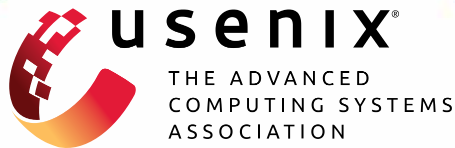


On Stacking a Persistent Memory File System on Legacy File Systems

Hobin Woo, *Samsung Electronics;* Daegyu Han, *Sungkyunkwan University;* Seungjoon Ha, *Samsung Electronics;* Sam H. Noh, *UNIST & Virginia Tech;* Beomseok Nam, *Sungkyunkwan University*

https://www.usenix.org/conference/fast23/presentation/woo

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Open access to the Proceedings of the 21st USENIX Conference on File and Storage Technologies is sponsored by

On Stacking a Persistent Memory File System on Legacy File Systems

Hobin Woo
Samsung Electronics

Daegyu Han
Sungkyunkwan University[*]

Seungjoon Ha
Samsung Electronics

Sam H. Noh
UNIST and Virginia Tech[†]

Beomseok Nam
Sungkyunkwan University

Abstract

In this work, we design and implement a Stackable Persistent memory File System (*SPFS*), which serves NVMM as a persistent writeback cache to NVMM-oblivious filesystems. SPFS can be stacked on a disk-optimized file system to improve I/O performance by absorbing frequent order-preserving small synchronous writes in NVMM while also exploiting the VFS cache of the underlying disk-optimized file system for non-synchronous writes. A stackable file system must be lightweight in that it manages only NVMM and not the disk or VFS cache. Therefore, SPFS manages all file system metadata including extents using simple but highly efficient dynamic hash tables. To manage extents using hash tables, we design a novel Extent Hashing algorithm that exhibits fast insertion as well as fast scan performance. Our performance study shows that SPFS effectively improves I/O performance of the lower file system by up to 9.9×.

1 Introduction

Non-volatile main memory (NVMM) has low access latency and byte-addressability similar to DRAM but ensures non-volatility of data similar to secondary storage. Intel's DC Persistent Memory module (DCPMM) is one of the first commercialized NVMM products, which provides exciting performance as storage class memory (SCM). Despite its shortcomings such as (i) latency higher than DRAM, (ii) bandwidth lower than DRAM, (iii) high sensitivity to NUMA effects, and (iv) a larger media access granularity (i.e., 256-byte XPLine), extensive research have been conducted to explore the desirable features of DCPMM, i.e., persistency with much lower latency than NVMe SSDs [7]. While the future of DCPMM is uncertain in short term [31] due to the recent Intel's unfortunate decision to shut down its Optane business, nevertheless, DCPMM has left various positive legacy, including NVMM-aware file systems [24, 37, 39] and key-value stores [11, 20, 21, 23, 34, 36]. Although such systems are still in their infancy, they have shown the potential to significantly outperform legacy systems and thus, other types of NVMM (e.g., MRAM and battery-backed DRAM) are likely to succeed DCPMM in the near future. However, the biggest weakness of current developments such as MRAM and battery-backed DRAM devices is their limited capacity. As such, for the immediate future, small NVMMs are expected to be used in conjunction with traditional storage devices. In this paper, we present a file system that can be deployed with only a relatively small amount of NVMM harnessing the benefits of NVMM, while, at the same time, continuing to make use of the underlying conventional file systems for block storage devices.

Previous studies have attempted to develop *monolithic file systems* that manage both NVMM and block device storage and that determine which device to service the read and write requests based on the I/O characteristics [24, 39]. However, managing multiple storage device types with a single, monolithic file system has its limitations. First, monolithic file systems for tiered storage devices, such as Ziggurat [39] and Strata [24], are hard to tailor for various combinations of multiple block device types. Second, developing a file system from scratch takes considerable time and effort to mature into a stable file system. Moreover, managing multiple tiered storage devices adds even more complexity. Third, from a deployment point of view, monolithic file systems cause a bit of inconvenience as they are oblivious of existing file systems; To deploy these systems in practice, a backup of the enormous number of files managed by legacy file systems must first be made, then the new NVMM and disk setting formatted, and then the backup copied back.

In this paper, we advocate a modular approach through the use of stackable file systems (aka overlay or union file systems) [9, 14, 29, 38]. Specifically, we present *SPFS (Stackable PM File System)*, a stackable file system that can be deployed with only a relatively small amount of NVMM, whose goal is to absorb frequent small synchronous writes required to maintain storage write order. For example, modern I/O stack enforces log entries and commit marks to be flushed to durable storage devices in serialization order such that recovery is pos-

[*]Department of Electrical and Computer Engineering
[†]This work was done at UNIST.

sible. For this, conventional file systems interleave small write requests with expensive `fsync()` system calls, which leads to performance degradation. The primary goal of SPFS is to let NVMM absorb such synchronous small writes and reduce the overhead of enforcing durability in block device file systems.

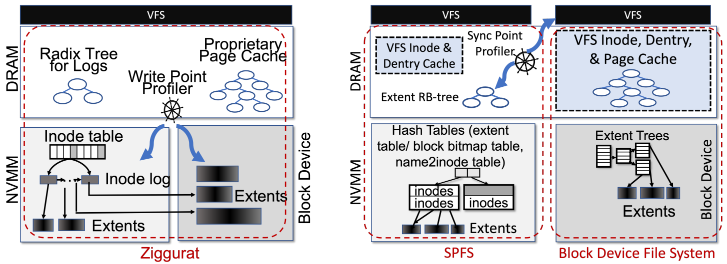
In addition, the NVMM-optimized ("upper") SPFS file system can be stacked on top of any other block device-optimized ("lower") file system *x* to provide a file system that is a union of both. Such a modular approach allows SPFS+*x* file system configurations that provide the best aspects of both NVMM and conventional devices as well as file systems for these devices. More specifically, aside from the performance benefits aforementioned, higher stability as well as flexibility can be attained. This is because we can exploit as the lower file system, any mature file system, e.g., EXT4, XFS, or F2FS, allowing delegation of large or asynchronous writes to the lower file system that can benefit from the highly efficient VFS cache. Furthermore, as SPFS is specifically designed and implemented to absorb frequent small synchronous writes in NVMM, its logic is simple and thus, easy to verify. Also, our modular approach is easier to deploy than monolithic file systems for tiered storage because SPFS can be stacked on any production file system on the fly. This makes deploying and taking advantage of NVMM simple.

In return for these advantages, stackable file systems may double the file system management overhead as it is layering two file systems. Therefore, a stackable file system must be lightweight. To this end, SPFS manages file system metadata in lightweight and efficient hash tables using a novel hashing algorithm that supports efficient lookup as well as scans.

The main contributions of this study are as follows.

- We design and implement SPFS, a stackable file system that allows any kernel file system *x* to reap the performance of NVMM while requiring no changes to *x*.
- SPFS, and its resulting SPFS+*x*, allows leveraging of the strengths of each storage device type, i.e., asynchronous writes of the VFS cache (DRAM), synchronous small writes of SPFS (NVMM), and various desirable features of disk-optimized file systems (SSDs).
- SPFS manages all file metadata in hash tables that ensure fast insertion and lookup. We also propose a novel *Extent Hashing* algorithm to hash key ranges and support extents in hash-based file mappings.
- Our performance study shows that SPFS+EXT4, SPFS+XFS, and SPFS+F2FS improves the performance of the lower file system by up to 9.9×.

The rest of this paper is organized as follows. In Section 2, we present the background and motivation. In Section 3, we present how SPFS profiles synchronous writes and steers them to NVMM. In Section 4, we present how SPFS manages file system metadata using hash tables. In Section 5, we evaluate the performance of SPFS. In Section 6, we conclude the paper.



(a) Monolithic Tiered File System (b) Stackable File System

Figure 1: Comparison of Ziggurat and SPFS

2 Background and Motivation

2.1 Stackable File System

File systems often make tradeoffs for a specific type of storage device [24]. For example, F2FS [25] is designed to accommodate the characteristics of NAND flash memory-based storage devices. Leveraging the hardware properties of each storage device has been studied for a long time. Such developments are expected to continue as evolution of storage devices (e.g., ultra low latency NVMes, Zoned Namespace SSDs, CXL devices, etc.) continues [5, 8, 15]. Therefore, we question whether it is desirable to have one file system that rules them all, and also question whether the effort of optimizing legacy block device-only file systems needs to be duplicated for monolithic file systems for tiered storage as well. For instance, Ziggurat does not use the well optimized VFS cache. Instead, it implements its own proprietary page cache as shown in Figure 1(a).

A stackable file system is a lightweight file system that runs on top of another file system. It is often used to change the behavior of the lower file system, e.g., encryption, access control, etc., without its own storage device (e.g., eCryptFS [16]), or to combine two mount points into one to provide a single file system image (e.g., UnionFS [14], OverlayFS [9], and AUFS [29]), such that an immutable Docker container image can be provided as a base and the upper level stackable and mutable file system can overlay new files or directories on top of the base. Wrapfs [38] is a small null-layer (i.e., template) stackable file system from which one could implement a new upper level file system. Wrapfs is an implementation of stackable *vnode* interface [30], which allows multiple vnodes to be chained for a single file. Using the vnode chain, a stackable filesystem can interact with the lower file system via VFS interfaces (e.g. `call_read_iter`) or direct operation calls (e.g. `inode_operations.fiemap`). With the vnode chain, stackable file systems can perform various functionalities, such as encrypting/decrypting files or making copies of data blocks in the upper level storage device to hide the data blocks in the lower level file system.

