Revisiting Virtual File System for Metadata Optimized Non-Volatile Main Memory File System

Ying Wang, Dejun Jiang and Jin Xiong
SKL Computer Architecture, ICT, CAS; University of Chinese Academy of Sciences
{wangying01, jiangdejun, xiongjin}@ict.ac.cn

***Abstract*—Emerging non-volatile main memories (NVMMs) provide persistency and low access latency than disk and SSD. This motivates a number of works to build file systems based on NVMM by reducing I/O stack overhead. Metadata plays an important role in file system. In this paper, we revisit virtual file system to find two main sources that limit metadata performance and scalability. We thus explore to build a metadata optimized file system for NVMM-DirectFS. In DirectFS, VFS cachelet is first co-designed with VFS and NVMM file system to reduce conventional VFS cache management overhead meanwhile retaining file lookup performance. DirectFS then adopts a global hash based metadata index to manage both VFS cachelet and metadata in NVMM file system. This helps to avoid duplicated index management in conventional VFS and physical file system. In order to increase metadata operation scalability, DirectFS adopts both fine-grained flags and atomic write to reduce limitations during concurrency control and crash consistency guaranteeing. We implement DirectFS in Linux kernel 4.18.8 and evaluate it against state-of-the-art NVMM file systems. The evaluation results show that DirectFS improves performance by up to 59.2% for system calls. For real-world application varmail, DirectFS improves performance by up to 66.0%. Besides, DirectFS scales well for common metadata operations.**

***Index Terms*—non-volatile memory, file system, virtual file system, metadata performance, metadata scalability**

I. Introduction

Emerging fast and byte-addressable Non-Volatile Memories (NVMs), such as ReRAM [4], Phase Change Memory [2], [44] and recent 3D-XPoint [19], allow one to store data persistently in main memory. This motivates a number of works to build file systems on the non-volatile main memory (NVMM) [9], [13], [15], [16], [23], [37], [45], [50], [54]–[56]. Since NVMM provides sub-microsecond latency that is much lower than hard disk and solid state drive (SSD), the software overhead of file system itself becomes the main factor affecting file system performance. In addition, as the number of CPU cores increases, it allows a large number of threads to access file system simultaneously. Thus, the scalability of file system also becomes critical. Recently, a number of NVMM file systems are proposed to bypass page cache and generic block layer to reduce software overhead [9], [10], [16], [55] and allocate multiple resources to improve scalability [17], [29], [55].

Metadata plays an important role in file systems. In general, metadata provides descriptive information about files, directories, and file system itself. Metadata is usually involved in most file system operations, such as file creation/deletion and concurrency control. For example, path lookup is a common metadata operation before accessing a file. Previous work shows that path lookup reaches 54% of total execution time for some command-line applications [49]. Since NVMM file systems greatly reduce data access latency, metadata performance becomes critical especially for operating small files. We observe that metadata operations account for 53.3% of the total execution time when reading 1 M small files of 8 KB in NVMM file system (evaluated on Intel Optane DC persistent memory [20]), which is a typical scenario for thumbnails and profile pictures workload in Internet services (e.g., social network and e-commerce) [5]. Moreover, with the increasing number of CPU cores, metadata scalability is also critical to allow file system to catch up the hardware trends and fully utilize multi-cores [36].

Traditionally, metadata is stored and managed separately in two-layer: virtual file system (VFS) and physical file system (such as ext4). In this paper, we revisit virtual file system for NVMM file systems and find two main sources that limit metadata performance and scalability. First, VFS caches file system metadata (e.g., inode and dentry) in DRAM to accelerate file operations by avoiding frequently accessing underlying slow disks but is volatile. Physical file system stores metadata in persistent storage media (SSD, disk) to prevent data loss but is slow. For disk-based file systems, this two-layer metadata management can optimize performance on the basis of ensuring data persistence. However, NVMM provides access latency of the same order of magnitude as DRAM. Managing metadata separately in VFS and physical file system results in redundant metadata management (e.g., metadata index management) as well as lookup overhead.

Secondly, the two-layer metadata management requires to update both VFS and physical file system atomically for metadata writes. For example, the *create* system call not only requires building dentry, inode in physical file system and the related metadata cache in VFS, but also requires updating the metadata and the cached one of the parent directory in both physical file system and VFS. During this process, in order to provide concurrency control, VFS adopts a lock to the directory in which write operations execute. This limits NVMM file system scalability. Moreover, as for write operations, the NVMM file system adopts techniques (e.g. journaling and log-structuring) to provide crash consistency. Guaranteeing crash consistency itself does not scale well when multiple threads compete for journal or log resources. As a result, these two limitations are combined together when manipulating shared directory and further decrease file system

scalability. Although a few recent works propose to reduce lock granularity by partitioning [24], [25], [29], [34], [47], [55], metadata operations during both concurrency control and crash consistency guaranteeing are still required to be scalable.

Similar to removing page cache to optimize data path [9], [13], [16], [29], [55], one intuitive approach is to remove VFS and directly manage metadata in NVMM file system. However, VFS is an important abstraction on top of physical file systems, providing operation compatibility to various physical file systems. Directly removing VFS breaks this compatibility. In addition, NVMM has higher read latency than DRAM [20] and reading metadata directly from NVMM suffers from lookup performance degradation.

Thus, in this paper, we take the principle of retaining VFS abstraction and co-design VFS and NVMM file system to improve metadata performance and scalability. We first propose VFS cachelet by revisiting the conventional VFS cache design. VFS cachelet is a reduced read cache in VFS for caching read-dominate per-file metadata to sustain file lookup performance. For other metadata, we directly read from underlying NVMM file system. In such doing, VFS cachelet can reduce metadata management overhead (e.g. cache (de-)construction/maintenance). Secondly, we design a global hash based index to manage both VFS cachelet and metadata in NVMM file system. This unified indexing approach avoids duplicated index operations in conventional VFS and physical file system. Finally, we fully exploit fine-grained flags and atomic write to remove directory lock in VFS. In addition, we design atomic write based logging structure to reduce scalability limitations during crash consistency guaranteeing.

