A High Performance File System for Non-Volatile Main Memory

Jiaxin Ou Jiwu Shu * Youyou Lu
Department of Computer Science and Technology, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology
ojx11@mails.tsinghua.edu.cn, {shujw, luyouyou}@tsinghua.edu.cn

Abstract

Emerging *non-volatile main memories* (NVMMs) provide data persistence at the main memory level. To avoid the double-copy overheads among the user buffer, the OS page cache, and the storage layer, state-of-the-art NVMM-aware file systems bypass the OS page cache which directly copy data between the user buffer and the NVMM storage. However, one major drawback of existing NVMM technologies is the slow writes. As a result, such direct access for all file operations can lead to suboptimal system performance.

In this paper, we propose HiNFS, a high performance file system for non-volatile main memory. Specifically, HiNFS uses an *NVMM-aware Write Buffer* policy to buffer the lazy-persistent file writes in DRAM and persists them to NVMM lazily to hide the long write latency of NVMM. However, HiNFS performs direct access to NVMM for eager-persistent file writes, and directly reads file data from both DRAM and NVMM as they have similar read performance, in order to eliminate the double-copy overheads from the critical path. To ensure read consistency, HiNFS uses a combination of the *DRAM Block Index* and *Cacheline Bitmap* to track the latest data between DRAM and NVMM. Finally, HiNFS employs a *Buffer Benefit Model* to identify the eager-persistent file writes before issuing the write operations. Using software NVMM emulators, we evaluate HiNFS's performance with various workloads. Comparing with state-of-the-art NVMM-aware file systems - PMFS and EXT4-DAX, surprisingly, our results show that HiNFS improves the system throughput by up to 184% for filebench microbenchmarks and reduces the execution time by up to 64% for data-intensive traces and macro-benchmarks, demonstrating the benefits of hiding the long write latency of NVMM.

1. Introduction

Emerging fast, byte-addressable non-volatile memories (NVMs), such as phase change memory (PCM) [8, 17, 29], resistive RAM (ReRAM), and memristor [51], are promised to be employed to build fast, cheap, and persistent memory systems. Attaching NVMs directly to processors produces non-volatile main memories (NVMMs), exposing the performance, flexibility, and persistence of these memories to applications [52, 53]. Moreover, these devices are expected to become a common component of the memory/storage hierarchy for laptops, PCs, and servers in the near future [10, 13, 21, 24, 30, 40, 54].

Given the anticipated high performance characteristics of emerging NVMMs, recent research [6, 13, 18, 49] shows that the overheads from the generic block layer and copying data between the OS page cache and the NVMM storage significantly degrade the system performance. To avoid these overheads, state-of-the-art NVMM-aware file systems, such as BPFS [13], PMFS [18], EXT4-DAX [6, 7], etc., bypass the OS page cache and the generic block layer. Specifically, all of them directly copy data between the user buffer and the NVMM storage without going through the OS page cache, implying that all requests incur prompt access to NVMM.

Unfortunately, one major drawback of NVMM is the slow writes [10, 20, 46]. The asymmetric read-write performance of NVMM indicates that, while DRAM and NVMM have similar read performance, the write operations of existing NVMM technologies, such as PCM and ReRAM, incur longer latency and lower bandwidth compared to DRAM [44, 52]. Therefore, direct access to NVMM can lead to suboptimal system performance as it exposes the long write latency of NVMM to the critical path. Furthermore, our experiments of running existing NVMM-aware file systems on a simulated NVMM device show that the overhead from the direct write access can dominate the system performance degradation.

The relatively large write performance gap between DRAM and NVMM indicates that buffering writes in DRAM is important for improving the NVMM system performance, because (1) writes to the same block may be coalesced since many I/O workloads have access locality [35, 38, 42, 43], and (2) writes to files that are later deleted do not need to be performed. In addition, writes in file

[*] Corresponding author: Jiwu Shu (shujw@tsinghua.edu.cn).

EuroSys '16, April 18 - 21, 2016, London, United Kingdom
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2901318.2901324

systems typically involve a trade-off between performance and persistence, and applications usually have alternative approaches to persisting their data [19, 37].

However, simply using DRAM as a cache of NVMM is inefficient due to the double-copy overheads in the critical path among the user buffer, the DRAM cache, and the NVMM storage [6, 18]. On one hand, reading data to a block not present in the DRAM cache causes the double-copy overhead in the read path, because the operating system needs to first copy the data from the storage layer to the DRAM cache, and then copy it from the DRAM cache to the user buffer. On the other hand, synchronous writes or synchronization operations, such as `fsync`, also lead to the double-copy overheads in the write path. For instance, if an application issues a write operation to block *A* followed by a `fsync` operation to persist block *A*, it incurs double data copies for block *A*. (The operating system first copies it to the DRAM cache at the write operation, and then copies it to the storage layer at the `fsync` operation.) The double-copy overheads can substantially impact the system performance when the storage device is attached directly to the memory bus and can be accessed at memory speeds [6, 13, 18, 49].

To address these problems, we propose HiNFS, a high performance file system for non-volatile main memory. The **goal** of HiNFS is to hide the long write latency of NVMM whenever possible but without incurring extra overheads, such as the double-copy or software stack overheads, thereby improving the system performance. Specifically, HiNFS buffers the lazy-persistent file writes (i.e., write operations that are allowed to be persisted lazily by file systems) in DRAM temporarily to hide the long write latency of NVMM. To improve the fetch/writeback performance of a buffer block, HiNFS manages the DRAM buffer at a fine-grained granularity by leveraging the byte-addressable property of NVMM. In addition, HiNFS interacts between the DRAM buffer and the NVMM storage using a memory interface, rather than going through the generic block layer, in order to avoid the high software stack overhead. To eliminate the double-copy overheads from the critical path, HiNFS performs direct access to NVMM for the eager-persistent file writes (i.e., write operations that are required to be persisted immediately), and directly reads file data from both DRAM and NVMM as they have similar read performance. However, writing data to DRAM and NVMM alternatively imposes a challenge for ensuring read consistency. Meanwhile, it also requires the file system to identify the eager-persistent writes before issuing the write operations.

This paper makes four contributions:

- We reveal the problem of the direct access overheads by quantifying the copy overheads of state-of-the-art NVMM-aware file systems on a simulated NVMM device. Based on our experimental results, we find that the overhead from the direct write access dominates the system performance degradation in most cases.
- We propose an *NVMM-aware Write Buffer* policy to hide the long write latency of NVMM by buffering the lazy-persistent file writes in DRAM temporarily. To eliminate the double-copy overheads, we use direct access for file reads and eager-persistent file writes.
- We ensure read consistency by using a combination of the *DRAM Block Index* and *Cacheline Bitmap* to track the latest data between DRAM and NVMM. We also design a *Buffer Benefit Model* to identify the eager-persistent file writes before issuing the write operations.
- We implement HiNFS as a kernel module in Linux kernel 3.11.0 and evaluate it on software NVMM emulators using various workloads. Our evaluations show that, comparing with state-of-the-art NVMM-aware file systems - PMFS and EXT4-DAX, surprisingly, HiNFS significantly improves the performance, demonstrating the benefits of hiding the long write latency of NVMM. Moreover, HiNFS outperforms traditional EXT2/EXT4 file systems on a RAMDISK-like NVMM Block Device (NVMMBD) emulator, which use the OS page cache to manage the DRAM buffer, by up to an order of magnitude, suggesting that it is essential to eliminate the double-copy overheads as it can offset the benefits of the DRAM buffer.

The remainder of this paper is organized as follows. Section 2 discusses the problem in state-of-the-art NVMM-aware file systems and analyzes their direct access overheads. We present the design and implementation of HiNFS in Section 3 and Section 4, respectively. We then present the evaluation results of HiNFS in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2. Background and Motivation

2.1 Problem in NVMM-aware File Systems

State-of-the-art NVMM-aware file systems, like BPFS [13], SCMFS [49], PMFS [18], and EXT4-DAX [7], eliminate the OS page cache which access the byte-addressable NVMM storage device directly. As an example, a `write()` syscall copies the written data from the user buffer to the NVMM device directly without going through the OS page cache and the generic block layer.

While this approach avoids the double-copy overheads, direct access to NVMM also exposes its long write latency to the critical path, leading to suboptimal system performance. In addition, to ensure data persistence and consistency, file systems either employ a cache bypass write interface[1] or use a combination of the `clflush` and `mfence` instructions behind write operations to explicitly flush data

[1] Different from the DRAM buffer cache, the CPU cache is hardware controlled which is cumbersome for the file system to track the state of the written data. As a result, existing NVMM-aware file systems, such as PMFS, use a cache bypass interface (e.g., `copy_from_user_inatomic_nocache()`) to enforce that the written data becomes persistent before the associated file system metadata does, because they wouldn't be able to control the writeback from the processor caches to the NVMM storage without using an expensive *clflush* operation.

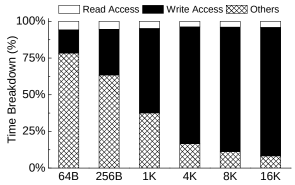


Figure 1. Time Breakdown of Running the Fio Benchmark on PMFS.