Consequently, if NVMM is used as an intermediate layer in the storage hierarchy, it is natural to design a *stackable* file system for NVMM that can be layered on a variety of

existing block device file systems instead of abandoning the legacy block device file systems that have been improved for decades. We believe there exists an unexplored opportunity of layering file systems optimized for each storage device as this allows one to easily get the most out of each storage device type. However, a stackable file system needs to be lightweight as layering two file systems may double the file system management overhead. As such, we aim to design and implement a lightweight hash-based stackable file system.

2.2 Steering Synchronous Writes

On spinning disk drives, the seek time may exceed the data transfer time if writes are small [18, 19]. Even on SSDs, it has been reported that small random writes fail to leverage the full device bandwidth because small random writes cause a large number of invalid pages to be scattered and valid pages are moved to different blocks via garbage collection. To mitigate such problems due to small writes, which we refer to as the *microwrite* problem, various block device file systems, including BetrFS [18, 19] and VT-tree [32], have been designed to absorb the small writes in a log-structured manner. Other remedies such as preallocation [26], defragmentation [22, 26], and block layer I/O scheduling techniques [35] have also been proposed. SPFS relies on these features of the conventional, lower file systems to address the microwrite problem. We believe handing the microwrite problems over to the DRAM cache in the lower file system is the most effective solution as it liberates SPFS to focus on the synchronization overhead (i.e., order-preserving writes), which cannot be resolved by the volatile DRAM cache.

Applications require synchronization mainly for two purposes - durability and *storage order* [35]. However, enforcing storage order by calling `fsync()` often results in frequent small synchronous writes, which leads to significant performance degradation because it prevents I/O parallelism [35]. SPFS steers this order-preserving synchronous writes to fast and durable NVMM while leveraging the VFS cache of the lower file system for *buffered* writes. As a stackable file system, SPFS does not duplicate the VFS cache to avoid the *double copy* problem [12].

Determining whether each write is synchronous or not is a hard problem. Ziggurat [39] and HiNFS [12] use DRAM as a write-back cache for buffered IO and determine if each write is synchronous or not based on the write size and fsync interval (Ziggurat) or based on the latency of each write (HiNFS), respectively. Both approaches are eager in detecting write types in that they determine the write type for each write.

In this work, we design and implement a lazy *Sync Point Profiler* to determine which blocks are to be placed in NVMM, or in DRAM or block device through the lower file system. By default, SPFS forwards incoming writes to the fast VFS cache first, then triggers block migration if certain conditions are met. This lazy approach benefits more from low DRAM latency, unlike the eager approaches of Zigurat and HiNFS.

2.3 Hash-based Global File Mapping

File mapping structures map logical offsets of a file to physical locations on the underlying device. In most traditional file systems, file mapping tables are tree-structured indexes such as extent trees and radix trees [28]. As the number and size of files increase, the size of the file mapping structures also increases. The resizing operation is particularly expensive in tree-based indexes because any update to internal tree nodes conflict with other concurrent operations that access different leaf nodes. To mitigate this problem, traditional file systems use *per-file* mapping structures to isolate concurrent accesses to different files and reduce lock contention.

In contrast to conventional wisdom, Neal et al. [28] recently show that as tree-based per-file mapping structures suffer from multiple levels of indirection and more memory references, a single hash table to manage global file mappings can be beneficial in NVMM. HashFS, the file system that they propose, requires a much smaller number of memory accesses than tree-based mappings, and as such, the performance of global hashing is shown to outperform per-file extent-trees and radix trees [28]. However, still, there is an unresolved limitation in hash-based global file mapping. That is, *block hashing* that is employed in HashFS is not suitable for sequential I/Os because block hashing does not manage extents. Extent-based file systems allow for files to be laid out contiguously on disk space, making sequential I/O fast. Extents also significantly reduce the amount of metadata by storing only two numbers, the first block number and the number of blocks covered by the extent. However, block hashing requires every block number in an extent to be canonically stored in its corresponding bucket, which slows down sequential I/Os. In this work, we develop a novel *Extent Hashing* algorithm to overcome these limitations, which we describe in Section 4.2.

3 Design of SPFS

SPFS consists of four key components that allow SPFS to be stacked with legacy file systems, as shown in Figure 1(b) - (i) the *Sync Point Profiler* that steers order-preserving small synchronous writes to NVMM, (ii) hash-based extent management (*extent table*), (iii) hash-based free space management (*block bitmap table*), and (iv) hash-based name resolution (*name2inode table*).

In this section, we concentrate on the Sync Point Profiler that determines which blocks are to be handled by SPFS and placed in NVMM or by the lower file system to be placed in conventional storage. The hash-based discussions are presented in Section 4.

3.1 File Block Placement Mode

Figure 2 shows the three file block placement modes supported in SPFS - *standalone*, *bypass*, and *stacked* modes. Note that SPFS places NVMM next to DRAM and disks, rather than in the middle of DRAM and block device hierarchy. SPFS is the upper file system, but only manages NVMM,

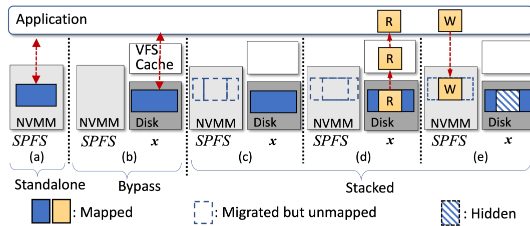
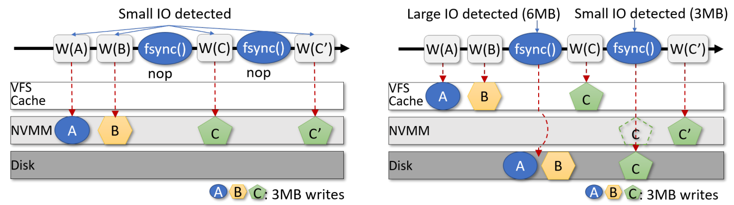


Figure 2: Three File Operation Modes



(a) Write Point Profiler (Ziggurat)

(b) Sync Point Profiler (SPFS)

Figure 3: Write Point Profiler vs. Sync Point Profiler

letting the faster DRAM VFS cache be managed by the lower file system.

Standalone mode: If a synchronous option (O_DIRECT, O_SYNC) is specified for the `open()` system call or if SPFS is used without a lower file system, all file blocks are placed in NVMM as shown in Figure 2(a).

System administrators can also force the use of NVMM on a per-directory basis via extended attribute or `ioctl`. That is, for example, if there is a directory that stores transactional log files or short-lived backup files, the administrator can specify the directory to be used in standalone mode.

Bypass mode: If the synchronous option is not specified for `open()`, NVMM is bypassed by default, placing the file in the lower file system as shown in Figure 2(b). By bypassing writes to the lower file system, read intensive workloads and non-synchronous writes can benefit from the fast VFS cache.

Stacked mode: Figures 2(c), (d), and (e) show the Stacked mode where both SPFS and the lower file system play roles as particular blocks of files are placed in either NVMM or conventional storage. If the *Sync Point Profiler*, to be described in Section 3.2, decides to place a block in NVMM, SPFS gets the file extent information using `fiemap ioctl` and prepares the file mapping in NVMM. In preparing the file mapping, SPFS does not yet physically migrate the extent as shown in Figure 2(c), where the dashed rectangle represents the mapping for the file to be migrated. Physical migration is delayed because the Sync Point Profiler makes the migration decision when `fsync()` is called, i.e., the dirty blocks have already been delegated to the lower file system, and they could have been persisted to disk via periodic write-back of the lower file system. Instead, the migration is deferred until the next write such that `read()` benefits from the low latency of the VFS page cache, as shown in Figure 2(d). Since there is no need to move a migration target block to NVMM unless the block is subsequently updated, physical migration is triggered by subsequent `write()` calls, avoiding the double copy problem [12]. As shown in Figure 2(e), SPFS checks the file mapping and writes blocks in NVMM if the file mapping indicates that the file is mapped in NVMM. By nature of stackable file systems, access to migrated blocks is serviced by the upper file system, that is, SPFS. Thus, the blocks in the lower file system become invisible to the user. When the blocks are actually migrated, the blocks in the lower file system are erased via `fallocate()`. Later, if the entire file is migrated to SPFS, the file is deleted from the lower file system.