Based on the above co-designs between VFS and NVMM file systems, we build DirectFS, a metadata optimized high performance and scalability file system for non-volatile main memory. We implement DirectFS in Linux kernel 4.18.8 and compare it against state-of-the-art NVMM kernel-based file systems ext4-dax [10], PMFS [16] and NOVA [55]. The evaluation results show that compared to NOVA, DirectFS reduces execution time by up to 59.2% and 36.86% on average for widely used system calls. And DirectFS reduces execution time by up to 66.3% and 27.0% on average to NOVA for common command-line applications. For typical real-world application scenarios, DirectFS outperforms NOVA by up to 50.0%, 66.0%, and 42.3% for small file access, varmail and fileserver respectively. And DirectFS scales well for metadata operations on the shared directory.

II. Background and Motivation

A. Non-volatile main memories

Non-Volatile Memories (NVMs), such as ReRAM [4], Phase Change Memory [2], [44] and Intel 3D-XPoint [19], provide fast, non-volatile and byte-addressable accessing. These techniques allow one to build file system on non-volatile main memory (NVMM). Some works assume that NVMM has the same read latency and 3x-10x write latency with DRAM [30], [37], [51], [57], [61], and they mainly optimize write operations on NVMM file systems. Recently, Intel releases an enterprise product: the Intel Optane DC Persistent Memory Module (Optane DC PMM), which provides true hardware performance of NVMM. We observe that Optane DC PMM has similar write latency to DRAM, but read latency is about 3x-5x slow than DRAM (These results are consistent with the previous work [20]).

B. VFS and metadata operations

Virtual File System (VFS) provides an abstraction for interacting with multiple instances of physical file systems. For example, VFS holds mounting points for different physical file systems and hands over file-related system calls to specific file system instance. In addition, VFS acts as a caching layer for file access. In physical file systems, a dentry metadata mainly contains file name and the corresponding inode number. An inode metadata records file properties (e.g., file size and creation time). VFS caches these two types of metadata as icache (inode cache) and dcache (dentry cache). Before accessing a file, one needs to lookup VFS. Taking path lookup as an example, one needs to first look up dcache for each path component. In case of finding dcache (namely warm cache), VFS ends the lookup process and returns the corresponding metadata. Otherwise, one needs to lookup dentry and inode in physical file system (namely cold cache). After successfully locating the file in physical file system, VFS builds icache and dcache for the fetched metadata. On the other hand, metadata write operations, such as rename and create, require updating both VFS and physical file system to maintain cache consistency. For NVMM file systems, VFS caching results in two-layer metadata operation overhead, including redundant metadata management as well as lookup overhead.

There are two main types of metadata. One is file system metadata, such as size and free space of file system. The free space management is a factor affecting file operations efficiency. We can use pre-allocating [29], [37], [55] to improve the performance and scalability of free space management. The other one is file metadata, including file properties (such as file name, file size and file mtime) and file data location. Since all file operations require accessing file properties first, we mainly focus on the file properties metadata in this paper. For file data location, existing works make efforts to optimize file data block management [9], [16], [28], [55], [56]. These works are orthogonal to us.

C. Metadata performance

Since low access latency of NVMM reduces the overhead of reading/writing file data, metadata becomes an important factor affecting file system performance. Existing two-layer metadata management introduces extra overhead in NVMM file system, including cache (de-)construction/lookup and maintaining cache consistency between VFS and physical file system.

We measure the execution time of metadata operations as well as the total execution time in ext4-dax [10], NOVA [55], PMFS [16] when reading/writing 1/8/32KB data. Note that,

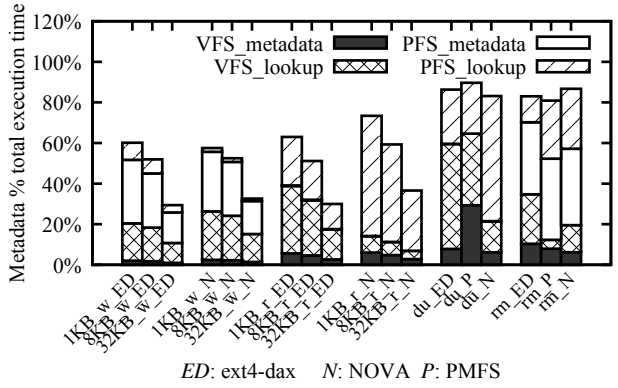


Fig. 1. The percentages of times spend on metadata operations when VFS in cold cache. For xtics, *1KB* presents file size, *r/w* presents read/write operations and *ED* presents ext4-dax file system.

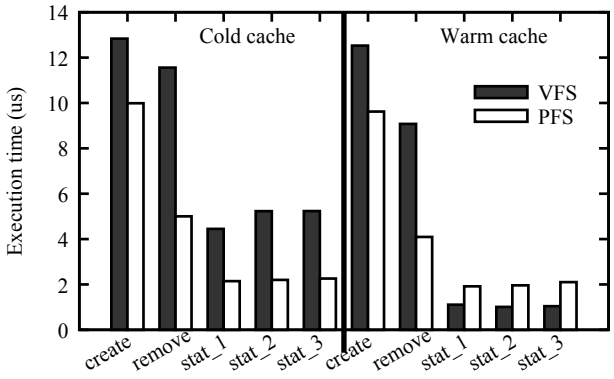


Fig. 2. The impact of metadata cache. *create*, *stat* and *remove* present system call in one-level directory. *"_2"* and *"_3"* refer to evaluation running in two-level and three-level directory respectively.

file reads are tested without metadata being cached in VFS. Figure 1 shows the time percentages of metadata operations [1]. We further divide metadata operations into four parts: metadata cache (de-)construction/maintenance (*VFS_metadata*), metadata operations in physical file system (*PFS_metadata*, including metadata read/write, rebuilding directory index and guaranteeing crash consistency) and metadata lookup in VFS (*VFS_lookup*) and physical file system (*PFS_lookup*). For reading/writing 1KB data, the metadata operations account for 63.5% of total execution time on average. Even for reading 32 KB data, the metadata takes about 33.3% of total execution time. This is because one needs to lookup file in both VFS and physical file system in case of no metadata cached in VFS. Then extra efforts are required building dcache and icache in VFS. Moreover, we test two metadata-dominant command-line applications *du* and *rm*. On average, the metadata operations account for 86.5% of total execution time. This is because performing lookup or metadata updating in both VFS and physical file system. Thus, the two-layer metadata management should be optimized for NVMM file systems.