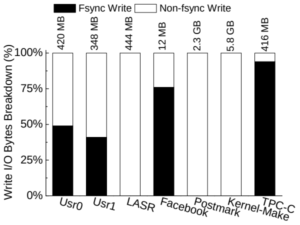


Figure 2. Percentage of Fsync Bytes with Different Workloads. *The value atop each bar shows total bytes written.*

from the CPU caches to the NVMM device to enforce ordering [18, 49], because existing cache hierarchies that were designed for volatile memory may reorder writes to improve the performance. For this reason, write latency is usually in the critical path, which cannot be tolerated by the CPU caches when NVMM is used as a persistent storage device rather than a volatile memory device [32, 33, 39]. Although BPFS's epoch-based caching architecture offers an elegant solution, it requires complex hardware modifications which involve non-trivial changes to cache and memory controllers [13]. In our work, we would therefore like to design an NVMM system without any hardware modifications.

In this paper, we mainly investigate how to design a high performance file system for NVMM by hiding the long write latency of NVMM but without introducing extra overheads. Our work is based on several assumptions shown as follows.

- First, we assume that NVMM devices are attached directly to the memory bus alongside DRAM, and the operating system is able to distinguish the NVMM devices from the DRAM ones [14].
- Second, we use the `clflush/mfence` instructions to enforce ordering and persistence, and assume that the `clflush` instruction guarantees that the flushing data actually reaches the persistent point (i.e., NVMM device). While Intel has proposed new instructions (`CLWB/CLFLUSHOPT/PCOMMIT`) to improve the cacheline flush performance and the CPU cache efficiency [15], these approaches are still unavailable in existing hardware. This paper, therefore, does not take them into consideration.
- Finally, HiNFS is mainly optimized for file-based I/O (i.e., *read* and *write* system calls) rather than memory-mapped I/O, as many important applications rely on traditional file I/O interfaces to access file data. However, HiNFS still supports direct access for memory-mapped I/O similar to existing NVMM-aware file systems (e.g., PMFS), which means that it does not sacrifice the performance of memory-mapped I/O. For the remainder of the paper, we refer to file write simply as write and file read simply as read.

2.2 The Direct Access Overheads of NVMM-aware File Systems

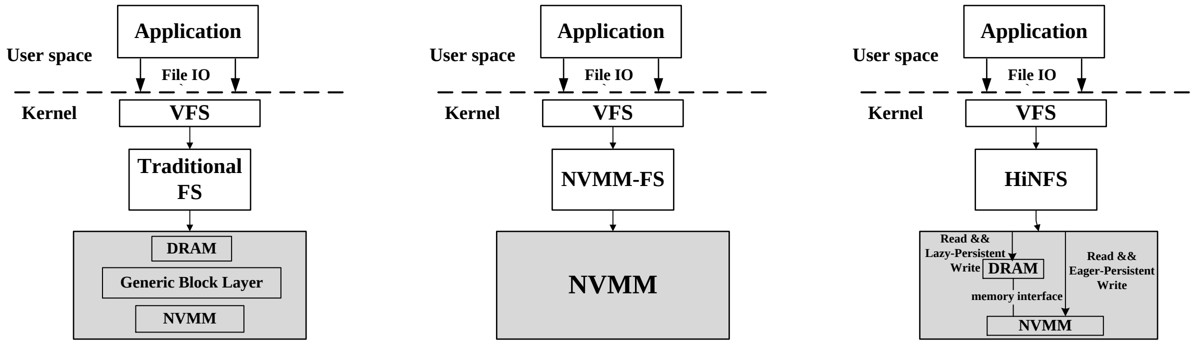
In this section, we will show that the overhead from the direct write access in existing NVMM-aware file systems can dominate the system performance degradation, and hence it is essential to reduce such overhead whenever possible.

To quantify the direct access overheads of existing NVMM-aware file systems, we run the `fio` [2] microbenchmark on PMFS [18][2], and use the `perf` profiling utility to obtain a breakdown of the time spent on running the benchmark. We use DRAM to emulate NVMM by introducing an extra configurable delay to NVMM writes to emulate NVMM's slower writes relative to DRAM. More technical details about our experimental setup are given in Section 5.1.

Each test is run for 60 seconds, and the results are shown in Figure 1. In all tests, we set the read/write ratio to 1:2 by default. In this figure, the time breakdown is organized into three categories: (1) *Read Access* refers to the overhead of copying data from the NVMM storage to the user buffer for read requests; (2) *Write Access* represents the overhead of copying data from the user buffer to the NVMM storage for write requests; and (3) *Others* is the overhead excluding the *Read Access* and *Write Access* overheads, which mainly includes overheads from user-kernel mode switch, file abstraction, etc. From this figure, we observe that the direct write access is a major source of overhead in most cases, and the proportion increases as the I/O size becomes larger. When the I/O size is no less than 4 KB, the direct write access overhead can account for over 80% of the total overheads, which substantially degrades the system performance. When the I/O size becomes smaller, such as 64 B, the direct write access overhead becomes relatively less significant than others, but still accounts for at least 16% of the total overheads.

While file systems can optimize the performance of the write operations that are not required to be persisted immediately, others, such as write operations enforced by synchronization operations, must enter the stable storage instantly to guarantee the data persistence required by user applications. Thus, their NVMM access overheads cannot be avoided. To see if there is enough room for optimizing those lazy-persistent writes, we perform another experiment that collects the `fsync` bytes across various workloads. Figure 2 shows the results of the percentage of `fsync` bytes with different workloads. More detailed descriptions of these workloads are given in Section 5. In this figure, we observe that different workloads have different persistence requirements. For example, TPC-C has over 90% `fsync` writes whereas

[2] We choose PMFS [18] as a case study of the baseline system because it along with EXT4-DAX [7] are the only available open-source NVMM-aware file systems at present. We also perform the same tests on EXT4-DAX, and it shows similar results. While BPFS [13] and SCMFS [49] are not open-source, we believe our observations also apply to them as they both perform direct access to NVMM.



(a) Traditional Block-based File System (b) NVMM-aware File System (c) HiNFS

Figure 3. Architecture Comparison of Different File Systems for NVMM.

LASR has no `fsync` writes. To conclude, a large number of applications have a significant portion of lazy-persistent writes, which are consistent with prior research results [19].

The above observations have interesting implications for the design of the file system for fast NVMM. On one hand, the revealed direct write access overhead strongly suggests that we need to reduce prompt writes to NVMM in order to improve the performance. On the other hand, we believe that an elegant design should be flexible. In other words, it should not improve the performance in some particular cases, while sacrificing the performance in other cases. For example, simply using DRAM as a cache of NVMM may improve the performance for workloads having many lazy-persistent writes, but this simple design will significantly degrade the system performance for workloads containing many eager-persistent writes due to the double-copy overheads.

3. HiNFS Design

In this section, we first describe the high-level system architecture comparison of existing file systems and HiNFS. We then present an *NVMM-aware Write Buffer* policy to reduce prompt writes to NVMM by buffering the lazy-persistent writes in DRAM temporarily. Finally, we discuss how to eliminate the double-copy overheads resulted from conventional buffer management.

3.1 System Architecture

Figure 3(a) shows the system architecture of traditional block-based file systems on a RAMDISK-like NVMM block device. This is the most straightforward way to use NVMM as a persistent storage in which legacy file systems, such as ext2/ext4, can directly work on NVMM without extra modifications by emulating it as a block device. In a block-based file system, each file I/O usually requires two data copies, one between the block device and the OS page cache through the generic block layer, and one between the OS page cache and the user buffer through the memory interface. However, it has been recently reported that the overheads from the double-copy and the generic block layer can significantly impact the NVMM system performance [6, 13, 18, 49]. As a result, state-of-the-art NVMM-aware file systems, such as BPFS [13], PMFS [18], etc., access the NVMM device directly as shown in Figure 3(b). In these NVMM-aware file systems, each file I/O requires only a single data copy, directly between the NVMM and the user buffer (a.k.a. direct access). Unfortunately, the major drawback of this approach is that it does not consider NVMM's relatively longer write latency compared to DRAM. Specifically, each write operation leads to prompt access to NVMM, which always expose the long write latency of NVMM to the critical path, leading to suboptimal system performance. Therefore, to get the best system performance, we propose another system architecture for the NVMM storage as shown in Figure 3(c). The design objectives of HiNFS are twofold:

(1) *Hiding the long write latency of NVMM behind the critical path.* HiNFS uses an `NVMM-aware Write Buffer` policy to buffer the lazy-persistent writes in DRAM temporarily. HiNFS design, including fine-grained buffer management and using a memory interface to interact between DRAM and NVMM, is optimized for the NVMM storage. (Section 3.2)

(2) *Eliminating the double-copy overheads.* Although buffering can help hide the long write latency of NVMM, it may introduce the double-copy overheads. For this reason, HiNFS optimizes read and eager-persistent write by avoiding unnecessary data copies. Read or eager-persistent write, in HiNFS, requires only a single data copy between DRAM/NVMM and the user buffer. (Section 3.3)

3.2 NVMM-aware Write Buffer Policy

To hide the relatively long write latency of NVMM behind the critical path, we propose an *NVMM-aware Write Buffer* policy to buffer the lazy-persistent writes in DRAM temporarily. Figure 4 shows an overview of HiNFS. When a write request is serviced, the *Eager-Persistent Write Checker* module would decide whether the current write operation is a lazy-persistent or eager-persistent write. We would like to discuss the approach of identifying the eager-persistent

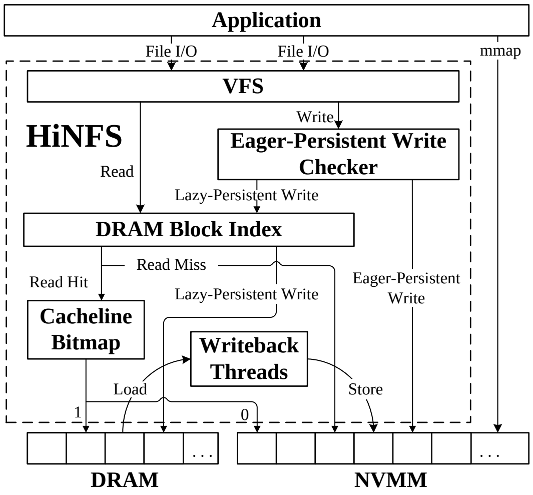


Figure 4. HiNFS Overview.