3.2 Profiling Mechanism: Sync Point Profiler

Order-preserving small synchronous writes often lead to orders of magnitude IOPS degradation [35] because it serializes potentially parallel activities. Such order-preserving small synchronous writes need to be steered to fast NVMM rather than slow block devices. Therefore, we devise a *Sync Point Profiler* that monitors `fsync()` calls. Specifically, at an `fsync()` call, if the previous `fsync()` call on the same file is within a certain threshold and the amount of flushed data is small, we consider this to be an order-preserving small synchronous write. The rationale behind this is that if the interval is short, there is a high probability that the `fsync()` calls are made with intention to keep the *storage order* [35]. In contrast, if the interval is large, then even if the write size is small, it is unlikely that applications are flushing writes in continued sequence. Thus, these can be serviced by the slower lower file system without much performance degradation. To determine small, we do not take individual write sizes at the point of writes, but take the total number of bytes written to the lower file system at `fsync()`. The rationale behind this is that large writes can benefit from disk bandwidth, and synchronous writes to maintain storage order are usually small (e.g., 4 KB WAL frames in DBMS). The default values in our setting are 1 second for the interval and 4 MB for the size. We take 4 MB as this is the value used in Ziggurat's synchronicity predictor's policy [39], while 1 second was chosen as we observe the performance of SPFS is insensitive to the threshold time unless it is set too small. In Section 5, we quantify the performance effects of the profiler parameters.

Figure 3 highlights the key differences between the Ziggurat's synchronicity and write size predictor and the SPFS Sync Point Profiler. The example in Figure 3(a) where an application issues four 3 MB small writes (`A`, `B`, `C`, and `C'`) shows how Ziggurat makes its decision for each individual `write()` (thus, Write Pointer Profiler) and eagerly persists small writes according to its *fast-first* policy. Its write size predictor will detect the first two writes, `A` and `B`, as small and store them in NVMM. When `fsync()` is called, its synchronicity predictor will detect the total number of bytes is larger than 4 MB and treat the file as an asynchronous file.

Nevertheless, A and B have already been flushed to NVMM. The write size predictor also steers C into NVMM since it is small. The second fsync(), however, considers the file as a synchronous file as only 3 MB (C) was written. Consequently, the next write C′ will also be written in NVMM. In conclusion, we see that all writes are stored into NVMM. As we will show later in Section 5, Ziggurat's profiling method fails to leverage faster DRAM and shows similar performance as the NVMM-only file system NOVA because it aggressively steers most writes to NVMM.

SPFS, on the other hand, makes block placement decisions when fsync() is called. Using the same example as above, Figure 3(b) shows how differently the SPFS profiler services the writes. For A and B, they are initially written to the VFS page cache allowing them to make use of the DRAM. Upon the first fsync(), because the total write size is 6 MB, both A and B are written to the block device via the lower file system. Similarly, C is also written to the page cache. When the second fsync() is called, the lower file system flushes C from the cache to disk, but at the same time, SPFS detects small synchronous writes and migrates its block mapping, (not data blocks), to NVMM. When subsequent writes are requested to some of the blocks of C (C′), these writes are steered to NVMM and directly written onto.

3.2.1 Migration to Lower File system

Compared to Ziggurat, SPFS uses NVMM sparingly. However, when the NVMM space is running low, SPFS selects victim files and migrates them to the lower file system. Note that the primary goal of SPFS is not to cache frequently accessed files but to absorb order-preserving small synchronous writes. Therefore, even if NVMM has free space, SPFS migrates a file to the lower file system if its access pattern changes, e.g., if the access pattern is read intensive, *demoting* the file to the lower file system can benefit from the VFS page cache.

SPFS uses a metric called *Sync Factor* to determine which file's recent I/O pattern is well suited for the criteria of order-preserving small synchronous writes. The formula that calculates the Sync Factor (SF) at time t is given by

$$SF_t = \alpha \cdot weight(IO_type) + (1-\alpha) \cdot SF_{t-1}$$

where α is the attenuation factor ($0 < \alpha < 1$), i.e., the formula employs exponential moving average to attenuate the effect of old file accesses. $weight(IO_type)$ is a fixed positive value if the current I/O at time t satisfies the Sync Point Profiler's condition. Otherwise it is zero, i.e., if a file is read-intensive or updated in large units, its Sync Factor gradually decreases. Sync Factor is maintained per file and updated only upon an I/O request. Therefore, its computation overhead is negligible.

When the NVMM space is running low, SPFS migrates the files with low Sync Factor back to the lower file system in the background. Administrators can also set a hard limit on the Sync Factor so that files can be migrated back to the lower file system if their Sync Factors are lower than the hard limit, even if NVMM has free space.

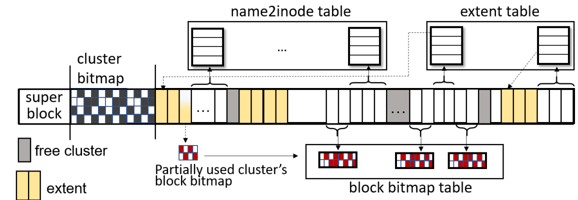


Figure 4: NVMM Space Layout for SPFS

4 Hash-based Block Management

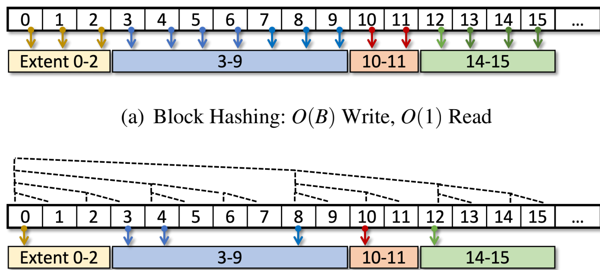
While SPFS is a stackable file system, it is also a standalone hash-based NVMM file system. As a file system, SPFS requires file system metadata to be managed persistently, and thus, metadata management overhead can be doubled. Since the target workload of SPFS is synchronous I/Os to a large number of small files, the conventional per-file mapping structures may waste storage space [28]. Therefore, similarly to HashFS [28], SPFS manages file block mapping information using global hash-based structures. Furthermore, SPFS reduces the size of the hash table by indexing extents, not blocks. However, to the best of our knowledge, no efficient means of hashing extents, i.e., range data, is known. To overcome this limitation, we propose a novel *Extent Hashing* algorithm.

In this section, we describe hash-based free space management (*block bitmap table*), hash-based extent management (*extent table*), and hash-based path-name resolution (*name2inode table*) in SPFS.

4.1 Free Space Management

Using extents, SPFS effectively reduces the aggregate size of file mapping metadata. The aggregate size of the file mapping structures is particularly important for a lightweight stackable file system because small mapping structures leave more room for file data blocks. SPFS employs dynamic hashing (in particular, CCEH [27]) to dynamically adjust the size of the multiple hash tables and efficiently manage the NVMM space. Specifically, if a hash collision cannot be avoided by linear probing or cuckoo hashing, SPFS dynamically allocates and assigns NVMM blocks to each hash table, namely, extent table, block bitmap table, and name2inode table.

SPFS manages data blocks at the granularity of 4 KB but metadata blocks at 256 bytes (also referred to as XPLine, the unit of physical access to DCPMM) by default. This is to reduce the waste of NVMM space as well as to avoid write amplification on hardware, which can be caused by read-modify-write operations. However, if we use small blocks, which is 1/16 of the traditional block size, SPFS needs to keep track of a 16× larger number of blocks, which leads to high metadata management overhead. To reduce this overhead, SPFS groups 16 contiguous free blocks into a *cluster* of 4 KB and manages the locations of free clusters in the *cluster bitmap* as in conventional file systems. For partially used clusters, we manage the locations of free blocks using *block bitmap hash table* and classical volatile segregated lists.



(a) Block Hashing: $O(B)$ Write, $O(1)$ Read

(b) Extent Hashing: $O(logB)$ Write, $O(logB)$ Read

Figure 5: Block vs. Extent Hashing

Figure 4 shows the layout of physical NVMM space for SPFS. The first 4 KBytes is the *superblock* that contains various metadata including the file system magic number, block/cluster/ inode size, the number of clusters, the number of inodes, metadata for the three hash tables, etc. Then comes the cluster bitmap, where each bit in the cluster bitmap indicates whether all blocks in the corresponding cluster are free or not. If any block in a cluster is in use, its corresponding bit in the cluster bitmap is set to one. Since the cluster bitmap uses one bit per cluster of 4 KB, the space overhead for the cluster bitmap is no larger than that of traditional file systems that manage free space at the granularity of 4 KB blocks.

The cluster bitmap does not indicate which blocks in a cluster are free or in use. Hence, each partially used cluster requires another metadata, the *block bitmap*, which is indexed in the *block bitmap* table. When SPFS allocates some, but not all, blocks in a cluster, it creates and inserts a block bitmap into the *block bitmap table*. The block bitmap table is used only for the clusters that are partially allocated. If a cluster has no free block, which is a common case for files larger than 4 KB, or if all blocks are free, which is also a common case when the file system is initially formatted, no block bitmap is needed in SPFS. To manage free blocks of partially used clusters and serve memory allocation requests quickly, SPFS manages volatile segregated lists constructed from the persistent block bitmap table. For a block allocation request, we select a segregated list based on the allocation request size.

4.2 Extent Hashing

SPFS indexes extents in a hash table called *extent table*. To the best of our knowledge, SPFS is the first hash-based file system that indexes extents using a hash table. HashFS [28], the state-of-the-art hash-based NVMM file system that also manages the file mapping information in a global hash table, requires every block number to be canonically stored in its corresponding bucket. That is, HashFS indexes blocks, not extents, as illustrated in Figure 5(a). Therefore, HashFS not only significantly increases the aggregate size of file mapping structures, but it also slows down writes because writing an extent of B blocks requires as many as B `store` instructions and cacheline flushes.