To analyze the impact of metadata cache on metadata performance, we evaluate the performance of metadata operations with metadata cache (VFS in Figure 2) and without metadata cache (performing all metadata operations on physical NVMM file system, PFS in Figure 2) based on NOVA [55][2].

Figure 2 shows the performance. *Warm cache* represents that VFS contains file metadata before accessing the file, which runs the experiment twice and drops the first one. *Cold cache* means VFS does not contain file metadata which runs the experiment once. Since *VFS* requires additional cache operation overhead for metadata write operations, *PFS* performs well for *open* and *remove* in warm cache and cold cache. For metadata read operations (*stat*) in cold cache, *PFS* reduces execution time by 55.5% compared to VFS. This is because when VFS does not contain file metadata cache, one requires finding both VFS and physical file system and building metadata cache in VFS. *PFS* only performs file lookup operations on the physical file system. For metadata read operations in warm cache, *PFS* increases execution time by 89.5% compared to VFS. This is because NVMM has poor read performance and traditional physical file system lookup slowly than VFS, such as slow index structure. Since the locality of file access, the VFS cache hit rate is closed to 100% in some operations [49].

From these observations, we can see that although existing VFS cache reduces the performance of some metadata operations for NVMM file systems, removing all metadata cache is also not desirable due to the long NVMM read latency. This motivates us to revisit metadata cache in VFS. To avoid double metadata lookups, we design a global metadata index to enable one-time lookup (Section III-C). Meanwhile, we propose a reduced metadata cache (Section III-B) to sustain lookup performance with reduced management costs.

D. Metadata scalability

For NVMM file systems, the scalability of metadata operations is important. However, it is currently limited by existing concurrency control with VFS and crash consistency guaranteeing in NVMM file systems. As for concurrency control, existing file systems use lock to guarantee atomicity [9], [13], [15], [16], [37], [45], [54]–[56]. They use a directory lock (rwlock) in VFS to sequentially perform all metadata update operations on the directory. For example, when two concurrent threads create files *"foo1"* and *"foo2"* separately in directory *"example"*, their creation operations are performed sequentially. The directory lock limits the scalability of concurrent operations. We run mdtest [22] to evaluate scalabilities of both VFS and physical file systems when accessing files in a shared directory[3]. Figure 3 shows the results of ext4-dax and NOVA with increasing threads. For metadata write operations (*create_ED*, *create_N*), they require operating on both VFS and physical file system and show poor scalability because of directory lock. For metadata

[1]The experiment configurations are the same in Section V.

[2]All VFS-based file systems, such as PMFS [16], SCMFS [54] show the similar results to NOVA.

[3]The experiment configurations are the same in Section V-C.

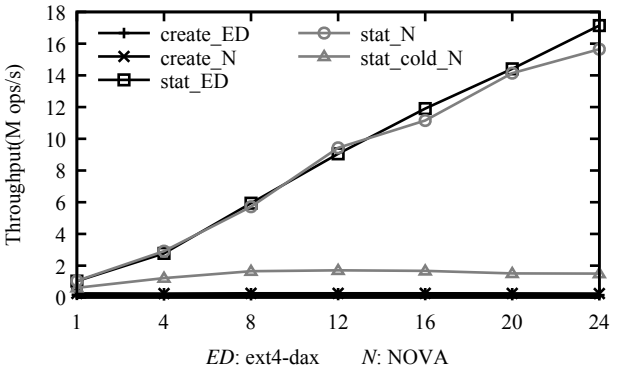


Fig. 3. Metadata scalability. The xtics shows the number of test threads.

read operations (*stat_ED*, *stat_N*), VFS uses RCU-walk [27] to support concurrent reads when metadata is cached (warm cache), and it scales well with increasing threads. However, when metadata is not cached (cold cache), the path lookup *stat_cold_N* fails to scale due to lock contention in VFS.

As for crash consistency guaranteeing, file systems usually use journaling and log-structuring techniques. The guaranteeing process suffers from multi-threads competing for log or journal resources. Some NVMM file systems [10], [16] use lock to protect multi-threaded contention. The lock reduces metadata scalability. NOVA [55] allocates logs to each file to improve scalability but cannot solve the contention when multiple threads operating on the shared directory. ScaleFS [47] and Strata [29] allocate log to each thread or CPU core but introducing expensive merge overhead when operating shared directory.

In summary, on one hand, the concurrency control based on VFS lock limits the scalability of metadata operations. We remove VFS lock and adopt resource pre-allocation, atomic write and asynchronous recycling to improve file metadata scalability. On the other hand, the resource contentions during crash consistency guaranteeing process also limit metadata scalability. We extend dentry and use atomic based logging to guarantee crash consistency and improve the scalability.

E. NVMM file systems

Recently, NVMM file systems are proposed to run on non-volatile main memory [9], [10], [16], [54]–[56]. In general, these NVMM file systems remove page cache and general block layer to improve file I/O performance. In this paper, we revisit VFS and co-design metadata optimizations between VFS and physical file systems. Since we focus on metadata path in NVMM file system, we adopt the widely cited NOVA [55], [56] as the design base of physical file system as [28], [51]. As for metadata related design, we keep the following techniques proposed in NOVA, including per-core resource pre-allocate (such as inode and free/journal space) as well as pre-file log space allocation. For data operations, NOVA removes page cache and block layer to improve data I/O performance. NOVA uses Copy-on-Write and log-structuring techniques to guarantee file data crash consistency. DirectFS follows the same data path in NOVA.

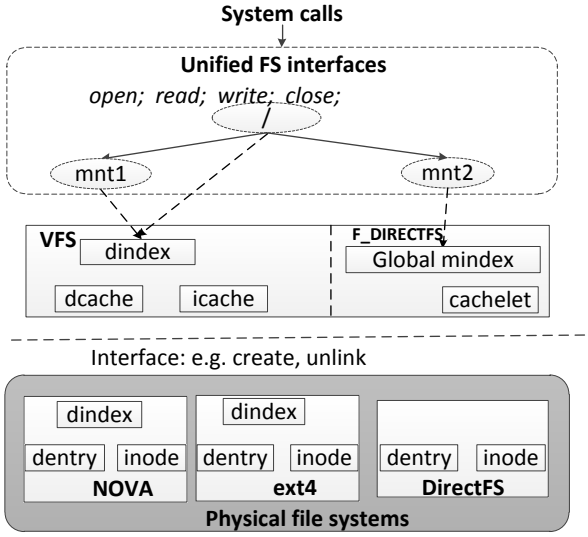


Fig. 4. The architecture and metadata layout of DirectFS. *dindex* represents directory index.