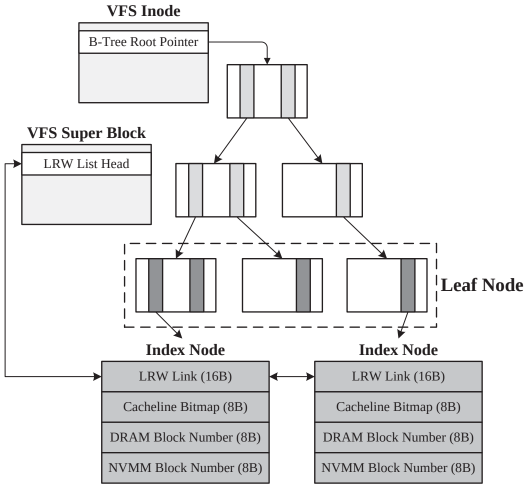


Figure 5. DRAM Block Index.

writes of HiNFS in Section 3.3.2. If it is a lazy-persistent write, HiNFS would like to issue this write request to the fast DRAM buffer, thereby eliminating the overhead of writing the NVMM. In HiNFS, allocation and replacement for the DRAM buffer are block-oriented. By default, the DRAM block size is 4 KB, which equals to the default block size of the NVMM storage. Currently, we use the LRW (Least Recently Written) policy, a variant of the LRU (Least Recently Used) algorithm, for the replacement of the DRAM buffer blocks due to the simplicity and efficiency of the LRU policy over decades [12, 16]. Specifically, we maintain the LRW list to keep track of the recency of write references of blocks in the DRAM buffer. That is, all the DRAM blocks are sorted by their last written time. When a DRAM block is written, it would be moved to the MRW (Most Recently Written) position. It is worth noting that this does not limit HiNFS of using other sophisticated buffer replacement policies, such as LFU (Least Frequently Used) [48], ARC (Adaptive Replacement Cache) [34], 2Q [23], etc. Different buffer replacement policies have different buffer write hit ratios, which decide how many writes can be coalesced before a buffer block is written back to the NVMM. However, these policies also increase the complexity of the buffer design, and the adding software overhead is non-trivial for the NVMM system. For this reason, we believe that the LRW-based policy is a good candidate to help us improve the performance, as a large majority of file system workloads show strong locality and high I/O skewness [35, 38, 42, 43]. We leave the research of using different buffer replacement policies in the future.

To efficiently index the DRAM blocks, HiNFS builds per-file B-tree in DRAM, one of the best options for indexing large amounts of possibly sparse data, to quickly perform search operations. Figure 5 shows the details of the *DRAM Block Index* in HiNFS. In the *DRAM Block Index*, the key of the index is the logic file offset which is aligned to the DRAM block size, while the value field (i.e., the Index Node shown in Figure 5) contains the physical DRAM block number and the corresponding physical NVMM block number. The NVMM block number in the value field enables the background writeback threads to flush the DRAM block to the corresponding NVMM block address. The root pointer of the B-tree is stored in the kernel's VFS inode structure. Moreover, all the index nodes are allocated from DRAM and linked to a global LRW list, the head of which is located in the kernel's VFS super block structure. Note that the *DRAM Block Index* structure is located in DRAM entirely, rather than in NVMM, in order to enable fast index operations.

We use the B-tree structure for the *DRAM Block Index*, because we would like to reuse the B-tree data structure from the PMFS [18] implementation as HiNFS is implemented based on it. While other index structures, such as hash table, can also be employed by HiNFS, the difference between B-tree and them may be only several bytes of DRAM access for each 4 KB block access, the overhead of which is far less than that of the data copy operations. Therefore, we believe that the data structure selection for the *DRAM Block Index* is not a critical issue, and thus there will be little performance difference between the index implementations of B-tree and other structures for HiNFS.

To ensure data persistence, HiNFS creates multiple independent kernel threads at mount time in order to flush the dirty DRAM blocks to the NVMM periodically in background. The flushed DRAM blocks can be released to secure free DRAM blocks for further buffering. There are two different cases of waking up the background writeback threads:

(1) The first case occurs when there are less than Low_f free DRAM blocks, where Low_f is a pre-defined threshold. In HiNFS, Low_f is set to 5% of the total DRAM blocks by default and is configurable.
(2) The second case is that the background thread wakes up every 5 seconds and periodically writes the updated data from the DRAM buffer to the NVMM storage.

When a writeback thread is woken up, it first selects the victim DRAM blocks from the LRW position of the LRW list. These victim DRAM blocks are then written back to the corresponding NVMM block addresses via a memory interface (e.g., memcpy()), rather than going through the generic block layer. After that, these DRAM blocks can be reclaimed for future write operations. The writeback thread reclaims several DRAM blocks at a time until the number of free DRAM blocks surpasses the $High_f$ threshold, which is set to 20% of the total DRAM blocks by default and can be adjusted. Then, the background writeback thread continues to scan the rest LRW list to write back any dirty DRAM blocks that were updated more than 30 seconds ago. In addition, HiNFS flushes all the DRAM blocks to the NVMM when unmounting the file system.

3.2.1 Fine-Grained Buffer Block Fetch and Writeback

Conventional buffer management in the OS page cache maintains the DRAM buffer space at the block granularity (i.e., 4 KB). This coarse-grained buffer management is inefficient for HiNFS. On one hand, an unaligned lazy-persistent write to a block not present in the DRAM buffer causes the operating system to synchronously fetch the block from the NVMM storage into the DRAM buffer before the write is applied. Such *fetch-before-write* requirement impacts the system performance, because the fetching process can block the writing process [9]. On the other hand, a whole buffer block would be flushed to storage even though only a few bytes of data are written to this block, causing a significant impact on the foreground application performance for two main reasons. First, when the DRAM buffer has no free blocks, the foreground lazy-persistent writes may stall until the background writeback threads reclaim enough free DRAM buffer space. Second, the background writeback threads can also compete the limited NVMM write bandwidth with the foreground eager-persistent writes. As a result, it is essential to improve the fetch/writeback performance of a buffer block in order to achieve higher system performance.

To address the above issue, we propose *Cacheline Level Fetch/Writeback (CLFW)*, which tracks the writes to the DRAM blocks on the basis of processor's cache lines. In *CLFW*, data is fetched from or flushed to NVMM in a fine-grained way rather than the block level. To do so, we use a *Cacheline Bitmap* (as shown in Figure 4) to track the state of each cacheline within a DRAM block. In this scheme, when a dirty DRAM block is selected for eviction, the writeback thread will check the *Cacheline Bitmap* of this block. Only if the P bit is 1 (i.e., the Pth cacheline is dirty), the cacheline should be written back to the NVMM. For an unaligned lazy-persistent write to a block not present in the DRAM buffer, we only need to fetch the corresponding cachelines instead of the whole block into the DRAM buffer. For example, for the baseline system with 4 KB DRAM block size and 64 B cacheline size, if a user writes to the 0∼112 B region of a block, traditional system needs to fetch the whole block (0∼4096 B) into the DRAM buffer, while *CLFW* only needs to fetch the second cacheline of this block (64∼128 B) into the DRAM buffer. In summary, *CLFW* significantly reduces the wasteful data-fetch and data-flush for workloads containing many small block-unaligned lazy-persistent writes, thereby improving the performance in these cases.

3.3 Elimination of the Double-Copy Overheads

As fast NVMM is attached directly to the processor's memory bus and can be accessed at memory speeds, extra data copies would be inefficient for NVMM systems which can substantially degrade their performance [6, 13, 18, 49]. As a result, it is essential to avoid such overheads whenever possible. To this end, we find two key reasons to cause the double-copy overheads resulted from conventional buffer management. This section describes them and discusses how we overcome them separately. It is worth noting that all the double-copy overheads, we pay attention to in this paper, mainly refer to those that occur in the critical I/O path, as they are the key factors of affecting the system performance.