In contrast to block hashing, our novel *Extent Hashing*

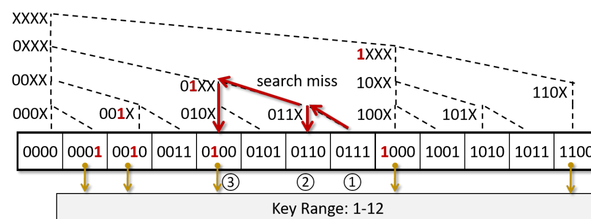


Figure 6: Searching Extent Hash Table

Algorithm 1 *Insert(inode, cluster_num, len, extent)*

```
1: if (len ≤ 0) return
2: current_key = hash(inode,cluster_num)
3: bucket = find_bucket(current_key)
4:     /* e.g., bucket_array[current_key%NumBuckets] */
5: bucket.store(inode, extent)
6: if len = 1 then
7:     return
8: else if cluster_num is odd then
9:     stride_size ← 1
10: else
11:     if cluster_num != 0 then
12:         TNZ ← ffs(cluster_num)−1
13:         stride_size ← previous_pow_of_two(min(len, 1 ≪ TNZ))
14:     else
15:         stride_size ← previous_pow_of_two(len)
16:     end if
17: end if
18: Insert(inode, cluster_num + stride_size, len - stride_size, extent)
```

selects only a few buckets based on the binary representation of cluster numbers, as shown in Figures 5(b) and 6. Extent Hashing bounds the number of pointers for a given extent by $log_2 B$, where B is the number of blocks in an extent.

The insertion and search algorithms of *Extent Hashing* are presented in Algorithms 1 and 2. The algorithms are short but they work with sophisticated bitwise operations such as `ffs` and `fls` (find the first/last bit set in a key) operations. Extent Hashing can be used with any hashing scheme including static and dynamic hashing schemes with various ad hoc optimizations such as linear probing, chaining, and cuckoo hashing. However, for ease of explanation, we will assume we are using static hashing and explain the insert and search algorithms using a walk-through example shown in Figure 6. In the example, we assume hash keys are of 4 bits, and the hash function `hash(inode, cluster_num)` returns `cluster_num` for ease of presentation. Note that Algorithms 1 and 2 can be implemented with any hash function and any hash table implementation.

Insert: Suppose we insert a range of keys [1,12] ([0001_2, 1100_2]) as shown in Figure 6. Initially, we start with the first key and store a pointer to the extent in its corresponding bucket. In the example, we store a pointer in bucket 0001_2. Then, we check how many consecutive zero bits are in the postfix of the hash key of the current bucket, which we refer to as *TNZ* (trailing number of zeros). I.e., TNZ is the number of consecutive zeros in the binary representation with no non-zero digits to the right of it. Since 0001_2 has no zeroes

Algorithm 2 *Search(inode, cluster_num, hash)*

```
1:  pos ← cluster_num
2:  mask ← (1 ≪ fls(cluster_num)) − 1
3:  while true do
4:      key = hash(inode,pos)
5:      bucket ← find_bucket(key)
6:      if bucket.contains(inode, cluster_num) then
7:          return bucket.getExtent(inode, cluster_num)
8:      end if
9:      if pos = 0 then
10:         break
11:     end if
12:     pos ← pos & ((mask≪ffs(pos)) & mask)
13: end while
```

on the right side of the rightmost bit 1, its TNZ is 0. The TNZ determines how many buckets to skip, i.e., the distance between the current bucket and the next bucket where we store the same pointer to the extent. We refer to this distance as *stride length*, which is set to 2^{TNZ}. In the example, since TNZ is 0, the stride length is $2^0 = 1$. Hence, we move to the next bucket (0010_2) and store another pointer to the extent.

The hash key of the current bucket (0010_2) has one zero after the rightmost bit 1. Hence, the TNZ is 1 and the stride length is 2, i.e., (2^1). Therefore, we skip the next bucket and move to the next next bucket (0100_2). Then, we store another pointer there and check the next stride. Since the current TNZ is 2, the stride length is 4. So, we move to bucket 1000_2 (0100_2+4). In bucket 1000_2, we have three consecutive zero bits in the postfix. So, the stride length is 8 (2^3). However, the next bucket offset ($1000_2 + 8$) cannot exceed the range of the given extent. Hence, we decrease the TNZ value one by one ($2^3 \rightarrow 2^2$) until the next bucket position is within the given key range. Finally, we store another pointer in bucket 1100_2 ($1000_2 + 2^2$), and the insertion is complete.

Although the extent size is 12, only 5 pointers are stored in the hash table. If the size of a given extent is B, Extent Hashing stores a maximum of $2 \times log_2 B$ pointers in the worst case. In the best case, we store just one pointer in the hash table. As such, we can significantly reduce the number of pointers (from B to $2 \times log_2 B$) compared to block hashing, in particular, when the extent size is large.

Search: Although the extent hash table does not have a pointer for each hash key, we can find the extent using any hash key within the key range. The search algorithm shown in Algorithm 2 works as follows. If a query searches for an extent using a hash key k, whose binary number is $(b_1 b_2 b_3 b_4)_2$, we first look up bucket[$(b_1 b_2 b_3 b_4)_2$]. If the bucket does not have a pointer to the extent (i.e., a search miss occurs), it could be because the current bucket is not the starting point of a stride, not because the hash table does not contain that data. Therefore, we need to compute the starting index of a possible stride by flipping the trailing non-zero bits starting from the rightmost one and moving left. Thus, assuming b_4 is a non-zero, the next bucket we look up is bucket[$(b_1 b_2 b_3 0)_2$]. If this bucket, again, does not have a pointer to the extent, we continue in the said manner and look up, in sequence, bucket[$(b_1 b_2 00)_2$], bucket[$(b_1 000)_2$], and bucket[$(0000)_2$].

For example, suppose a query searches for hash key 7 (0111_2) in the example shown in Figure 6. Then, we look up bucket[0111_2] (step ①), which will fail as it does not have a pointer to the extent. Then, we search bucket[$(0110)_2$] (step ②) and, finally, bucket[$(0100)_2$] (step ③), which has a pointer to the extent.

The best-case complexity of this search algorithm is $O(1)$, but its worst-case complexity is not constant, but $O(log_2 B)$ where B is the number of buckets. That is, Extent Hashing trades-off search performance for insertion performance.

Probabilistic Fast Lookup: To strike a balance between insertion and search performance, we develop a *fast lookup* optimization. This optimization keeps track of which stride length is the most common and has each query first access the bucket with the most common stride length. For example, the most common stride length in Figure 5(b) is 4 due to the pointers in bucket 0, 4, 8, and 12. Note that there is one pointer with stride 1 in bucket 3, and there is also one pointer with stride 2 in bucket 10. If a query searches for cluster 7 where the most common stride length is 4, the fast lookup optimization searches bucket 4 (4 = 7 - (7%4)) before it accesses bucket 7 and 6 following the search path in order. The rationale behind this optimization is as follows. The search algorithm requires each query to access the nearest buckets in a log scale because the extent size is not known to queries. However, it may result in unnecessary accesses to a large number of buckets if the extent size is large. Therefore, if the bucket corresponding to the most common stride length is first checked, there is a chance of reducing the number of bucket visits. Since the stride length increases in power of 2, i.e., the number of different stride lengths is limited to log scale, the overhead of keeping track of the common stride length is not significant.

4.3 Path-name Resolution

SPFS manages directory entries in another hash table called the *name2inode table*. The name2inode hash table stores file/directory name and directory entry block number pairs using the hash key generated from the VFS dentry, its parent inode number, and the file name. Since SPFS indexes each file/directory entry rather than the full file path, renaming a directory does not affect other files in its sub-directories.

As a stackable file system, the name2inode hash table has directory entries only if the directory has a regular file in NVMM. Stacking files/directories in SPFS follows the standard conventions of stackable file systems [9, 14, 29, 38], i.e., i) if a given regular file name appears in both the upper and lower file systems, then the lower file is hidden; ii) if a given name is a directory, directory entries are combined; iii) if a `readdir` request does not find a directory entry from the name2inode hash table, SPFS forwards the `readdir` request to the lower file system. The directory entry is stored in the

name2inode table only when blocks are migrated from a lower file system to SPFS, if it does not have one already.

One of the drawbacks of using a hash table is that directory entries in the same directory are normally stored in different buckets that causes problems to `readdir`. To resolve this problem, SPFS provides two options. One is to add two persistent pointers to each inode in the name2inode hash table to construct a doubly linked list for the inodes in the same directory. SPFS performs micro-logging (i.e., I/O operation-level logging) when file metadata is updated because multiple indexing structures need to be updated in a failure-atomic manner. The other option is to construct a volatile *readdir index* in DRAM when SPFS is mounted. The readdir index is different from the dentry cache in that it manages the entire structure of all directories in the file system regardless of whether a directory is loaded or not. Therefore, the volatile readdir index must be constructed when SPFS is mounted, and it must be persisted as a persistent index when SPFS is unmounted. Upon a system crash, we may lose updates in the volatile readdir index unlike the persistent readdir chain. To recover from system failures, the readdir index can be reconstructed from scratch by scanning the name2inode hash table. Although the second option increases the memory usage slightly, the low DRAM latency helps improve performance by up to 8% if the workloads are metadata-intensive (that make extensive use of calls such as `create`, `unlink`, and `rename`). For the performance study presented in Section 5, we use the latter option.