III. DirectFS Design

In order to reduce the performance and scalability overheads of the two-layer metadata management in VFS and NVMM file system, we co-design metadata optimization with VFS and NVMM file system. In this section, we present DirectFS, a metadata optimized NVMM file system by addressing the co-design issues. We first describe the system overview of DirectFS, then we present the detailed design of DirectFS.

A. DirectFS overview

Existing NVMM file systems, such as NOVA [55], [56], ext4-dax [10], PMFS [16] and HiNFS [37], are built under VFS. Figure 4 shows the system architecture and metadata layout of DirectFS compared to existing file systems. For existing file systems, they maintain metadata in both VFS and physical file system and manage them separately. This results in two-layer metadata management overhead, such as (de-)constructing/maintaining dcache and icache in VFS as well as managing dentry and inode in physical file system.

To balance the management overhead and file system performance, we propose metadata optimizations for improving both metadata performance and metadata scalability. Table I shows the key metadata optimizations proposed in DirectFS. We first propose cachelet (Section III-B) and global metadata index (Section III-C) for improving metadata read/write performance. To accelerate metadata read and hard link accessing, we adopt inode quick path and nv_pointer respectively (Section III-D). We also use ".." dentry item in physical file system to support *getcwd* operations.

TABLE I
KEY TECHNIQUES OF METADATA OPTIMIZATION IN DIRECTFS.

Optimizaiton category	Optamized operations	Key techniques
Metadata performance	Metadata read	1. Building cachelet 2. Building global metadata index 3. Building inode quick path
	Hard link	Building nv_pointer
	Metadata write	1. Building Cachelet 2. Building Global metadata index
	Getcwd	Using ".." dentry item
Metadata concurrency control	Creation	1. Pre-allocating dentry and inode 2. Atomic write on concurrency flags
	Deletion	1. Atomic write on concurrency flags 2. Asynchronous recycling
	Lookup	Using RCU-lock and reference count
	Rename	1. Atomic write on concurrency flags 2. Suspending file lookup
Metadata consistency guaranteeing	Creation/ Deletion	1. Atomic write based extend_dentry 2. logging
	Rename	Recording journal

We then scale the concurrency control for four metadata operations, including file creation, deletion, lookup and rename (Section III-E). We pre-allocate dentry and inode to avoid resource contention during file creation. Atomic write is adopted to update concurrency flags for file creation/deletion/rename. We also use RCU-lock [39] and reference counter to mark the file being accessed. We apply asynchronous recycling in the background to prevent the file that is being accessed from being deleted. Since file rename operation spans two files, DirectFS sets rename flag and suspends the file lookup threads to ensure atomicity.

As for scaling crash consistency guarantee, we propose extending the dentry (extend_dentry) and use atomic write based logging to extend_dentry for file creation/deletion (Section III-F). We also use journaling to guarantee the crash consistency of rename operations.

Note that, DirectFS retains the general abstraction provided by VFS and can co-exist with other physical file systems. As shown in Figure 4, a flag **F_DIRECTFS** is added into VFS when DirectFS is mounted[4]. When accessing file system, once this flag is encountered, subsequent metadata operations are performed by DirectFS. Otherwise, the conventional metadata operations process is performed.

B. VFS cachelet

Since the read latency of NVMM is higher than DRAM [20], completely removing VFS cache is not desirable as illustrated in Section II-C. Instead, we build VFS cachelet, a reduced read cache with the minimum size to save management cost and meanwhile retain lookup performance.

Traditionally, VFS cache contains dcache and icache. Since dcache mainly contains file name which can be read from the dentry in NVMM file system directly, we remove dcache directly. The per-file metadata contained in icache mainly includes file properties (e.g. file size and file operation flags), file reference count and read-write lock. Unlike dcache, certain metadata contained in icache provide runtime status of a file and are commonly accessed by most file operations. Thus, by analyzing all metadata related system calls, we keep the most frequently read per-file metadata in VFS cachelet.

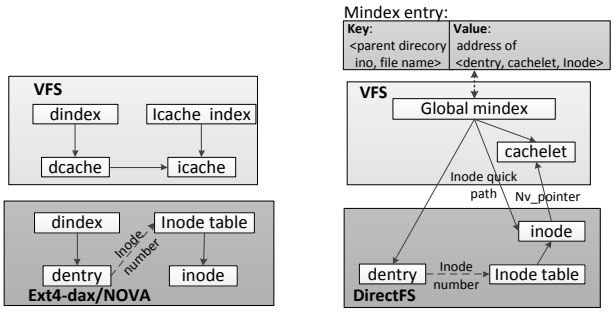
By analyzing all metadata related system calls, we keep the most frequently read per-file metadata in VFS cachelet. For each VFS cachelet entry, it includes file size (8 B), file access permission (2 B), file type (2 B), file link counter (2 B), file group/owner id (8 B), file inode number (8 B) and reference count (8 B). In addition, each VFS cachelet entry uses 2 B for file concurrency control flags (readable/delete/rename flags as illustrated in Section III-E) and 40 B for a mutex lock used in scaling concurrency control (illustrated in Section III-E). For a given physical file system, the addresses of its file operations are originally cached in VFS icache. DirectFS itself directly records these function addresses instead of caching them in each VFS cachelet entry. Finally, the security information to support Linux extensible security module framework [6], access control list [18] and file notification are remained in VFS cachelet for security check. In such doing, each VFS cachelet entry occupies 128 B, which is greatly reduced compared to the original 768 B of a dcache and an icache. This can help to save memory space especially when holding a large volume of files.

C. Global hash based metadata index

Conventionally, metadata is separately indexed in both VFS and physical file systems. This requires efforts to maintain two complex indices, such as hash table in VFS and HTree [3], [10], B-tree [8], [42], radix tree [55], [56] or hash table [37], [50] in physical file systems. Moreover, two separate indices bring twice index search for a file lookup in case of VFS cache miss. Thus, we propose a global hash based metadata index (mindex) to manage both VFS cachelet and metadata in NVMM file system. Hash index outperforms tree-structure index on NVMM in terms of both performance and scalability. Although tree-like index structures support range scan, single-key lookup dominates metadata operations in file systems (e.g., *stat* and *access*). Note that, the system call *readdir* actually fully scans dentry data blocks instead of applying range scan to directory index. Thus, the global hash based mindex is sufficient to support common file system operations.