3.3.1 Direct Read

In conventional buffer management, reading data to a block not present in the DRAM buffer causes the operating system to fetch the block into the DRAM buffer first, and then copy the data from the DRAM buffer to the user buffer, thereby leading to the double-copy overhead in the read path. To address this issue, HiNFS directly read data from both DRAM and NVMM to the user buffer, as they have similar read performance. Such direct copy policy is more efficient than conventional two-step copy policy as it eliminates unnecessary data copies.

However, writing data to DRAM and NVMM alternatively brings a new challenge to HiNFS to ensure read consistency. To find the up-to-date data for a read operation, HiNFS first checks the *DRAM Block Index* to see if the corresponding block is in DRAM. If not, it uses the file system block index to get the corresponding NVMM block address, and then performs this read operation to NVMM directly. Otherwise, it further checks the *Cacheline Bitmap* of the corresponding DRAM block to see which parts of data are in the DRAM block and which parts of data are in the NVMM block, and then copies the corresponding parts of data to the user buffer from both the DRAM and NVMM blocks on the basis of the *Cacheline Bitmap*. To minimize the number of memory copy (i.e., memcpy) operations, a single memcpy operation is used to copy the data in the consecutive cachelines, the corresponding bits of which in the *Cacheline Bitmap* have the same value, to the user buffer.

3.3.2 Direct Eager-Persistent Write

To further avoid the double-copy overhead in the write path, we issue the eager-persistent writes to NVMM directly rather than copying them to DRAM first. This is because writing them to DRAM not only causes unnecessary copy overheads, but also pollutes the buffer space which may evict other valuable buffer blocks. In HiNFS, the *eager-persistent writes* are defined as the following two cases:

(1) *Synchronous writes*. This happens when the file system is mounted with the sync option or the written file is opened with the O_SYNC flag.

(2) *Asynchronous writes followed by explicit synchronization operations*. We divide this scenario into two cases. If enough asynchronous writes can be coalesced before the arrival of the next explicit synchronization operation, in which case buffering is more efficient than direct access, we still regard them as the lazy-persistent writes. Otherwise, they are considered as the eager-persistent writes.

As HiNFS needs to choose either direct or buffer write mode for a write request, it is important to identify the eager-persistent writes before issuing the write operations. It is straightforward to identify case (1), because we can check the file system state by reading the file system super block and the file opening state by reading the file inode. However, identifying case (2) is particularly challenging, as we cannot know if the users would issue an explicit synchronization operation or how many writes can be coalesced before the arrival of the next synchronization operation in advance.

To overcome this challenge, we design a *Buffer Benefit Model* to decide if enough asynchronous writes can be coalesced before the arrival of the next synchronization operation. In this model, we identify case (2) using the most recent synchronization information, as it remains nearly the same within a short time period in most cases based on our observation from various workloads, which will be discussed later. Moreover, we identify case (2) on the basis of a data block. To this end, we add a new state, namely `Eager-Persistent`, to each data block. In HiNFS, each 4 KB data block needs only one bit to indicate its current state, implying that this overhead is very small and can be acceptable. Moreover, we store the block states in DRAM rather than in slow NVMM. If a data block is decided to be in the `Eager-Persistent` state, all the subsequent asynchronous writes to this data block are considered as the eager-persistent writes. Otherwise, they are considered as the lazy-persistent writes which are issued to the DRAM buffer first.

In the *Buffer Benefit Model*, the DRAM write latency is denoted as L_{dram}, and the NVMM write latency is expressed as L_{nvmm}. N_{cw} indicates the total number of cacheline writes between the previous and current synchronization operation of a data block, while N_{cf} is the total number of cacheline flushes from DRAM to NVMM of a data block which are performed by the current synchronization process rather than the background writeback threads. Then, buffering is more efficient than non-buffering for this block only if it satisfies the following inequality:

$$N_{cw} * L_{dram} + N_{cf} * L_{nvmm} < N_{cw} * L_{nvmm} \qquad (1)$$

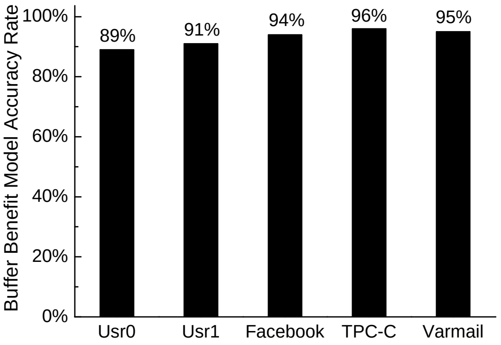


Figure 6. The Accuracy Rate of the Buffer Benefit Model Using the Most Recent Synchronization Information for Different Workloads.

This inequality means that the total execution time if writing to DRAM first is less than that if writing to NVMM directly for a data block. If a block satisfies this inequality, it will be set to the `Lazy-Persistent` state. Otherwise, it would be set to the `Eager-Persistent` state.

When the file system is mounted, all the existing or newly created data blocks are initialized to the `Lazy-Persistent` state before the arrival of their first synchronization operations. After that, we dynamically decide the data block states at each file operation. At each synchronization operation[3], we calculate to see if the related data blocks, which are required to be persisted to NVMM in the current synchronization operation, satisfy the above inequality. If a data block cannot satisfy this inequality, the state of this block is set to `Eager-Persistent`, which means that any subsequent asynchronous writes to this data block go directly to NVMM. Otherwise, we set the block state to `Lazy-Persistent`. Moreover, the state of a data block is switched from `Eager-Persistent` to `Lazy-Persistent` if it has not met a synchronization operation for a certain period of time, which is set to 5 seconds by default and can be adjusted. It is worth noting that we achieve this by deciding the data block state at the time of writing this block using the last synchronization time of its dependent file[4], rather than scanning all the data blocks at each fixed time, as it is lightweight to record the file synchronization time.

To get the value of N_{cf} of a buffer block, we maintain a *ghost buffer* to measure the total number of cacheline flushes from DRAM to NVMM of a buffer block during each synchronization operation. Ghost buffer assumes that every write goes to the DRAM buffer first but maintains only the buffer index metadata rather than the actual data. This leads

[3] In the current implementation, HiNFS only regards the fsync system call as the synchronization operation. While the msync operation is also a synchronization point in HiNFS, it is related to mmap I/O rather than file I/O.

[4] As the synchronization operation, such as fsync, is based on the file granularity, HiNFS adds a new field to the file metadata structure to record the last synchronization time of its related data blocks.

to low memory overhead which requires less than 1% of the total DRAM buffer space.

To see whether using the most recent synchronization information of a block to predict the state of its next synchronization operation is accurate, we measure the accuracy rate of our model using various workloads. The results are shown in Figure 6. We select five workloads that contain the synchronization operations and the descriptions of these workloads are shown in Section 5. Moreover, we measure it during the synchronization operations for each block. That is, if both the current and previous synchronization operation for a block satisfy or violate Inequality (1), it is accurate; Otherwise, it is inaccurate. In this figure, we can see that the accuracy ratio is close to 90% even in the worst case (i.e., Usr0). These results demonstrate that the synchronization information of a block remains nearly the same within a short time period, and thus our *Buffer Benefit Model* is effective in most cases.

To ensure consistency of the data blocks between DRAM and NVMM, when a write operation is identified as the eager-persistent write, if it is in case (1), we further check if the written block is present in the DRAM buffer before directly accessing the NVMM. If so, we still write the data to the corresponding DRAM block, and explicitly evict it from the DRAM buffer before returning to users. Fortunately, this case rarely happens, unless the file opening or file system state is altered frequently. If it is in case (2), we can always perform direct access to NVMM as long as the written block is in the `Eager-Persistent` state, because the latest data of this block is guaranteed to be persisted to NVMM since the last synchronization operation of this block.

4. Implementation

HiNFS is implemented based on the PMFS [18] file system in Linux kernel 3.11.0. HiNFS shares the file system data structures of PMFS but adds a new DRAM buffer layer and modifies the file I/O execution paths. In this section, we mainly discuss some details related to the implementation.

4.1 System Consistency

To maintain file system consistency, traditional journaling file systems provide multiple levels of consistency using different journaling modes (e.g., *writeback*, *ordered data*, or *journal data* mode). However, the current implementation of HiNFS only provides *ordered data* mode, which means that it only guarantees the data updates become persistent before the related metadata updates. To achieve this, HiNFS reuses the PMFS's journaling mechanism which only journals the file system metadata at the cacheline granularity [18]. Note that HiNFS does not buffer any file system metadata (e.g., inode or directory entry).

Different from the journaling mechanism in PMFS, HiNFS needs to keep the persistence ordering of the lazy-persistent writes. To do this, each lazy-persistent write operation will create a new transaction. The file system data blocks in the lazy-persistent write operation are buffered to DRAM first without being journaled to NVMM. These data blocks in DRAM are tracked using a transaction handler. In contrast, the file system metadata and its undo log entries are written to NVMM directly using the PMFS's logging scheme. To guarantee the ordered mode journaling invariant, HiNFS does not write the commit log entry to the NVMM log space until the related DRAM data blocks are persisted to NVMM. Additionally, HiNFS ensures ordering and persistence using the `clflush` and `mfence` instructions. Each writeback operation of a data block is followed by the `clflush/mfence` instructions so that the subsequent commit log entry will not be persisted to NVMM before this data block.