4.4 Recovery

SPFS performs micro-logging when file metadata is updated so that `fsck` can rollback uncommitted I/O operations. If a system crashes while creating a file, `fsck` will look up the name2inode hash table using the file name in the operation log and delete its corresponding entry. It will also delete the directory entry using the block number stored in the I/O operation log. In addition, `fsck` will walk the directory tree structure and perform a sanity check as in classic file system recovery methods.

5 Evaluation

We implement SPFS [1] in Linux kernel 5.1 We validated the reliability, robustness, and stability of SPFS using the POSIX file system test suite [3] and the Linux Test Suite [2]. SPFS passed both test suites successfully. In the following, we focus only on the performance aspect of SPFS.

5.1 Experimental Setup

We run experiments on two testbed servers, one with DCPMM and the other with NVDIMM-N. DCPMM server has dual Intel Xeon Gold 5215 processors (10 cores, 2.50 GHz), 128 GB of DDR4 DRAM, 256 GB of Optane DCPMM (2×128 GB), and a 2 TB Samsung 860 EVO mSATA SSD. The NVDIMM-N server has dual Intel Xeon Gold 5218 processors (16 cores, 2.30 GHz), 192 GB of DDR4 DRAM, 16 GB Dell EMC NVDIMM-N, and 512 GB Samsung 970 PRO NVMe SSD. On the NVDIMM-N server, we evaluate SPFS in a virtual environment (16 cores and 32 GB DRAM) using QEMU. Despite the future of DCPMM is uncertain, CXL Type 3 memory devices that provide durability will work with the existing PMDK (or OpenMPDK) ecosystem, and their latency will be higher than that of DRAM (170∼250 nsec) [1]. Therefore, we present the performance on the DCPMM server to evaluate how SPFS performs with NVMMs slower than DRAM.

Table 1: Filebench Workload Characteristics

Workload	File Size	R/W Size	# threads	R:W	# files
Fileserver	128 KB	1024 KB	50	1:2	100K
Webproxy	16 KB	1024/16 KB	100	5:1	100K
Webserver	16 KB	1024/16 KB	100	10:1	100K
Varmail	16 KB	1024/16 KB	16	1:1	100K
OLTP	10 KB	2/2256 KB	200	20:1	10

Table 2: FIU Workload Characteristics

Workload	Dataset Size	Read Size	Write Size	fsync (%)
Moodle	54 GB	55 GB	31 GB	38.922
Usr1	161 GB	171 GB	8 GB	86.025
Usr2	1.5 GB	5 GB	1 GB	75.114

We first quantify the performance effect of Extent Hashing, evaluate the performance of SPFS in standalone mode, and compare SPFS against EXT4-DAX and NOVA on the DCPMM server. Then, we quantify the performance effect of each stackable design of SPFS. Finally, we deploy SPFS on top of three popular Linux file systems, namely, EXT4, F2FS, and XFS, and compare the performance of SPFS+x against x and Ziggurat in both DCPMM and NVDIMM-N servers. File systems are mounted with the default mount options on top of the storage targeted by each design: (1) EXT4, F2FS, and XFS: SSD, (2) NOVA (Copy-on-Write (CoW) mode), EXT4-DAX, and SPFS in standalone mode: DCPMM (3) SPFS+x in stacked mode and Ziggurat: DCPMM+SSD or NVDIMM-N+SSD.

We run experiments using the Flexible I/O tester (FIO) micro-benchmarks [6] and the Filebench macro-benchmarks [33] as well as SNIA's FIU Filesystem SysCall Traces [10]. Tables 1 and 2 show the characteristics of the Filebench and FIU Filesystem SysCall Traces workloads, respectively. We also experiment with RocksDB [4] using the YCSB benchmark [13].

5.2 Analysis of Extent Hashing

In the first set of experiments, we compare the performance of file mapping structures - i) *per-file* `ExtentTree`, which is implemented using the FAST and FAIR B+tree [17], ii) *global* `BlockHash` (proposed and used in HashFS [28]), and iii) *global* `ExtentHash`. Both global block hashing and Extent Hashing are implemented on CCEH [27]. We evaluate the performance of indexing using microbenchmarks.

In the experiments shown in Figure 7, we measure the performance of indexing the extents that make up 8000 256-MB files with varying extent sizes, i.e., the larger the extent

[1] The code is available at https://github.com/DICL/spfs.

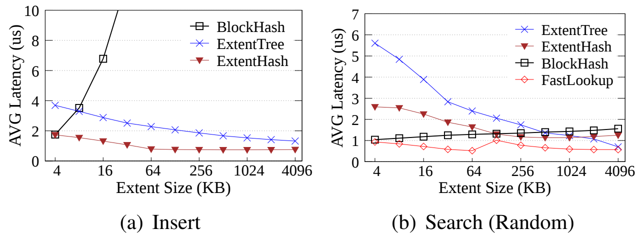


Figure 7: Performance of File Mapping Structures

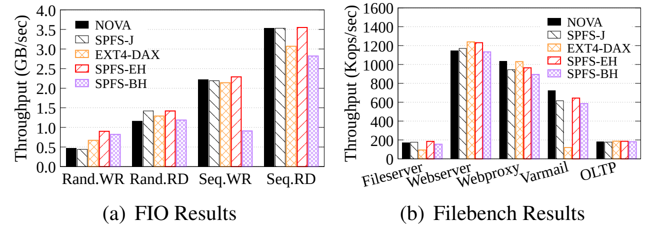


Figure 8: Performance in Standalone Mode (DCPMM)

size, the fewer extents are indexed. Figure 7(a) shows the average latency of inserting one extent to each index. As the extent size increases, the insert latency of `ExtentTree` and `ExtentHash` decreases because the index size decreases. Specifically, when the extent size is 4 KB, the tree height is 4, but when the extent size is greater than 256 KB, the tree height is reduced to 2.

Block hashing shows the worst insertion performance as the extent size increases because the number of pointers to index increases. Specifically, when the extent size is 4 MB, it has to update and call `clwb` for as many times as 1024. As such, its insertion latency is up to 906× higher than that of `ExtentTree`. Extent Hashing shows the fastest insertion latency because hash-based indexes updates fewer number of cachelines than FAST and FAIR B+tree.

Figure 7(b) shows that when the extent size is smaller than 128 KB, `BlockHash` outperforms `ExtentTree` and `ExtentHash` due to its constant lookup cost. Note that, `ExtentHash` accesses multiple buckets following the search path described in Section 4.2. However, as the extent size increases, `ExtentTree` benefits from the reduced index size, making the performance of all indexes similar.

`FastLookup` denotes the performance of Extent Hashing with the fast lookup optimization that we described in Section 4.2. Fast lookup is an optimization affected by probability, but it finds an extent in $O(1)$ with very high probability in the experiments. Therefore, `FastLookup` outperforms `BlockHash`, which suffers from a much larger number of pointers in the hash table. We also observe in the experiments with FIU Filesystem SysCall traces that the probability of finding an extent in $O(1)$ in Usr1 and Usr2 workloads is as high as 60% and 98%, respectively.

5.3 Standalone Mode with DCPMM

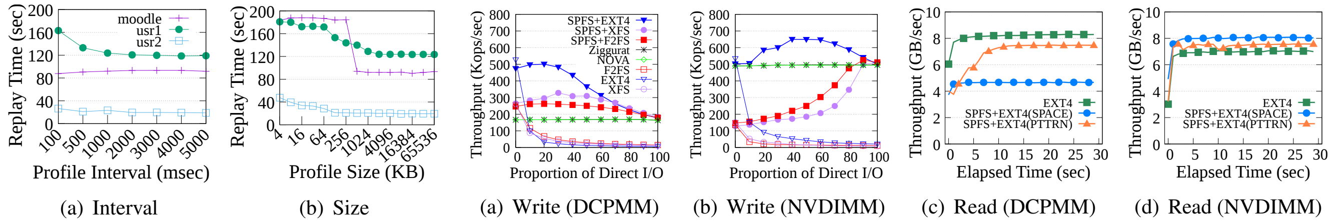
We now compare the performance of SPFS in standalone mode against NOVA and EXT4-DAX. We run the experiments in DCPMM server because SPFS is not intended for use in standalone mode for small NVDIMM-N. To evaluate the performance effect of Extent Hashing, we faithfully implemented the block hashing scheme as proposed in HashFS. We denote the performance of SPFS with Extent Hashing and block hashing as SPFS-EH and SPFS-BH, respectively. Both SPFS-EH and SPFS-BH run in *metadata* mode, i.e., they do not guarantee strong data consistency, but only metadata consistency is guaranteed as in EXT4-DAX. SPFS-J denotes the performance of SPFS-EH in *journal* mode, which logs all data and metadata changes by copying the data into an undo log region if the write size is smaller than 256 KB. If the write size is larger than 256 KB, it performs CoW as in NOVA. We set this threshold size to 256 KB because it conservatively balances the logging overhead and fragmentation issue. If the threshold size is smaller, it leads to fragmentation, i.e., extents are frequently split because CoW allocates a new extent. If it is larger, the logging overhead becomes non-negligible.