As shown in Figure 5(b), each entry in the global mindex uses the name of a file and the inode number of its parent directory as the entry key. The entry value contains the NVMM address of the file dentry and the DRAM address of its corresponding cachelet entry. In addition, the mindex entry value includes the NVMM address of the file inode (namely inode quick path as illustrated in Section III-D). Since the global mindex stores indices for all dentries within a whole NVMM file system, the hash table of mindex suffers from table space issue. Rehash is an expensive operation [62] and thus DirectFS applies garbage collection to reclaim less

[4]The **F_DIRECTFS** flag is actually added into the icache of the mounted directory in VFS.



(a) Metadata index structure of VFS. *dindex* refers to directory index. (b) Metadata index structure of DirectFS

Fig. 5. Metadata index structure.

accessed dentries (see Section III-G). This guarantees access performance of the global mindex without suffering from rehash cost.

One can choose to persist the global mindex on NVMM. By doing so, the global mindex can immediately serve queries after system reboot. However, due to the long NVMM read latency, accessing the global mindex from NVMM suffers from expensive NVMM reads and thus reduces file lookup performance. On the contrary, placing the global mindex on DRAM turns to incur the rebuilding overhead. We measure the building cost when opening a directory with 1 million sub-files. It takes 113ms using a single thread. In case of rebuilding the global mindex using 24 threads, it only takes 16ms. Besides, the rebuild operation is only performed when the directory is first accessed. The cost is considered to be acceptable. Therefore, we build global mindex on DRAM.

D. Metadata operations

Metadata read By using the global mindex, DirectFS directly looks up the dentry for a given path component (e.g. first looking up *"example"* for the full path *"/example/foo"*). Once the target dentry *"example"* is found, DirectFS needs to obtain its inode from the dentry. In conventional physical file system, inode is searched in the inode table according to its inode number in dentry (Figure 5(a)). This lookup process usually involves array search (e.g. fixed-sized inode table in ext4 [3] and PMFS) or linked list/B+-tree search (e.g., variable-sized inode table in NOVA and btrfs [42]). This degrades metadata read performance. Instead, DirectFS builds an inode quick path in the global mindex as shown in Figure 5(b) by recording the NVMM address of an inode related with the dentry in mindex entry. Therefore, if the inode address exists in the global mindex when finding a file, DirectFS returns the inode immediately to accelerate lookup performance. In case of accessing a dentry for the first time, DirectFS first finds the inode by using the inode number (fetched from the dentry) in inode table, and then adds the inode address in the global mindex to build the inode quick path. After that, DirectFS searches the next path component *"foo"* in the sub-entries of *"example"*. The above process is repeated for each path component until the last one is reached.

Handling hard link Hard link commonly exists in file system. Assuming file *"/home/foo1"* is a hard link of file *"/home/foo"*. After accessing file *"foo"*, its cachelet entry is built in VFS. Then, file *"foo1"* is subsequently accessed. DirectFS needs to check if there exists the same cachelet entry for the same inode. Conventional VFS avoids establishing two icaches for hard link by searching the whole icaches, which is costly. Unlike VFS, as shown in Figure 5(b), DirectFS adds a non-volatile pointer (namely nv_pointer) in inode to refer to its corresponding cachelet entry after the cachelet entry is created. Each time DirectFS creates a cachelet entry, it checks the nv_pointer of the related inode. A new cachelet entry is created if the nv_pointer is NULL. In such doing, DirectFS avoids creating duplicated cachelet entries for hard link meanwhile without searching the whole VFS cachelet.

DirectFS reclaims VFS cachelet when the reference count of the related file becomes 0. The nv_pointer in the inode is then set to NULL. Note that, when system crashes, the nv_pointer becomes invalid but without being set to NULL. To identify invalid nv_pointers, we embed an epoch number in the nv_pointer. The epoch number is increased by one after crash. DirectFS only considers a nv_pointer with latest epoch number to be valid, which is similar to [13], [51].

Metadata write Metadata write operations in existing file systems are similar to caching update. To provide VFS cache consistency, one needs to modify both VFS and physical file system. Taking *creating "/example/foo"* as an example. VFS first looks up the target directory (*"example"*) (step 1). To prevent concurrent updates, VFS locks the directory *"example"* (step 2). Then, one performs creation operations, including allocating new inode and dentry of file *"foo"* in physical file system (step 3) as well as creating its dcache and icache in VFS (step 4). Meanwhile, one updates inode and icache of the directory *"example"* (step 5) as well as updates its dentry index in both physical file system and VFS (step 6). Until then, VFS releases the lock on the directory *"example"* (step 7) and the file creation ends.

In DirectFS, the above step 1 is executed using the global mindex as described for metadata reads. Then step 3 is directly executed and in step 4, only a VFS cachelet entry is created for file *"foo"*. As for steps 5 to 6, DirectFS update the inode and the related cachelet entry of directory *"example"*. Then, DirectFS only requires to update the global mindex once compared to the original step 6. This helps to reduce index management overhead. DirectFS designs optimization to scale concurrency control in Section III-E, and thus the steps 2 and 7 are not required any more.

Handling *getcwd* system call The system call *getcwd* returns the full working path of the current process. Conventionally, VFS has a pointer in each dcache referring to its parent directory. VFS iterates dcaches of all path components to obtain the full path. In DirectFS, we use the ".." dentry item in physical file system to get the inode of parent directory. We then search current directory inode number in the parent

directory to obtain current directory name. By repeating this process, DirectFS gets the current working path of the process.

E. Scaling concurrency control

Common file operations usually involve multiple metadata changes. Taking file creation for example, when creating a file, file system creates its inode and dentry, and then updates the inode of its parent directory. Due to the two-layer metadata management, existing file systems rely on the directory lock in VFS to guarantee the concurrency control of multi-metadata updates. This limits metadata operation scalability especially for concurrently accessing shared directory. DirectFS adopts selective concurrency control to increase scalability of different metadata operations.

