overhead of 3% to 5% on foreground I/O throughput. We compare FSDV to a page-mapped FTL which has optimal performance but requires a large mapping table that does not fit in common device

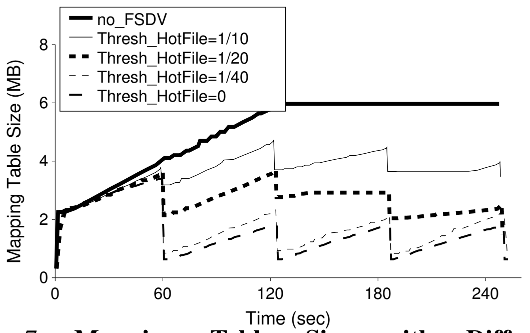


Fig. 7: **Mapping Table Size with Different $Thresh_{HotFile}$**

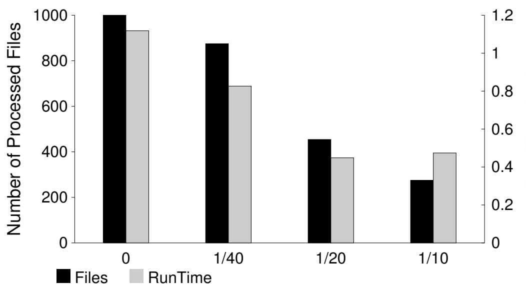


Fig. 8: **Effect of Different $Thresh_{HotFile}$**

DRAMs. When FSDV is processing a block, it blocks foreground I/O to this block until the prcocecessing is finished. We measure such blocking time to be $5ms$ on average. Such blocking only happens when a foreground I/O to a block is issued when FSDV is processing it, and I/Os to other blocks are not blocked.

### *C. FSDV Optimizations*

Finally, we measure how effective the FSDV optimization techniques are. We run the F2 workload and invoke the FSDV tool periodically (once a minute) to evaluate the effect of FSDV optimization policies.

Figure 7 presents the mapping table size change over time without FSDV and with FSDV using different hot data thresholds ($Thresh_{HotFile}$). $Thresh_{HotFile}$ is set so that the files that are most recently updated within 1/10, 1/20, and 1/40 of the time window between two FSDV runs are not processed. $Thresh_{HotFile}$ 0 represents that FSDV processes all modified files. As expected, as $Thresh_{HotFile}$ decreases, FSDV processes more files and reduces more mapping table space. FSDV with $Thresh_{HotFile}$ 0 reduces most mapping space.

Figure 8 shows the run time and number of processed inode with different $Thresh_{HotFile}$. A lower $Thresh_{HotFile}$ results in more files processed but longer FSDV run time.

## VII. Related Work

Most current flash-based SSDs use a coarser granularity of address mapping for most of the flash memory region [13], [14], or use fine granularity for all the flash memory and cache recently used mappings in RAM [7]. These methods reduce the SSD mapping table size to some extent at the cost of sacrificing SSD performance (*e.g.*, during garbage collection or when the working set size is big). FSDV reduces more SSD mapping table space without sacrificing foreground I/O performance.

DFS [8] is a system that moves the SSD virtualization layer and the mapping tables from SSDs to a software layer in the host memory. With this approach, the cost of virtualization within the device is removed, but the virtualization cost still exists in the host. FSDV is orthogonal to this approach. It can reduce the SSD virtualization costs whether the virtualization layer is in the hardware or in the host software.

Another approach to reduce the virtualization costs in SSDs is to use flash-oriented file systems [18], [19] that manages flash directly without an SSD FTL. These solutions sacrifice portability and flexibility, and introduce performance overhead and security vulnerabilities. For example, during garbage collection and wear leveling operations, data is moved back and forth between the device and the host through I/O buses that are slower than SSD-internal buses. Moreover, a bug or a security breach in the file system could potentially wear out an SSD fast. FSDV still lets SSDs manage flash memory and thus avoids such performance overhead and security vulnerabilities.

Most related to this work is our own previous work that proposes a new interface called nameless writes [20]. With nameless writes, the file system does not perform allocation and sends only data to the device. The device then allocates a physical address and returns it to the file system for future reads. Nameless writes largely reduce both the memory space and performance costs of SSD virtualization. However, our study [16] showed that this approach requires fundamental changes in device interface, the OS, and the device firmware, making it difficult to integrate to current systems. Another problem with nameless writes is that they require all I/Os to be de-virtualized and thus are not suitable for systems where indirection only needs to be removed dynamically. FSDV requires much less change to file systems and devices. FSDV is also more dynamic than nameless writes, since it can be invoked at any time (*e.g.*, when the device is idle). Thus, FSDV can avoid affecting foreground I/Os.

## VIII. Conclusion

We present the FSDV tool which dynamically reduces the virtualization costs in flash-based SSDs. FSDV removes these costs by changing file system pointers to use device addresses. Our evaluation results demonstrate that FSDV can remove SSD virtualization costs significantly and dynamically.

## Acknowledgment

We thank the anonymous reviewers and our shepherd Jin-Soo Kim for their enormously valuable feedback and comments. This work was supported in part by NSF grants CNS-1421033, CNS-1319405, and CNS-1218405 as well as generous donations from Cisco, EMC, Facebook, Google, Huawei, IBM, Microsoft, NetApp, Samsung, Seagate, and VMWare. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

## References

[1] E2fsprogs: Ext2/3/4 Filesystem Utilities. http://e2fsprogs.sourceforge.net/.

[2] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, San Francisco, California, February 2009.

[3] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[4] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. HarEl, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[5] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. System Software for Flash Memory: A Survey. In *Proceedings of thei 5th International Conference on Embedded and Ubiquitous Computing (EUC '06)*, pages 394–404, August 2006.

[6] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys*, 37:138–163, June 2005.

[7] A. Gupta, Y. Kim, and B. 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.

[8] W. K. Josephson, L. A. Bongo, K. Li, and D. 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.

[9] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee. A Group-based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '07)*, October 2007.

[10] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A Superblock-Based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software (EMSOFT '08)*, Seoul, Korea, August 2006.

[11] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-Memory Based File System. In *Proceedings of the USENIX 1995 Winter Technical Conference*, New Orleans, Louisiana, January 1995.

[12] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST '15)*, Santa Clara, California, February 2015.

[13] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. *In Proceedings of the International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED2008)*, February 2008.

[14] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. *IEEE Transactions on Embedded Computing Systems*, 6, 2007.

[15] Samsung. Samsung SSD 840 Pro. http://www.samsung.com/uk/consumer/memory-cards-hdd-odd/ssd/840-pro.

[16] M. Saxena, Y. Zhang, M. M. Swift, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[17] Sun Microsystems. Solaris Internals: FileBench. http://www.solarisinternals.com/wiki/index.php/FileBench.

[18] D. Woodhouse. JFFS2: The Journalling Flash File System, Version 2, 2003. http://sources.redhat.com/jffs2/jffs2.

[19] YAFFS. YAFFS: A flash file system for embedded use, 2006. http://www.yaffs.net/.

[20] Yiying Zhang and Leo Prasath Arulraj and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.