5.3.1 FIO Results

The FIO benchmark is used to evaluate sequential and random *read* and *write* performance. Each workload accesses a 10 GB file, and read/write sizes are set to 256KB and 4KB for sequential and random workloads, respectively. Figure 8(a) shows the results. SPFS-EH shows up to 60% higher sequential write throughput than SPFS-BH because block hashing requires much larger metadata accesses. The sequential read throughput of SPFS-BH is also 20% lower than SPFS-EH because larger file mapping metadata adversely affects read performance as well as write performance. For the same reason, the random read and write throughput of block hashing is also 16% and 9% lower than that of Extent Hashing, respectively. In particular, FIO allocates very large extents in advance regardless of the type of workload, i.e., even for random I/Os. Despite large extents, `SPFS-BH` indexes a large number of individual blocks and the lookup performance deteriorates.

NOVA shows similar sequential read and write performance with SPFS-EH. However, the random read and write throughput of NOVA is 19% and 39% lower than that of SFPS-EH because NOVA provides strong data consistency whereas SPFS-EH supports only metadata consistency. With data journaling enabled, SPFS-J shows similar write performance with NOVA as both of them perform CoW. On the other hand, SPFS-J shows 1.2× higher throughput than NOVA for random reads because SPFS manages data blocks in units of extents in DRAM while NOVA indexes write logs in units of pages in DRAM.

As a stackable file system, SPFS does not have to enforce strong data consistency if the lower file system does not require strong data consistency. Eliminating the logging overhead, SPFS-EH shows up to 40% performance improvement for the random write workload compared to NOVA. Since EXT4-DAX also does not log data blocks, it shows higher random write throughput than NOVA and SPFS-J. However,



(a) Interval (b) Size

Figure 9: Profile Parameters (NVDIMM)

(a) Write (DCPMM) (b) Write (NVDIMM) (c) Read (DCPMM) (d) Read (NVDIMM)

Figure 10: Performance Effect of Delegating I/O Requests to Lower File System

EXT4-DAX is consistently outperformed by SPFS-EH for all FIO workloads. Furthermore, for sequential reads and writes, EXT4-DAX is even outperformed by NOVA and SPFS-J despite the fact that they guarantee stronger data consistency. This is because NOVA and SPFS take advantage of CoW for sequential I/O.

5.3.2 Filebench Results

Figure 8(b) shows the experimental results with the Filebench workloads. For the *Fileserver* workload that creates, deletes, reads, appends, and copy files in large I/O units, i.e., 1 MB for reads and writes (copy) and 16 KB for appends, SPFS-J shows 5.7% higher throughput than NOVA (167.67 vs. 176.37) because SPFS-J benefits from the efficient extent-based metadata management, whereas NOVA replaces write logs in units of pages in DRAM. EXT4-DAX shows the worst performance because it suffers from the overhead of unconditional block initialization. Unlike EXT4-DAX, NOVA and SPFS initialize the unwritten portion of the cluster only when needed. SPFS-J and SPFS-EH show similar performance with the *Fileserver* workload because it does not overwrite existing blocks and the logging overhead is negligible. SPFS-BH has 16% lower throughput than SPFS-EH because the read and write granularity of Fileserver workload is 1 MB and Extent Hashing manages large extents more efficiently.

Webserver is a read intensive workload where each thread opens, reads, and closes a file, and every 10th read operation appends a small data to a log file. In this workload, SPFS-J and NOVA show comparable performance.

The *Webproxy* and *Varmail* workloads create and delete many small files in a single directory. In these two workloads, SPFS is outperformed by NOVA because SPFS frequently allocates and deallocates blocks for directory entries, and thus performs metadata journaling for file system consistency, whereas NOVA appends directory entries in a log-structured fashion and hides deallocation overhead via background garbage collection. As a result, NOVA shows up to 9% and 17% higher throughput than SPFS-EH for Webproxy and Varmail, respectively. Efficient directory management is of paramount importance in the native file system, but the primary goal of SPFS is to serves NVMM as a persistent writeback cache to NVMM-oblivious filesystems. Therefore, directory management performance is not optimized. We leave the directory management optimization for future work. EXT4-DAX shows very poor performance for *Varmail* because of two reasons. One is the unconditional block initialization problem mentioned earlier. The other reason is because of additional memory copy overhead from metadata journaling. This overhead is negligible in other workloads because journaling is done in the background. However, *Varmail* calls `fsync()` frequently, which incurs metadata journaling overhead thereby affecting the workload throughput.

The *OLTP* workload emulates database transactions at the file system level. In this transactional workload, synchronous writes affect file system throughput the most. Since all file systems store synchronous writes in NVMM, they do not show any meaningful difference. SPFS-BH shows only 2% lower throughput than SPFS-EH because of small (i.e., 2 KB) random reads/writes that rarely benefit from extents.

5.4 Quantification of Stackable Design

5.4.1 Parameters for Sync Point Profiler

In this section, we quantify how the profile interval and the write size threshold for the Sync Point Profiler affects performance using three FIU Filesystem SysCall Traces workloads. We present the results on the NVDIMM-N server, but the results on the DCPMM server are almost the same.

Figure 9(a) shows that the performance of SPFS is insensitive to the profile interval unless it is set small ($<$ 500ms). Obviously, the interval between transactional writes can vary across applications. Therefore, for the rest of the experiments, we choose 1 second as the default. Figure 9(b) shows the results as the write size parameter is varied (x-axis). It shows that the replay time of the `Moodle` workload improves significantly when the write size parameter is set to be equal or larger than 1 MB because this workload has relatively large 1 MB synchronous writes. For `Usr1` and `Usr2`, we see that performance gradually improves as the write size increases, but then remains relatively constant beyond 1 MB. Based on these observations, we conservatively set the default write size parameter to 4 MB - a sufficiently large value that was also used as the default value in Ziggurat [39]. All results that follow use this value.

5.4.2 Delegating I/O Requests to Lower File System

We now perform synthetic microbenchmark experiments to validate our proposition that migrating files to NVMM does not always guarantee better performance. That is, we analyze

which types of I/Os benefit from promotion and when they benefit from demotion.

Performance Effect of Delegation: As a stackable file system, SPFS shines when synchronous and asynchronous I/O workloads are mixed. To test various mixed workloads, we use *diomix*, which is a synthetic workload generated from a mix of two sequences of file operations, one for buffered I/O (BIO) and the other for direct I/O (DIO), of the *Fileserver* workload of Filebench, and whose ratio between BIO and DIO can be controlled.

Figure 10(a) shows the results for *diomix* in DCPMM server, as the DIO rate changes. We disable background demotion to only quantify the effect of promotion. We observe that the I/O throughput of Ziggurat is insensitive to the DIO rate and that it fails to leverage the faster page cache in DCPMM server, which leads to the same performance as NOVA. As a result, they are consistently outperformed by SPFS+*x*, which delegates BIO to EXT4, F2FS and XFS, and benefits from the low latency of the page cache in DRAM. Note that EXT4, F2FS, and XFS also benefit from the page cache, and when there is no DIO, each file system shows 10%, 3%, and 1% higher throughput, respectively, than its SPFS+*x* counterpart. This is because of the overhead that comes from the stackable design. Specifically, SPFS+*x* looks up its name2inode table just to find out it does not have the requested file. This exemplifies the importance of indexing performance in SPFS. We observe sharper and then continued performance decline as the rate of DIO increases. Unlike EXT4, F2FS and XFS, the throughput of SPFS+*x* show much smoother curves as SPFS+*x* detects the I/O types and steers the BIOs to the lower file system while absorbing the DIOs in DCPMM, benefiting from the device aware stackable design of SPFS+*x*.

Figure 10(b) shows the results for the same *diomix* workload in NVDIMM server. Because the page cache of the lower file system has the same access latency with NVDIMM, SPFS+*x* does not benefit from delegating write requests to the lower file system but suffers from its stackable design overhead. Again, the performance of Ziggurat is similar to NOVA as it aggressively steers most writes to NVMM. In contrast, SPFS+*x* is designed to use NVMM conservatively and gradually demote files in the background. We could not evaluate Ziggurat for the case when NVMM is full as it crashes.