Metadata operations usually involve read operations (such as stat) and write operations. The write operations include creating (mkdir, create), deleting (unlink, rmdir) and updating (chmod, chown) operations. Updating operations only updates one attribution of a single file. Thus, DirectFS executes them using atomic write to increase the concurrency. DirectFS firstly atomically updates inode and then atomically updates cachelet. The concurrent threads only read the cachelet to obtain metadata information, which can read the old or new metadata information. This is correct and can guarantee the atomicity of update operations. Therefore, we mainly consider the scalability of concurrency control for creation, deletion and lookup (read) files.

Creation For creating a file, DirectFS first allocates dentry, inode and VFS cachelet entry (step 1). And it marks the VFS cachelet entry as unreadable by setting the readable flag to 0. After that, DirectFS inserts these structures into the global mindex (step 2). Only after ensuring the metadata of the newly created file is persisted into NVMM (step 3), DirectFS atomically marks the cachelet entry as readable again by setting the readable flag to 1 (step 4). Once the file is marked as readable, the creation operation ends and the file is visible to other concurrent threads. Step 2 ensures that only one thread succeeds when multiple threads create the same file, avoiding generating multiple files with the same name on a directory. In order to provide concurrency control, DirectFS only sets the readable flag of the corresponding cachelet entry to instead of using the directory lock as in conventional VFS. This allows multiple simultaneous metadata operations in the same directory, such as concurrent file creations and lookups.

Note that, file creation also involves metadata allocation, such as inode and dentry. DirectFS pre-allocates inodes to each CPU core as in [55]. Moreover, DirectFS pre-allocates dentries to each CPU core. DirectFS adopts both journaling and log structuring to provide crash consistency (Section III-F). Thus, DirectFS pre-allocates journal space to each CPU core and log space to each directory. In such doing, metadata allocation contention can be reduced.

Deletion For deleting a file, DirectFS first atomically sets the delete flag in VFS cachelet entry of the target file (step 1). Then, DirectFS makes the deletion persistently by using atomic write to inode (step 2). Finally, DirectFS marks the file as invalid by updating VFS cachelet as deleted (step 3). Once a file is marked as invalid, the file cannot be found by other concurrent metadata operations. The delete flag in Step 1 prevents other threads from performing delete operations on the file at the same time. Similar to file creation, setting the delete flag of a cachelet entry allows concurrent metadata operations in the same directory. DirectFS asynchronous recycles the file data of the deleted file in the background only when the file is not accessed by other threads. These help to scale the concurrency control of file deletion in DirectFS.

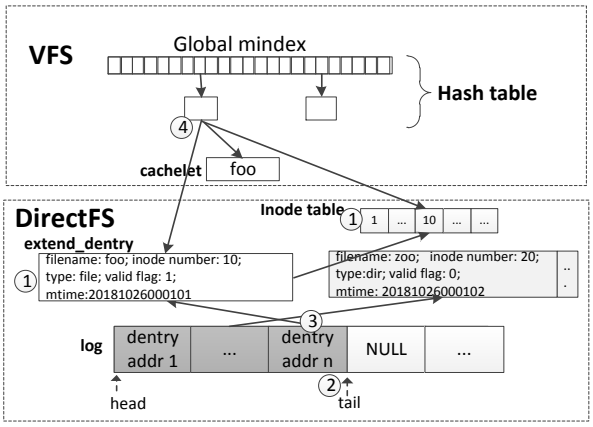


Fig. 6. The design of log-structuring operations.

Lookup When looking up a file/directory, DirectFS applies efficient RCU_lock [39] to allow concurrent deletions towards any component along the path. The concurrent deletion only invalidates the corresponding file without reclaiming inode/dentry/VFS cachelet entry. The invalid file prevents any subsequent lookup. Once DirectFS obtains the target inode, it increases the reference count in cachelet entry before releasing RCU_lock. This indicates that the file/directory is incurring certain operations (e.g., file read or write after lookup). By doing so, concurrent lookup and deletion can be served correctly and the system scalability is increased.

Rename Unlike most metadata updates, *rename* manipulates two files and needs to modify global mindex twice. The atomicity of this operation cannot be guaranteed using atomic write. *rename* first executes operations such as recording journals and allocating new dentry, and finally modifies the status of both source file and destination file. In order to guarantee concurrency control, DirectFS adds a rename flag in VFS cachelet entry to indicate the execution of rename and block concurrent file lookup operations. However, the rename flags of both source file and destination file are only set when modifying their status. In such doing, DirectFS allows concurrent metadata operations during the first stage of rename operation.

F. Scaling crash consistency guaranteeing

In order to ensure crash consistency, DirectFS employs log-structuring technique [31], [43], [55] to record metadata changes. Taking file creation as an example. When creating a file, file system creates its inode and dentry, and then updates the inode of its parent directory by updating hard link and mtime [5]. To provide crash consistency, the newly created dentry and the updated metadata of the parent directory are logged[6].

Existing log structuring adopts lock to guarantee the atomicity of the logging, which limits the scalability of the logging. In DirectFS, as shown in Figure 6, we extend the dentry (named as extend_dentry) to record the updated *mtime* of the parent directory in addition to common dentry information (e.g., file name and inode number). In such doing, DirectFS only needs to record the 8 B address of extend_dentry in log. This can be done using an 8 B atomic write. Besides, we use Compare and Swap (CAS) instruction to atomically allocate log space for each thread. As a result, DirectFS benefits from the lockless log structuring and improves the scalability of guaranteeing crash consistency.

The extend_dentry also records a flag indicating its own validity (Figure 6). This flag is set to 1 when creating file and set to 0 when deleting file. This flag is used in recovery and file deletion. Note that, the extend_dentry does not record the hard link of parent directory. When recovering a directory, DirectFS scans the types of all extend_dentries in the directory, and increases the hard link by 1 when finding a sub-directory.

Creation For creating a file, DirectFS allocates extend_dentry, inode and then persists them (① in Figure 6). Then DirectFS atomically allocates log space using CAS instruction (② in Figure 6) and writes the address of extend_dentry in log and persists it (③ in Figure 6). Then DirectFS updates the mtime, hard link and global mindex of parent directory[7] (④ in Figure 6). If system crashes before persisting the address of extend_dentry, the extend_dentry address in log is NULL. DirectFS ignores all NULL extend_dentry addresses when rebuilding the directory from the log. If system fails after persisting extend_dentry address, DirectFS can restore parent directory by reading extend_dentries. In such doing, DirectFS remains in consistent state.