To be able to identify the partially written log entries during recovery, HiNFS includes a valid flag in each cacheline size log entry, and leverages the architectural guarantee in the processor caching hierarchy that writes to the same cacheline are never reordered, to indicate the integrity of a log entry, the approach of which is similar to that of PMFS [18]. To achieve this, the valid flag is written last when writing a log entry so that it will not become persistent before the data of this log entry.

4.2 Direct Memory-mapped I/O (mmap) Support

One of the key features of state-of-the-art NVMM-aware file systems (e.g., PMFS) is that they can support direct memory-mapped I/O, thus removing unnecessary data copies. HiNFS also supports this feature. When *mmap* a file, HiNFS first flushes all the dirty DRAM blocks of this file to NVMM, and then set the states of all its related data blocks to `Eager-Persistent`, which remain unchanged until this file is *munmapped*. Then, it directly maps the file data into the application's virtual address space so that users can access NVMM directly. However, the mmap write operations are not guaranteed to be persistent until the arrival of the next *msync* operation, as they are performed to the CPU caches first before being persisted to the NVMM storage.

5. Evaluation

In this section, we evaluate HiNFS to address the following questions:

(1) How does HiNFS perform against existing file systems ?
(2) What are the benefits of eliminating the double-copy overheads ?
(3) How is the scalability of HiNFS compared to other file systems ?
(4) How is HiNFS sensitive to the variation of the I/O size of the workload, the DRAM buffer size, and the NVMM write latency ?

We use the Filebench microbenchmark [3] to address (1), (2), (3), and (4). We use a variety of data-intensive traces and macrobenchmarks to further analyze question (1) and

Type	Workload	Description
Micro	Fileserver	Emulates a simple file server which consists of creates, deletes, appends, reads and writes.
	Webserver	Emulates a web server which performs file reads and log appends.
	Webproxy	Emulates a simple web proxy server with a mix of create-write-close, open-read-close, and delete operations, as well as log appends.
	Varmail	Emulates a mail server comprised of create-append-sync, read-append-sync, read and delete operations.
Macro	Postmark [26]	Measures the performance of a file system used for e-mail and web-based services.
	TPC-C	Emulates the activity of a wholesale supplier where a population of users execute transactions against a database, we execute DBT2 workload [1] on PostgreSQL 8.4.10 database system with 3 warehouses.
	Kernel-Grep	Searching for an absent pattern under the Linux 3.11.0 kernel source directory.
	Kernel-Make	Running `make` inside the Linux 3.11.0 kernel source tree.
Traces	Usr0	System call trace collected from research desktop by FIU [5].
	Usr1	System call trace collected from research desktop by FIU [5] at different time from Usr0.
	LASR [4]	System call trace collected from computers used for software development by CS researchers.
	Facebook	MobiBench [28] facebook system call trace.

Table 1. Workloads and Descriptions.

(2). Table 1 provides a description of all the workloads we evaluate.

5.1 Experimental Setup

NVMM Emulator

As real NVMM devices are not available for us yet, we develop a simple performance emulator based on the NVM-M emulator used in the Mnemosyne [46] project to evaluate HiNFS's performance. Similar to prior projects [20, 46, 47], our NVMM emulator introduces an extra latency for each NVMM store operation to emulate the slower writes of N-VMM relative to DRAM, while introducing no extra latency on the NVMM load operations. We have two considerations in assuming that NVMM and DRAM have the same read latency. First, we focus on the asymmetry of the read and write operations of NVMMs in HiNFS, and our evaluations focus on showing the benefits of the write performance rather than the read performance of HiNFS compared to state-of-the-art NVMM-aware file systems. Second, emulating the NVM-M read latency is complicated due to CPU features such as speculative execution, memory parallelism, prefetching, etc., which is hard to make it accurate [18].

NVMM Latency Emulation: Our emulator emulates N-VMM using DRAM. To account for NVMM's slower writes relative to DRAM, we introduce an extra configurable delay when writing to NVMM. We create delays using a software spin loop that uses the x86 RDTSCP instruction to read the processor timestamp counter and spins until the counter reaches the intended delay. Moreover, we add these delays after executing the *clflush* instruction. By default, we set the NVMM write latency to 200 ns [46].

NVMM Bandwidth Emulation: NVMM has significantly lower write bandwidth than DRAM [44, 52]. Assume that B_{NVMM} indicates NVMM's write bandwidth and L_{NVMM} is NVMM's write latency. Then, we emulate the NVMM write bandwidth by limiting the maximum number of the concurrent NVMM writing threads (denoted as N_w), where N_w equals to $(B_{NVMM}/(1/L_{NVMM}))$. An NVMM writing thread would be queued if the number of the current N-VMM writing threads reaches N_w, and the waiting queue will be woken up when one of the current NVMM writing threads completes. By default, the maximum sustained write bandwidth of NVMM is set to 1 GB/s, about 1/8 of the available DRAM bandwidth on the unmodified system [18].

CPU	Intel Xeon E5-2620, 2.1 GHz
CPU cores	12
Processor cache	384 KB 8-way L1, 1.5 MB 8-way L2, 15 MB 20-way L3
DRAM	16 GB
NVMM	Emulated with slowdown, the write latency is 200 ns, the write bandwidth is 1 GB/s
Operating system	RHEL 6.3, kernel version 3.11.0

Table 2. Server Configurations.

PMFS [18]	an NVMM-aware file system with direct access to NVMM
EXT4+DAX [7]	DAX is a kernel patch which supports EXT4 for bypassing the OS page cache
EXT2+NVMMBD	a traditional file system without journaling
EXT4+NVMMBD	a traditional journaling file system

Table 3. Existing File Systems for Comparison.

NVMMBD Emulator

To compare HiNFS against traditional block-based file systems, we construct another emulator, *NVMMBD*, to emulate the NVMM-based block device. We modify Linux's RAM disk module (`brd` device driver) and use the above NVMM performance model to emulate the NVMM latency and bandwidth.

We evaluate the performance of HiNFS against four existing file systems listed in Table 3. PMFS and EXT4+DAX are the two available open-source NVMM-aware file systems which access NVMM directly. EXT2/EXT4+NVMMBD are traditional block-based file systems, which are built on the NVMMBD block device emulator. Both of them are mounted with default settings. All the experiments are conducted on a x86 server with NVMM and NVMMBD emulators. The configurations of the server are listed in Table 2. For all the

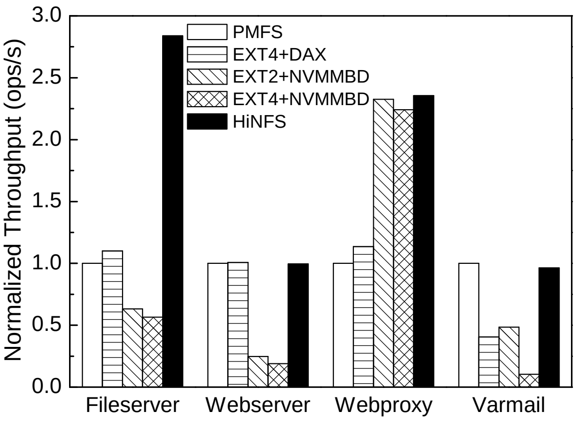


Figure 7. Overall Performance.

experiments, each data-point is calculated using the average of at least 5 executions.

5.2 Microbenchmarks

In this section, we run four types of workloads from the Filebench benchmark. Each workload is run for 60 seconds using 5 GB pre-allocated files after clearing the contents of the OS page cache. Unless otherwise specified, all the experiments are run with multiple threads and the mean I/O size is set to 1 MB by default[5]. Moreover, HiNFS is mounted with 2 GB DRAM buffer size, while EXT2/EXT4+NVMMBD are run with the available memory size being set to 8 GB (5 GB for storing the dataset on the NVMMBD and 3 GB for the system memory). We use the number of operations per second, which is reported by the Filebench benchmark, as the performance metric.

5.2.1 Overall Performance

We first evaluate the overall performance. Figure 7 shows the throughput normalized to that of PMFS. As shown in the figure, HiNFS achieves the best performance among the five file systems for all the evaluated workloads. Comparing HiNFS with PMFS and EXT4+DAX, HiNFS gains performance improvement by up to 184% (i.e., Fileserver), this is because almost all the writes in the Fileserver workload are lazy-persistent, and HiNFS asynchronously persists them to NVMM, thereby hiding the long write latency of NVMM behind the critical path. However, while EXT2+NVMMBD and EXT4+NVMMBD use the OS page cache to buffer the writes, we can see that only in one case (i.e., Webproxy) where they outperform PMFS and EXT4+DAX, due to the strong access locality exhibited in this workload. In contrast, they significantly underperform PMFS and EXT4+DAX in the rest cases, as the benefits of the DRAM buffer are offset by the overheads from the double-copy and the generic block layer. For the read-intensive workload, such as Webserver, EXT2 and EXT4 with NVMMBD show 3× lower performance than PMFS due to the unnecessary read copies between the DRAM buffer and the NVMM storage. Comparatively, we can see that HiNFS and PMFS achieve almost the same performance for the Webserver workload, demonstrating the benefits of eliminating the double-copy overheads.