Performance Effect of Demotion: While promotion improves write performance of the lower file system, it may degrade read performance when NVMM is slower than DRAM. In the experiments shown in Figures 10(c) and 10(d), we pre-populate file systems with 64 16 MB transactional log files, i.e., SPFS stores them in NVMM, and run a synthetic microbenchamrk that reads random blocks from those files. In DCPMM server, the read throughput of EXT4 file system is higher than SPFS+EXT4 because it eagerly copies all the requested blocks to the page cache, whereas SPFS+*x* demotes files, i.e., copies them from DCPMM to DRAM/disk in a lazy manner. Specifically, `SPFS+EXT4(PTTRN)` demotes a file to

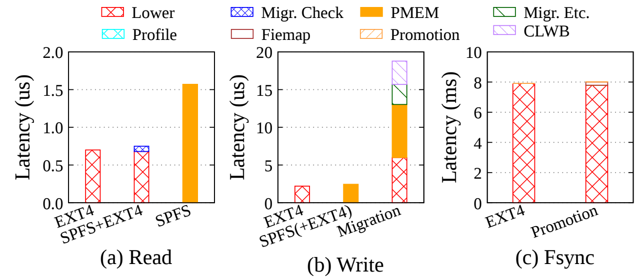


Figure 11: Latency breakdown of each mode in DCPMM

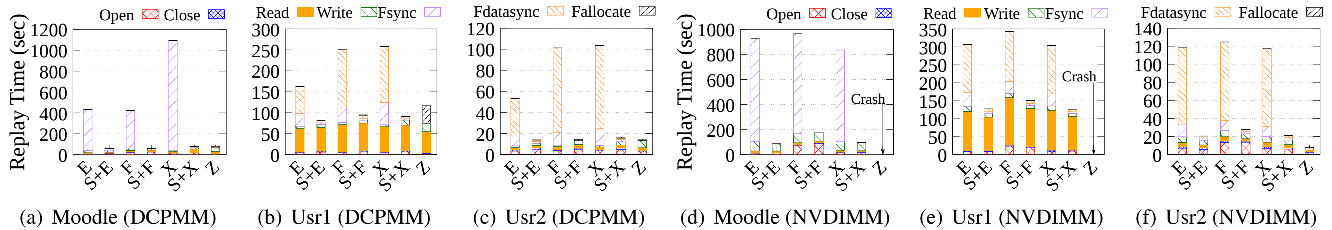
the lower file system and copies to the page cache if the file access pattern changes to be read-intensive. As more read requests are processed, the Sync Factors of promoted files decrease, and when they become lower than a hard limit set by administrators, they are demoted to the lower file system such that they can benefit from the page cache in DRAM. Thus, read performance of `SPFS+EXT4(PTTRN)` improves over time to a level similar to page cache performance. This result confirms the well-known fact that performance is improved by placing frequently accessed data in the fastest memory, which Ziggurat has neglected.

If such a hard limit on the Sync Factor is not set by administrators, a file is not demoted to the lower file system unless the NVMM space is running low (denoted as `SPFS+EXT4(SPACE)`. Therefore, `SPFS+EXT4(SPACE)` does not demote files and read requests to those files suffer from higher access latency of DCPMM. In contrast, demoting files from NVDIMM to the page cache on the NVDIMM server does not improve read performance, but counteracts it. As a result, `SPFS+EXT4(SPACE)` shows the highest throughput in NVDIMM server. In the default settings, background demotion is triggered when more than 80% of NVMM space is used. When files are demoted to the lower file system in the background, the foreground write throughput of SPFS+*x* is reduced by up to 40% due to the limited bandwidth of NVMM and also due to conflicting SPFS metadata updates. To minimize performance interference, SPFS suspends the background demotion while foreground processes perform I/O, unless there are no free blocks in NVMM.

5.4.3 Stacking Overhead

As a stackable file system, SPFS places additional latency on the lower file system in exchange for improving the performance of small synchronous writes. In the experiments shown in Figure 11, we breakdown the latency of read, write, and fsync using a synthetic workload that performs random reads and writes to 1 MB file that consists of a single extent.

`SPFS+EXT4` denotes the read latency when the extent is not found in SPFS and is read from the lower file system - EXT4. The stacking overhead (i.e., the overhead to check whether the corresponding extent has been migrated to NVMM or not) accounts for 9.89%, and thus `SPFS+EXT4` shows similar latency with `EXT4`. `SPFS` denotes the read latency when the



(a) Moodle (DCPMM) (b) Usr1 (DCPMM) (c) Usr2 (DCPMM) (d) Moodle (NVDIMM) (e) Usr1 (NVDIMM) (f) Usr2 (NVDIMM)

Figure 12: FIU Trace Replay Time S:SPFS, E:EXT4, X:XFS, F:F2FX, Z:Ziggurat

extent is in SPFS, i.e., NVMM. Due to the high latency of Optane DCPMM, the read latency of `SPFS` is about 2.26× higher than that of EXT4.

In Figure 11(b), the write latency of EXT4 (`EXT4`) is similar to the write latency when the write is steered to NVMM (`SPFS(+EXT4)`). That is, unless fsync is called, there is not much difference in latency whether the write is steered to DCPMM or the VFS cache. `Migration` denotes the latency of the first write to an extent that the profiler decided to migrate from the lower file system to SPFS. `Migration` has a high latency, but it is a one time tax. Once migrated, subsequent fsync calls will be replaced with nop..

Figure 11(c) shows the fsync latencies when SPFS bypasses fsync to the lower file system and when it decides to promote a file in the lower file system to SPFS. The promotion overheads such as `Fiemap` and `Profile` account for 2.68% and 0.01% of total latency.

5.5 Stacked Mode Performance Comparison

Finally, we run real world trace FIU and YCSB workloads and compare the performance of SPFS in stacked mode (SPFS+*x*) against file systems for large block devices, i.e., EXT4, XFS, F2FS, and Ziggurat.

5.5.1 FIU Traces

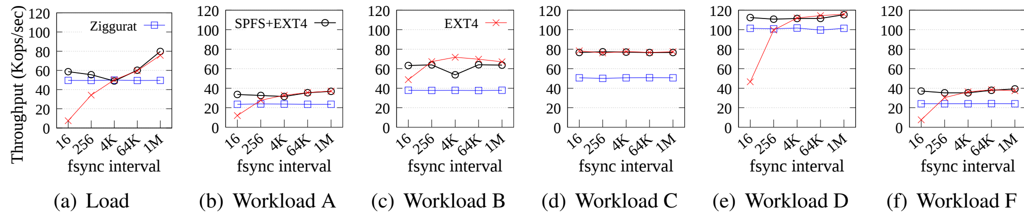
Figure 12 shows the performance results using the FIU Filesystem SysCall Traces [10]. For these experiments, we measure the replay time on each file system as we submit the file system operations from the traces in batches. Thus, for all results for the FIU workloads, lower is better.

Although the FIU workload consists of traces from six applications - *Backup*, *Gsf-filesrv*, *Ug-filesrv*, *Moddle*, *Usr1*, and *Usr2*, the performance results of *Backup* (1.2 TB, 0.001% `fsync()`), *Gsf-filesrv* (190 GB, 0.326% `fsync()`), and *Ug-filesrv* (812 GB, 0.001% `fsync()`) are not presented because Ziggurat crashes for those large FIU workloads not only in NVDIMM but also in DCPMM servers and also because they are not transactional workloads, i.e., `fsync()` calls account for less than 0.3%. Even if we evaluated the performance of Ziggurat by reducing the size of those workloads small enough to fit in DCPMM, we observed that Ziggurat is outperformed by EXT4, F2FS, and XFS, and SPFX+*x* because Ziggurat fails to leverage the fast VFS cache. The performance of SPFS+*x* (`SPFS+EXT4`, `SPFS+F2FS`, and `SPFS+XFS`) is similar or slightly worse than that of *x* (EXT4, F2FS, and XFS) for the workloads where `fsync()` calls are rarely made. Without `fsync()` calls being made and steering writes to NVMM, the added overhead of the stacked file system tends to make SPFS+*x* perform worse than *x*.

With *Moodle*, *Usr1* and *Usr2*, calls to `fsync()` are frequently made. Therefore, EXT4, F2FS, and XFS suffer from high synchronization overhead while SPFS+*x* and Ziggurat eliminate this overhead by steering synchronous writes to NVMM. Thus, for *Moodle*, SPFS+*x* reduces the trace reply time in DCPMM server to only 14%, 15%, and 7% of the *x* counterparts EXT4, F2FS, and XFS, respectively. Similarly, for Usr1, SPFS+*x* shows 2×, 2.6×, and 2.8×, and for Usr2, 3.8×, 7.1×, and 6.5× faster trace replay times, compared to the *x* counterparts EXT4, F2FS, and XFS, respectively.

Ziggurat also outperforms EXT4, F2FS, and XFS for the *Moodle*, *Usr1*, and *Usr2* workloads. Compared to Ziggurat, `read()`, `write()`, and `fallocate()` are consistently faster with SPFS+*x*. The write time of Ziggurat is higher than SPFS+*x* because it profiles and steers each individual write to NVMM while SPFS+*x* migrates them in a lazy manner, that is, only at `fsync()` calls with intervals less than one second and aggregate flush sizes less than 4 MB. For `fallocate()`, Ziggurat spends a significant amount of time initializing allocated blocks. However, SPFS creates files on the lower file systems first, such that it benefits from the highly efficient *uninit* and *unwritten* states (i.e., allocated and mapped but uninitialized blocks) of the disk file systems, which prevents applications from reading garbage blocks even if allocated blocks have not yet been initialized. However, due to the stacking overhead, `open()` is slower in SPFS+*x* than Ziggurat. Also, `fsync()` and `fdatasync()` are faster with Ziggurat because they are no-ops if previous writes were steered to NVMM. Overall, due to faster `write()` and `fallocate()`, SPFS+*x* is up to 1.44× and on average 1.16× faster than Ziggurat in DCPMM server.