Deletion For deleting a file, DirectFS directly sets the tuple (valid flag, mtime) in the extend_dentry of the file to be *<0, System.CurrentTimeStamp>* to invalidate the extend_dentry. This tuple occupies 5 B, in which the valid flag takes 1 B and mtime takes 4 B. DirectFS adopts 8 B atomic write for tuple update in extend_dentry to ensure its crash consistency. DirectFS then updates the inode of parent directory by updating its hard link, mtime and global mindex. If system crashes after adopting 8 B atomic write, DirectFS scans the log and updates *mtime* of parent directory for file deletion according to the logged extend_dentry. Besides, DirectFS updates the hard link of parent directory by identifying the type of extend_dentry. Since atomic write incurs little overhead, file deletion in DirectFS scales well.

Other operations For other metadata updates, such as *setattr* and *chmod*, DirectFS first creates a property_entry to record updated metadata. Then, DirectFS uses 8 B atomic write to log the address of property_entry. DirectFS finally updates the actual metadata (e.g., inode). The process also scales well due to the low overhead of atomic write.

Rename Unlike most metadata updates, *rename* manipulates two files which may span over different directories. This involves logging extend_dentries in two directories. Thus, DirectFS adopts per-core journaling to guarantee the crash consistency meanwhile exploiting atomic write based logging to achieve improved scalability. When executing *rename*, DirectFS starts a new transaction in journal and records rename operation. Then, DirectFS performs rename by creating new extend_dentry in destination directory and deleting old one in original directory. After that, DirectFS commits the transaction in journaling.

Algorithm 1 Creating a new file in DirectFS

```
1: Allocate and initialize extend_dentry, inode and VFS cachelet, and marks
   VFS cachelet as unreadable
2: Persist inode and extend_dentry using clflush
3: Insert extend_dentry, inode and VFS cachelet in global mindex
4: new_pos = compare_and_swap(log_tail, log_tail +8)
5: If log_tail update fails due to no enough log space
6:     Lock parent directory using VFS cachelet
7:     Allocate 4 KB new space and assign the start address to new_pos
8:     log_tail = new_pos + 8
9:     Unlock parent directory
10: Persist log_tail using clflush and mfence
11: Record extend_dentry address at new_pos using atomic write
12: Persist extend_dentry address using clflush and mfence
13: Update parent directory inode
14: Update parent directory VFS cachelet
15: Mark the file (VFS cachelet) in global mindex as readable
```

Case study We use file creation as an example to combine both scaling concurrent control and scaling crash consistency guaranteeing. Algorithm 1 describes this procedure. When creating a file, DirectFS first allocates and initializes the extend_dentry, inode and VFS cachelet, marks VFS cachelet as unreadable (line 1), and then persists extend_dentry and inode (line 2, we can use *clflush, clflushopt or clwb*). Then DirectFS inserts extend_dentry, inode and VFS cachelet in global mindex (line 3). Then DirectFS atomically allocates log space by updating the log tail (lines 4-10), logs the extend_dentry address (line 11) and persists it (line 12). After that, DirectFS updates inode (line 13) and VFS cachelet (line 14) of parent directory. Finally, DirectFS marks the file in global mindex as readable (line 15). In case of lacking enough log space, DirectFS acquires the mutex lock in the VFS cachelet of the parent directory and allocates another 4 KB space (line 6-9). The 4 KB space allows 512 log

[5]The hard link of the parent directory is only updated when a sub-directory is created.

[6]Note that, the newly created inode is recorded in inode table before logging the dentry, and is marked as valid after the logging finishes.

[7]DirectFS updates the mtime of parent directory only when the new mtime is later than the mtime of parent directory. This is because the concurrent threads may have updated the parent directory with a later mtime.

entry allocations, which avoids frequent allocation and lock contention.

G. Garbage collection

DirectFS executes garbage collection to reclaim spaces occupied by less-frequently accessed global mindex entires and the deleted files. Currently, DirectFS adopts a simple reference-based eviction policy for evicting global mindex entries. We use the reference count of a directory as the indicator of its hotness. Once the reference count of a directory becomes 0, we add it into a recycle queue for reclaiming. The recycle queue is processed with a first-in first-out policy as a FIFO eviction [53]. However, we also provide second chance to the directory if it is re-accessed again by removing it from the recycle queue.

DirectFS uses asynchronous recycling to reclaim spaces of deleted files, including invalid entries (extend_dentry, property_entry)/inodes as well as log spaces. DirectFS starts to reclaim invalid entries/inodes when the number of invalid entries in a directory exceeds certain threshold (e.g., 100 dentries in this paper) or the reference count of a directory becomes 0 (no thread access). On the other hand, since the log only records addresses of extend_dentries, each log entry occupies 8 B. To increase the reclaiming efficiency, DirectFS starts to reclaim log space only when the summed space of invalid log entries reach 4 KB. DirectFS copies all valid log entries into a new and compacted log space. Then DirectFS replaces the old log with the new one.

H. Recovery

In case of remount after a normal shutdown, DirectFS simply restores all consistent metadata (e.g. pre-allocated spaces and extend_dentry) without scanning journals and logs. During the recovery after system failure, DirectFS first scans per-core journal to apply any uncommitted transactions (mainly for *rename* operations). Then, DirectFS starts a recovery thread on each CPU core and scans log for each valid inode pre-allocated to this core. For example, DirectFS can apply the latest *mtime* to a corresponding directory and recover *hard link* as mentioned in Section III-E. By counting recorded extend_dentry, DirectFS can reclaim any unused extend_dentry to avoid space leakage. This recovery process can be executed in parallel on each CPU core. Our evaluation shows that DirectFS takes 10 ms to identify all allocated dentries for 10 M files by using 24 threads.

IV. Implementation

DirectFS is implemented based on NOVA [55] in Linux kernel 4.18.8. To show the metadata cache optimization for other NVMM file systems, including VFS cachelet and global mindex, we also implement metadata cache optimization in DirectFS for PMFS [16]. Since DirectFS is a metadata optimized file system, we can share data path and data structure of NOVA but implement all proposed metadata techniques. Similar to NOVA, DirectFS removes page cache and generic block layer. For data consistency, DirectFS updates data by using Copy-on-Write, and updates metadata by using log-structuring technique. Besides, DirectFS uses RCU-lock [39] to support multiple threads reading a shared file and a mutex lock in VFS cachelet to support concurrent writing a shared file.