The eager-persistent writes also causes the double-copy overheads. For the Varmail workload, we find that it contains a large part of synchronization operations. Moreover, all the writes in this workload are append operations, which cannot be coalesced in the DRAM buffer before the arrival of a synchronization operation. Therefore, we can see that HiNFS performs at par with PMFS due to that HiNFS bypasses the buffer for these eager-persistent writes. However, EXT4+DAX shows much lower performance than PMFS in this case. This is because the Varmail workload contains many metadata operations, and EXT4+DAX still follows the cache-oriented methods for them, while PMFS follows direct access for both data and metadata.

5.2.2 System Scalability

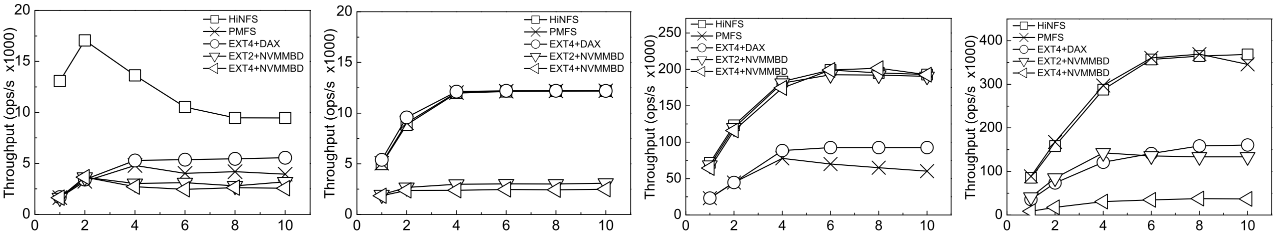
We also evaluate the system scalability of HiNFS and other file systems. Figure 8 shows the throughput for the four filebench workloads as we vary the number of threads in a single client process. Surprisingly, HiNFS achieves the best scalability for all the evaluated workloads. For the Fileserver workload, the performance of PMFS and EXT4+DAX are gradually limited by the NVMM write bandwidth when going from 1 to 10 threads, while the performance of EXT2/EXT4+NVMMBD is constrained by the overheads from the double-copy and the generic block layer. Therefore, HiNFS scales better than the other four file systems as it buffers and coalesces the writes before writing to NVMM. However, we find that HiNFS's throughput drops when the thread count goes from 2 to 8, this is because the buffer write hit ratio decreases as the number of threads increases. Fortunately, the performance becomes stable beyond 8 threads, and HiNFS still achieves nearly 1.5× higher performance than PMFS when going to 10 threads. In fact, the performance of HiNFS basically depends on the write locality of the workloads. With better write locality, such as Webproxy, we can see that HiNFS always scales well and its performance never decreases as the thread count increases.

For read-intensive workloads and workloads containing many eager-persistent writes, such as the Webserver and Varmail workloads, HiNFS achieves almost the same scalability with PMFS, both of which are much better than EXT2/EXT4+NVMMBD.

5.2.3 Sensitivity Analysis

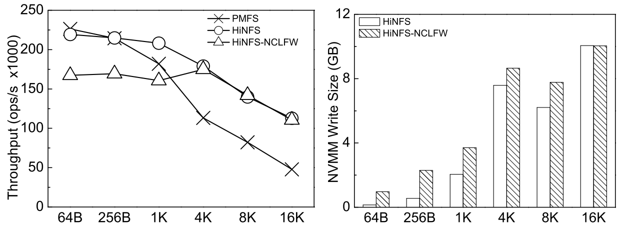
As the I/O size of the workload, the DRAM buffer size, and the NVMM write latency can affect the system perfor-

[5] We choose 1 MB as the mean I/O size for two reasons. First, this is the default configuration of the Filebench benchmark. Second, we adopt this configuration from the Aerie paper [47]. Sensitivity to different I/O sizes is also evaluated in Section 5.2.3 and Figure 9.



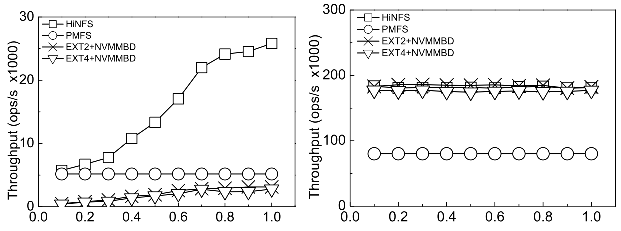
(a) # Threads - Fileserver (b) # Threads - Webserver (c) # Threads - Webproxy (d) # Threads - Varmail

Figure 8. Throughput (Operations per Second) for 1-10 Threads.



(a) I/O Size vs Throughput (b) I/O Size vs NVMM Write Size

Figure 9. Throughput (Operations per Second) and NVMM Write Size with Different I/O Sizes for Fileserver Workload.



(a) Buffer Size Ratio - Fileserver (b) Buffer Size Ratio - Webproxy

Figure 10. Throughput (Operations per second) as a Function of the DRAM Buffer Size.

mance, we measure their impacts on HiNFS's performance in this section.

Sensitivity to the I/O Size

The I/O size of the workload can affect the performance. Figure 9(a) presents the throughput performance with different I/O sizes for the Fileserver workload. For brevity, we omit the other three workloads. Webserver is a read-intensive workload while Varmail includes a large portion of eager-persistent writes, both of which cannot benefit from the DRAM buffer, thus HiNFS always yields performance similar to PMFS with different I/O sizes. We omit the Webproxy workload because it shows similar results with the Fileserver workload. To investigate the benefits of the *CLFW* scheme, we compare the performance and NVMM write sizes (i.e., total bytes that are written to NVMM) of HiNFS and HiNFS-NCLFW. HiNFS-NCLFW is a version of HiNFS that does not implement the *CLFW* scheme.

From Figure 9(a), we observe that HiNFS and HiNFS-NCLFW show a great difference in throughput when the I/O size is less than the DRAM block size (i.e., 4 KB), and HiNFS shows up to nearly 30% performance improvement over HiNFS-NCLFW. From Figure 9(b), we can see that HiNFS shows a remarkable drop in NVMM write size compared to HiNFS-NCLFW when the I/O size is less than the DRAM block size. The reason is that the background NVMM write traffic can also impact the system performance, because when the DRAM buffer is full, the normal writing threads may need to wait for the background writeback threads to clean out free buffer blocks. HiNFS significantly reduces the NVMM write traffic when the I/O size is unaligned to the DRAM block size, thereby improving the system performance. In contrast, the performance gap between them is bridged when the I/O size is larger than and aligned to the DRAM block size.

We also make another observation from Figure 9(a) that the performance gap between HiNFS and PMFS grows as the I/O size increases. For example, HiNFS outperforms PMFS by 58% when the I/O size is 4 KB, while improves the performance by 136% over PMFS when the I/O size is 16 KB. This is mainly due to that the copy overheads gradually become relatively more significant than other parts as the I/O size increases. When the I/O size is small (e.g., 64 B), the overheads from other parts, such as system call, user-kernel mode switch, etc., become dominant, thus hiding the benefits of reducing the copy overheads.

Sensitivity to the DRAM Buffer Size

The DRAM buffer size also has a strong impact on HiNFS's performance. Figure 10 shows the throughput performance as we vary the buffer size from 0.1 (10%) to 1.0 (100%) relative to the workload size. In Figure 10, we observe that the performance of HiNFS exhibits great improvement as the buffer size increases for the Fileserver workload, because more write operations will hit in the buffer when the buffer size increases. However, HiNFS's throughput remains nearly unchanged for the Webproxy workload when the buffer size ratio goes from 0.1 to 1.0 due to that the Webproxy workload has strong locality. Moreover, we find that the Webproxy workload exhibits many short-lived files, which would be deleted before the written data is flushed to NVMM. Therefore, the Webproxy workload is insensitive to the buffer size, and this is the only case where EXT2/EXT4+NVMMBD and HiNFS show nearly

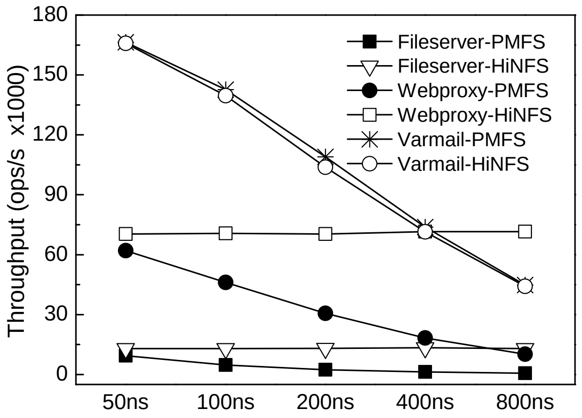


Figure 11. Throughput (Operations per Second) for Different NVMM Write Latencies.