On the NVDIMM server, we could not run the *Moodle* and *Usr1* workloads with Ziggurat because their sizes are larger than the NVDIMM size. For the *Usr2* workload where the average I/O size is 1.2 KB, Ziggurat steers most writes to NVDIMM, and thus shows the performance of the in-memory file system - NOVA and outperforms SPFS +*x*. Similar to the results on the DCPMM server, SPFS+*x* improves the performance of *x* by up to 9.9×, 2.4×, and 5.8× for the Moodle, Usr1, and Usr2 workloads, respectively.



(a) Load (b) Workload A (c) Workload B (d) Workload C (e) Workload D (f) Workload F

Figure 13: YCSB Throughput of RocksDB with Varying Frequency of `fsync()` (DCPMM)

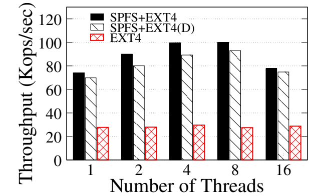


Figure 14: Load Throughput (NVDIMM)

5.5.2 RocksDB

Finally, we evaluate the performance of SPFS+EXT4, Ziggurat, and EXT4 using RocksDB v.6.2.2. For the experiments shown in Figure 13, we load the database with 4 million 1 KB records in the loading phase (`Load`) in the DCPMM server. In the transactions phase, 4 million queries in uniform distribution are submitted in batches for each workload. RocksDB offers various write options including whether to call `fsync()` to flush dirty pages. Our experiments measure YCSB throughput while varying the `fsync()` interval by enabling the `setSync` option for every 16 writes, 256 writes, 4096 writes, and so on, that is, in multiple of 16 increments. Figure 13 shows the throughput results, which can be summarized as follows.

EXT4: EXT4 performance improves as we increase the `fsync()` interval. If `fsync()` is called often, EXT4 suffers because of the slow block device. As `fsync()` is called less, EXT4 benefits from the low latency of the page cache and performance improves.

Ziggurat: Ziggurat performance is insensitive to the `fsync()` interval. This is because Ziggurat profiles individual writes, and as all writes are smaller than 4 MB, they are all steered to NVMM resulting in the same performance as NOVA.

SPFS+EXT4: In contrast, SPFS+EXT4 considers the total number of written bytes to be flushed by `fsync()`. Therefore, if `fsync()` is called per every 4096 or fewer writes, SPFS stores the 4 MB or smaller synchronous writes in NVMM and significantly reduces the `fsync()` overhead. Thus, for the `Load` workload, `SPFS+EXT4` shows 7.7× and 1.6× higher throughput than EXT4 when the `fsync()` interval is 16 and 256 writes, respectively. We observe all small WAL log files are migrated to NVMM as expected, whereas all SSTable files, which contain key-value records, are stored in the EXT4 file system because its size (64 MB) is much larger than the profiling threshold of SPFS (4 MB). Nonetheless, SPFS+EXT4 outperforms Ziggurat, which stores both WAL and SSTables in NVMM. This is because SPFS leverages DRAM, NVMM, and SSD characteristics altogether, while Ziggurat relies only on NVMM. If `fsync()` is called less frequently, e.g., `fsync()` is called every 64K or 1M writes, SPFS+EXT4 stores all writes in EXT4 and benefits from the VFS cache and periodic writebacks. Therefore, `SPFS+EXT4` shows similar performance with EXT4.

In the experiments shown in Figure 14, we measure YCSB Load throughput on the NVDIMM server, varying the number of client threads while the `fsync()` interval is fixed to 256. Due to the frequent `fsync()`, EXT4 does not scale with the number of client threads. However, the throughput of SPFS+EXT4 increases up to 8 threads because it absorbs the synchronous writes in NVDIMM. When the number of client threads is 16, the throughput degrades because the number of total threads (i.e., client and background compaction threads) exceeds the number of available cores and memory contention occurs. Note that SPFS+EXT4(D) denotes the performance of SPFS+EXT4 when NVDIMM is full and the background demotion migrates files from NVDIMM to the lower file system. Due to the performance interference, the demotion decreases the throughput by up to 7%.

6 Conclusion

Managing two different storage devices with completely different properties in a single file system has practical limitations. In this study, we designed and implemented *SPFS*, a stackable file system for NVMM that exploits the performance of NVMM for order-preserving small synchronous writes and yet takes advantage of the faster DRAM cache as well as the large capacity that legacy block device file systems provide. In addition, SPFS manages all file system metadata in dynamic hash tables that are built on Extent Hashing that exhibits fast insertion as well as fast scan performance.

We perform extensive evaluations and compare SPFS with state-of-the-art file systems. In standalone mode, SPFS shows comparable performance to NOVA, while in stacked mode, SPFS+x improves performance by up to 9.9× compared to the lower file system x executing alone.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments and feedback. We also thank our shepherd, Youyou Lu for guiding us during the revision process.
This research was supported in part by Samsung Electronics, and also by NRF (grant No. NRF2022R1A2C2091680), IITP (grant No. 2021-0-01817), and ETRI (grant No. 20ZS1310). Much of this work was done when Hobin Woo was enrolled in the Master program at Sungkyunkwan University. Daegyu Han contributed equally as he took over the work after H. Woo graduated. The corresponding author is Beomseok Nam.

References

[1] Compute Express Link CXL Latency How Much is Added at HC34. https://www.servethehome.com/compute-express-link-cxl-latency-how-much-is-added-at-hc34/.

[2] Linux Test Suite. https://linux-test-project.github.io/.

[3] POSIX File System Test Suite. https://github.com/pjd/pjdfstest.

[4] RocksDB. https://rocksdb.org/.

[5] Samsung SZ985 Z-NAND SSD. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[6] Jens Axboe et al. FIO (Flexible I/O Tester). https://github.com/axboe/fio.

[7] Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. Persistent Memory: A Survey of Programming Support and Implementations. *ACM Computing Surveys (CSUR)*, 54(7):1–37, 2021.

[8] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, pages 689–703, 2021.

[9] Neil Brown. Overlay Filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[10] Daniel Campello, Hector Lopez, Ricardo Koller, Raju Rangaswami, and Luis Useche. Non-blocking Writes to Files. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 151–165, 2015.

[11] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1077–1091, 2020.

[12] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. HiNFS: A Persistent Memory File System with Both Buffering and Direct-Access. *ACM Transactions on Storage (TOS)*, 14(1):1–30, 2018.

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010.

[14] Puja Gupta, Harikesavan Krishnan, Charles P. Wright, Mohammad Nayyer Zubair, Jay Dave, and Erez Zadok. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical report, Stony Brook University, 2004.

[15] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–162, 2021.

[16] Tyler Hicks, Dustin Kirkland, and Michael Halcrow. eCryptFS. http://www.ecryptfs.org/.

[17] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th Usenix Conference on File and Storage Technologies (FAST)*, pages 187–200, 2018.

[18] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, 2015.

[19] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. BetrFS: Write-Optimization in a Kernel File System. *ACM Transactions on Storage (TOS)*, 11(4):1–29, 2015.

[20] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of the 17th Usenix Conference on File and Storage Technologies (FAST)*, pages 191–205, 2019.

[21] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 993–1005, 2018.

[22] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. Storage Gardening: Using a Virtualization Layer for Efficient Defragmentation in the WAFL File System. In *Proceedings of the 17th USENIX*

Conference on File and Storage Technologies (FAST), pages 65–78, 2019.

[23] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 161–177, 2022.

[24] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 460–477, 2017.

[25] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286, 2015.

[26] Avantika Mathur, M. Cao, and A. Dilger. Ext4: The Next Generation of the Ext3 File System. *;login: Usenix Magazine*, 32, 2007.

[27] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage (FAST)*, pages 31–44, 2019.

[28] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. Rethinking File Mapping for Persistent Memory. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 97–111, 2021.

[29] Junjiro R. Okajima. AUFS - Another Union Filesystem. http://aufs.sourceforge.net/.

[30] D. Rosenthal. Evolving the Vnode Interface. In *USENIX Summer*, 1990.

[31] Simon Sharwood. Last Week Intel Killed Optane. Today, Kioxia and Everspin Announced Comparable Tech: Rumors of Storage-Class Memory's Demise May Have Been Premature. https://www.theregister.com/2022/08/02/kioxia_everspin_persistent_memory/.

[32] Pradeep J Shetty, Richard P Spillane, Ravikant R Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building Workload-Independent Storage with VT-Trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–30, 2013.

[33] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *;login: USENIX Magazine*, 41(1):6–12, 2016.

[34] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 93–111, 2021.

[35] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO Stack for Flash Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 211–226, 2018.

[36] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC)*, pages 349–362, 2017.

[37] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, 2016.

[38] Erez Zadok, Ion Badulescu, and Alex Shender. Extending File Systems Using Stackable Templates. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC)*, pages 57–70, 1999.

[39] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, pages 207–219, 2019.