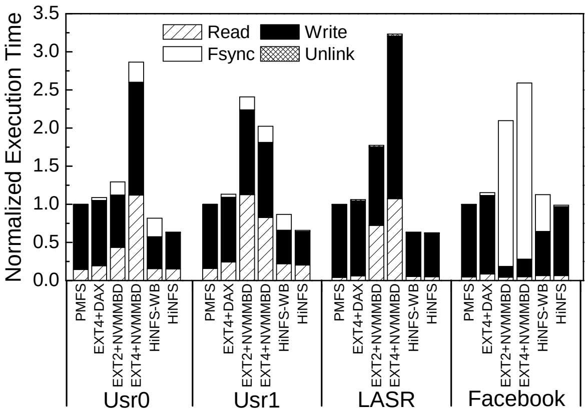


Figure 12. Breakdown of the Time Spent on Replaying Traces. *Normalized to PMFS's execution time.*

the same performance. For the Fileserver workload, EXT2/EXT4+NVMMBD have much lower performance than PMFS even when the buffer size ratio is 1.0, this is due to that the read copy overhead degrades the overall performance. Before the running the benchmark, we clear the contents of the OS page cache, so the read operations should first fetch the data from the NVMM storage into the DRAM buffer through the generic block layer. The overheads from the double-copy and the generic block layer significantly degrade their performance.

Sensitivity to the NVMM Write Latency

Another aspect that can affect the system performance is the NVMM write latency. Figure 11 shows the throughput performance when we vary the NVMM write latency from 50 ns to 800 ns using a single thread. In this figure, we can observe that the performance benefits of HiNFS become more obvious with longer NVMM write latency. For instance, HiNFS outperforms PMFS by only 53% when the NVMM write latency is 100 ns, but improves the performance by nearly 6× over PMFS when the NVMM write latency is 800 ns for the Webproxy workload. This is attributed to the fact that the system can get more performance benefits from the DRAM buffer as the speed gap between DRAM and NVMM increases. Even when the write latency of NVMM is close to that of DRAM (e.g., 50 ns), HiNFS still performs no worse than PMFS. This is because most of the write operations, in this case, will bypass the DRAM buffer with the *Buffer Benefit Model*, thereby eliminating the high double-copy overheads.

5.3 Data-Intensive Traces and Macrobenchmarks

To further investigate the performance of HiNFS and other file systems on real workloads, we replay a series of traces and run a set of macrobenchmarks on these file systems. In these experiments, the DRAM buffer size is set to 1/10 of the workload size by default. To demonstrate the benefits of bypassing the buffer for the eager-persistent writes, we also compare HiNFS with HiNFS-WB. HiNFS-WB refers to a system that simply uses DRAM as a write buffer of NVMM which is implemented by closing the function of the *Eager-Persistent Write Checker* in HiNFS. In HiNFS-WB, all the writes are buffered in DRAM first before being persisted to NVMM. For the traces replay, all the traces are system call level I/O traces, and we extract the *read*, *write*, *unlink*, and *fsync* operations from the traces, and replay them on the five different file systems. Moreover, we collect the time spent on these four different types of I/O operations respectively, and report a breakdown of the execution time in Figure 12. For the macrobenchmarks, we report the normalized runtime of all the benchmarks and show the results in Figure 13.

In Figure 12, we observe that HiNFS exhibits a reduction in execution time when comparing with PMFS by 37%, 35%, and 38% for the Usr0, Usr1, and LASR traces, respectively. As we can see in the figure, this is mainly attributed to the reduction of the write time of HiNFS compared to PMFS. HiNFS significantly outperforms PMFS except the Facebook trace, in which they yield similar performance. When we analyze this trace, we find that it contains a significant amount of `sync` operations. Moreover, we observe that HiNFS sets most of the related data blocks to the `Eager-Persistent` state with the *Buffer Benefit Model* in this case. Thus, it bypasses the DRAM buffer for most writes which are directly performed to NVMM, because the `sync` operations in this workload appear too frequent to coalesce enough writes in the DRAM buffer.

In Figure 13, HiNFS reduces the execution time of running the Postmark and Kernel-Make benchmarks by 60% and 64%, respectively, when comparing with PMFS. We find that the Postmark workload contains many short-lived files, where many lazy-persistent writes in this workload can benefit from the DRAM buffer for HiNFS, as writes to these files that are later deleted do not need to be performed to NVMM. In the rest two cases (i.e., TPC-C and Kernel-Grep), we can see that HiNFS and PMFS/(EXT4+DAX)

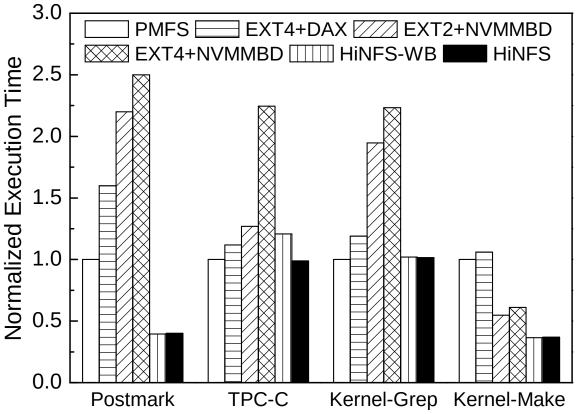


Figure 13. The Elapsed Time of Running Macrobenchmarks. *Normalized to PMFS's execution time.*

show nearly the same performance, all of which exhibit a remarkable drop in execution time when comparing with EXT2/EXT4+NVMMBD. We find that Kernel-Grep is a read-intensive workload while TPC-C contains many `sync` operations. In these cases, HiNFS bypasses the DRAM buffer for most I/O operations. This set of experiments also demonstrate the notable benefits of eliminating the double-copy overheads. In this figure, we also observe that EXT2+NVMMBD is much faster than EXT4+NVMMBD due to the absence of the journaling-related overheads.

Comparing HiNFS with HiNFS-WB in the two figures, we can see that HiNFS-WB increases the execution time over HiNFS by 28%, 32%, 14%, and 22% for the Usr0, Usr1, Facebook, and TPC-C workloads, respectively. As buffering the eager-persistent writes not only increases the system copy overheads, but also may evict other valuable buffer blocks which in turn decreases the ratio of write coalescing and increases the buffer writeback traffic, this performance improvement with HiNFS is due to that it effectively identifies the eager-persistent write operations, and then performs them to NVMM directly, demonstrating the benefits of the direct eager-persistent write policy of HiNFS. In other workloads, these two systems yield similar performance due to the absence of the synchronization operations in these workloads. However, because of the small mean I/O size (less than 1 KB) exhibited in the Facebook workload, we observe that it shows less difference between HiNFS and HiNFS-WB than that in the Usr0, Usr1, and TPC-C workloads.

6. Related Work

In this section, we discuss and draw connections to classes of previous work we feel most closely related.

Buffer Caching Algorithms. Most existing storage systems have been optimized under the assumption that the performance of the storage devices is several orders of magnitude lower than that of the main memory. For this reason, most existing works on buffer caching mainly concentrate on improving the cache hit ratio by keeping the blocks with most likely to be referenced again in the cache.

The LRU (Least Recently Used) [12, 16] algorithm achieves this by exploiting the recency of the last reference time, while LFU (Least Frequently Used) [48] considers the frequency of references. However, either of them considers only one factor while ignores others. As an improvement, ARC (Adaptive Replacement Cache) [34] adaptively considers both the recency and frequency of references. However, all of them focus on improving the cache hit ratio. HiNFS, in contrast, takes another unique perspective which aims to prevent the benefits of the buffer cache from being hidden by other extra overheads, such as the double-copy and software stack overheads, since these overheads can substantially impact the NVMM system performance. While HiNFS uses the LRW-based buffer replacement policy by default, other sophisticated buffer caching algorithms can also be seamlessly integrated into HiNFS to improve its performance.

Considering the relatively poor write performance of the flash memory, some cache studies [22, 25, 27] have investigated how to increase its write performance using a RAM write buffer. Flash-Aware Buffer (FAB) management [22] groups pages in the same flash block and evicts the group that has the largest number of pages when the buffer is full. However, FAB only considers the group size while overlooking the recency. To accommodate both the temporal locality and group size, the Cold and Largest Cluster (CLC) policy [25] combines the FAB and LRU algorithms. Both the FAB and CLC schemes aim to reduce the number of write and erase operations of the flash memory. In contrast, the Block Padding Least Recently Used (BPLRU) strategy [27] focuses on optimizing the random write performance of the flash memory by establishing a desirable write pattern with RAM buffering.

However, these flash-aware write buffer policies are not suitable for the NVMM storage due to the following reasons: First, they manage the buffer space at the page granularity rather than the cacheline level, which will generate a large amount of wasteful fetching and flushing data. Second, their designs are based on the unique characteristics of the flash storage, such as reducing the random write or erase operations, most of which are not applicable to the NVMM storage, as the random and sequential access of existing NVMM technologies are nearly identical and they have no erase operations. In contrast, the relatively high performance of existing NVMM technologies indicates that the system designers should carefully deal with the copy overheads among the user buffer, the file system buffer, and the NVMM storage [6, 18]. Therefore, HiNFS's write buffer policy is highly optimized for the NVMM storage, which focuses on reducing unnecessary data-fetch and data-flush by leveraging the unique characteristics of NVMM's byte addressability, and eliminating the double-copy overheads resulted from conventional buffer management from the critical path, thereby improving the NVMM system performance.

With high-speed storage medias, like PCM, have emerged recently, the performance gap between the main memory and the storage device drops dramatically. To figure out whether the buffer cache is still effective for them, Lee et al. [31] propose a new buffer cache management scheme appropriately designed for the system where the speed gap between cache and storage is narrow. To our knowledge, this is the only work that analyzes the effectiveness of the buffer cache under the fast NVM storage. Our work differs from them in the following aspects: First, their work is based on the assumption that NVM sits behind the I/O bus, while our work assumes that NVM is attached directly to the memory bus. Second, they aim to optimize the OS page cache and focus on improving the hit ratio of the buffer cache. HiNFS, in contrast, completely replaces the OS page cache with a new DRAM write buffer using a novel NVMM-aware buffer policy, which is cacheline-oriented and eliminates the software stack overhead of the block device layer altogether. Finally, their algorithm copies data to the buffer cache first for all file operations, which will incur the double-copy overheads. Based on our observation, these overheads are non-trivial for NVM storage system. HiNFS, therefore, buffers only the lazy-persistent writes, while uses direct access for reads and eager-persistent writes in order to eliminate the double-copy overheads from the critical path.

NVMM-aware File Systems. A number of file systems have been proposed to optimize for NVMM. BPFS [13] uses shadow paging techniques and 8-byte atomic updates to provide fast and consistent updates. However, BPFS doesn't support `mmap` and relies on a hardware approach (epochs) to support data persistence and ordering. While HiNFS is not optimized for `mmap` I/O, it still supports direct `mmap` access. PMFS [18] is a light-weight file system that is optimized for persistent memory, it avoids the block layer and eliminates the copy overheads by enabling applications to access persistent memory directly. Similar to PMFS's direct access policy, DAX [6, 7] is a kernel patch that can support traditional ext4 file system for bypassing the OS page cache and direct access to memory-like storage. However, all above three file systems do not take into account NVMM's slow write operations, and direct access to NVMM for all file operations leads to suboptimal system performance. In contrast, HiNFS buffers the lazy-persistent writes in the DRAM buffer, which can hide the long NVMM write latency, thereby improving the performance.

SCMFS [49] leverages the OS VMM to reduce the complexity of the file system. Aerie [47] provides flexible file system interfaces to reduce the hierarchical file system abstraction. Both SCMFS and Aerie focus on reducing the software overheads. However, based on our analysis, only in cases of metadata-intensive workloads or workloads with a small mean I/O size can the software overheads become relatively more significant than the storage access overheads. HiNFS, in contrast, focuses on reducing the storage access overheads (i.e., copy overheads) for data-intensive workloads.

Other NVMM Research. Since DRAM has faced with the scalability problem. Some research has proposed hybrid PCM/DRAM memory systems [40, 41]. Qureshi et al. [40] use a DRAM device as a cache of PCM in the hierarchy, while Ramos et al. [41] present a page placement policy on memory controller to implement PCM-DRAM hybrid memory systems. Our work is different from them in that we focus on the storage layer rather than the memory layer.

In addition, some research has proposed interesting programming models [11, 33, 36, 46], persistent data structures [45, 50], or new storage interfaces [53] for NVMM. However, legacy applications upon them require significant modifications. In contrast, we focus on the file system design for NVMM, and believe that the file system abstraction offers a good trade-off between supporting legacy applications and enabling optimized access to NVMM.

7. Conclusion

One major drawback of existing NVMM technologies is the slow writes. In this paper, we have presented HiNFS, a high performance file system for non-volatile main memory. HiNFS buffers the lazy-persistent writes in DRAM temporarily to hide the long write latency of NVMM, while eliminating the double-copy overheads resulted from conventional buffer management by using direct access for reads and eager-persistent writes. Extensive evaluations on software NVMM emulators demonstrate that HiNFS significantly outperforms both traditional block-based file systems and state-of-the-art NVMM-aware file systems.

Acknowledgments

We would like to thank our shepherd Ashvin Goel and the anonymous EuroSys reviewers for their insightful comments and suggestions, which have substantially improved the content and presentation of this paper. We are also thankful to Prof. Xubin He from the Virginia Commonwealth University for his valuable feedback. This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61433008), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), and the China Postdoctoral Science Foundation (Grant No. 2015M580098).

References

[1] Dbt2 test suite. `http://sourceforge.net/apps/mediawiki/osdldbt`.

[2] Flexible io (fio) tester. `http://freecode.com/projects/fio`.

[3] Filebench 1.4.9.1. http://sourceforge.net/projects/filebench/.

[4] Lasr system call io trace. http://iotta.snia.org/historical_section?tracetype_id=1.

[5] Fiu system call io trace. http://sylab-srv.cs.fiu.edu/dokuwiki/doku.php?id=projects:nbw:start.

[6] Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, 2014.

[7] Support ext4 on nv-dimms. http://lwn.net/Articles/588218/, 2014.

[8] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy. Phase change memory technology. *Journal of Vacuum Science & Technology B*, 28(2):223–262, 2010.

[9] D. Campello, H. Lopez, R. Koller, R. Rangaswami, and L. Useche. Non-blocking writes to files. In *13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 151–165, Santa Clara, CA, Feb. 2015.

[10] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, pages 21–31, 2011.

[11] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 105–118, 2011.

[12] E. G. Coffman and P. J. Denning. *Operating systems theory*, volume 973. 1973.

[13] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 133–146, 2009.

[14] I. Cooperation. Nvdimm namespace specification. http://pmem.io/documents/NVDIMM_Namespace_Spec.pdf, 2015.

[15] I. Cooperation. Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf, 2016.

[16] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[17] E. Doller. Phase change memory and its impacts on memory hierarchy. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf, 2009.

[18] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*, pages 15:1–15:15, 2014.

[19] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 71–83, 2011.

[20] J. Huang, K. Schwan, and M. K. Qureshi. Nvram-aware logging in transaction systems. *Proceedings of the VLDB Endowment*, 8(4):389–400, Dec. 2014.

[21] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. Childers. Improving write operations in mlc phase change memory. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA '12)*, pages 1–10, Feb 2012.

[22] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee. Fab: flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, May 2006.

[23] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*, pages 439–450, 1994.

[24] M. Jung, J. Shalf, and M. Kandemir. Design of a large-scale storage-class rram system. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*, pages 103–114, 2013.

[25] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha. Performance trade-offs in using nvram write buffer for flash memory-based storage devices. *IEEE Transactions on Computers (TC)*, 58 (6):744–758, June 2009.

[26] J. Katcher. Postmark: A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997.

[27] H. Kim and S. Ahn. Bplru: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, pages 16:1–16:14, 2008.

[28] E. LABORATORY. Mobibench benchmark tool. http://www.mobibench.co.kr/.

[29] B. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *Micro, IEEE*, 30(1):143–143, Jan 2010.

[30] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pages 2–13, 2009.

[31] E. Lee and H. Bahn. Caching strategies for high-performance storage media. *ACM Transactions on Storage (TOS)*, 10(3): 11:1–11:22, Aug. 2014.

[32] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *Proceeding of the 32nd IEEE International Conference on Computer Design (ICCD'14)*, pages 216–223, Oct 2014.

[33] Y. Lu, J. Shu, and L. Sun. Blurred persistence in transactional persistent memory. In *Proceeding of the 31st Symposium on Mass Storage Systems and Technologies (MSST'15)*, pages 1–13, May 2015.

[34] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 115–130, 2003.

[35] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. Sfs: random write considered harmful in solid state drives. In *10th USENIX Conference on File and Storage Technologies (FAST '12)*, pages 139–154, San Jose, CA, Feb. 2012.

[36] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS '13)*, pages 1:1–1:17, 2013.

[37] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 1–14, 2006.

[38] J. Ou, J. Shu, Y. Lu, L. Yi, and W. Wang. Edm: An endurance-aware data migration scheme for load balancing in ssd storage clusters. In *Proceedings of the 28th International Parallel and Distributed Processing Symposium (IPDPS'14)*, pages 787–796, May 2014.

[39] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*, pages 265–276, 2014.

[40] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pages 24–33, 2009.

[41] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*, pages 85–95, 2011.

[42] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '00)*, pages 41–54, 2000.

[43] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *USENIX Winter*, volume 93, pages 405–420, 1993.

[44] K. Suzuki and S. Swanson. The non-volatile memory technology database (nvmdb). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, May 2015.

[45] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*, pages 61–75, San Jose, CA, Feb. 2011.

[46] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pages 91–104, 2011.

[47] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*, pages 14:1–14:14, 2014.

[48] D. Willick, D. Eager, and R. Bunt. Disk cache replacement policies for network fileservers. In *Proceedings the 13th International Conference on Distributed Computing Systems (ICDCS '93)*, pages 2–11, May 1993.

[49] X. Wu and A. L. N. Reddy. Scmfs: A file system for storage class memory. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pages 39:1–39:11, 2011.

[50] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 167–181, Santa Clara, CA, Feb. 2015.

[51] J. J. Yang and R. S. Williams. Memristive devices in computing system: Promises and challenges. *Journal on Emerging Technologies in Computing Systems (JETC)*, 9(2):11:1–11:20, May 2013.

[52] Y. Zhang and S. Swanson. A study of application performance with non-volatile main memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST '15)*, pages 1–10, May 2015.

[53] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, pages 3–18, 2015.

[54] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pages 14–23, 2009.