DirectFS implements the global mindex using chained hash table. Each bucket in the hash table is a linked list for solving hash collisions [40]. Thus, the global mindex supports concurrent writes among buckets. Each bucket allows concurrent reads but exclusive write. The number of indexed files depends on the chain length. For example, the hash table can index 256 K files when the chain length is 1 (using 2MB memory space). To increase the number of indexed files, one can increase the chain length but at the cost of reduced lookup performance. Currently, DirectFS does not limit the chain length. However, DirectFS adopts the eviction policy (see Section III-G) to balance the lookup performance and hash table space.

In total, implementing DirectFS requires total 6,076 lines of C codes. Note that, our proposed metadata techniques in DirectFS can also be implemented in any other NVMM file systems, such as PMFS and HiNFS.

However, DirectFS currently has some limitations. DirectFS requires the first several member attributes of dentry/inode structures to be fixed, such as file name, mtime and file size. These attributes are co-managed by both VFS and physical file system. Fixing these member attributes allows VFS and physical file system to directly access them. However, the left member attributes are free to any specific file system. Secondly, DirectFS mainly focuses on building reduced metadata cache. As for sub-mounting another file system, DirectFS can support another DirectFS, but does not allow VFS-based file systems (such as ext4-dax) to be sub-mounted. We leave this feature to be implemented as the future work.

V. Evaluation

In this section we evaluate DirectFS and answer the following questions:

1) How does DirectFS perform against conventional NVMM file systems with VFS? How do our proposed metadata optimizations affect the performance of DirectFS, including inode quick path and global hash based mindex?
2) How is DirectFS scaled when multiple threads update metadata?

A. Experimental setup

Compared systems We compare DirectFS with ext4-dax [10] (using default data order mode), PMFS [16], and NOVA [55]. All these file systems are kernel-based file systems with VFS. We also implement metadata cache optimization on PMFS and NOVA, which can support basic system calls. We show their performance in Section V-B1. We run all experiments in CentOS 7.6.1810 with Linux kernel 4.18.8 by porting PMFS and NOVA to the same kernel. All experimental results are the average of at least 3 runs.

TABLE II
THE EXECUTION TIME (US) OF SYSTEM CALLS ON DIFFERENT FILE SYSTEMS. "NQP" REFERS TO METADATA CACHE OPTIMIZATION WITHOUT INODE QUICK. "_2" AND "_3" REFER TO EVALUATION RUNNING IN TWO-LEVEL DIRECTORY AND THREE-LEVEL DIRECTORY RESPECTIVELY. "_C" INDICATES WARM CACHE.

Time(us)	PMFS	PMFS_Nnqp	PMFS_N	NOVA	NOVA_Nnqp	NOVA_N	DirectFS
stat	37.5	1.2	1.1	4.5	2.2	2.1	2.1
stat_2	37.7	1.2	1.2	5.2	2.2	2.2	2.2
stat_3	37.7	1.3	1.3	5.2	2.3	2.3	2.3
stat_c	0.8	0.9	0.9	1.1	1.2	0.8	0.8
stat_2_c	0.9	0.9	0.9	1.0	1.2	0.8	0.8
stat_3_c	0.9	0.9	0.9	1.0	1.3	0.9	0.9
open	63.0	10.9	10.9	12.8	9.5	9.4	8.0
mkdir	59.8	11.2	11.1	14.0	12.0	11.9	10.6
open_c	61.6	10.8	10.4	12.1	9.2	9.0	6.9
mkdir_c	23.0	11.0	11.0	12.6	10.1	10.0	8.7
unlink	29.6	5.2	5.1	11.6	5.6	5.5	3.4
rmdir	48.6	5.8	5.7	15.0	8.1	8.1	4.8
unlink_c	10.4	5.2	5.1	9.1	4.0	3.7	2.3
rmdir_c	29.3	5.7	5.5	10.4	5.7	5.3	2.6

Hardware configurations We conduct all experiments on a server equipped with two Intel(R) Xeon(R) Gold 6271 processors, each having 24 cores (disable hyper-threading) and a shared 33 MB last level cache (LLC). The memory size is 64 GB and the NVMM size is 512 GB (two Optan DC PMM) for each NUMA node. We only run evaluation on NUMA node 0 to avoid the impact of NUMA architecture on performance.

B. Metadata performance

We first evaluate metadata performance of DirectFS against both cold VFS cache and warm VFS cache. In cold cache, all metadata operations require to be served by physical file systems. In contrast, for warm cache, VFS serves the cached metadata. To evaluate the performance effect of inode quick path, we also run metadata cache optimization without inode quick path (namely *nqp*).

1) Microbenchmarks: We use 5 common system calls: *stat*/*open (creating a new file)*/*mkdir*/*unlink*/*rmdir* as micro-benchmarks. We run these system calls (except *stat*) in a single-level directory with operating 10 K files. To evaluate metadata cache optimization when operating multi-level directories, we test *stat* in single-level directory (*stat*), two-level directory (*stat_2*) and three-level directory (*stat_3*) respectively. In this evaluation, we show the performance of metadata cache optimization with existing NVMM file systems. We implement metadata cache optimization, including VFS cachelet and global mindex on PMFS (PMFS_N) and NOVA (NOVA_N) to show the performance of basic system calls. Table II shows the execution time for system call.

For metadata read operations in cold cache (*stat*), PMFS_N and NOVA_N reduce execution time by 96.9% and 53.5% for PMFS and NOVA respectively. The reductions mainly come from the reduced metadata lookup operations in physical file system and the overhead of creating dcache and icache in VFS. When operating multi-level directories (*stat_2* and *stat_3*), metadata cache optimization in DirectFS still outperforms existing file systems. Since PMFS does not build directory index, it performs worst in finding files. After building global mindex for PMFS (PMFS_Nnqp, PMFS_N), NOVA_N and DirectFS performs worse than PMFS. This is because PMFS can quickly locate inode by calculating offset, and NOVA_N and DirectFS locate inode by looking up inode table. In case of warm cache for metadata read operations (*stat_c*), thanks to building global mindex and adding inode quick path, PMFS_N and NOVA_N provide comparable performance for PMFS and NOVA respectively. NOVA_Nnqp does not contain inode quick path, it preforms worse than NOVA_N by 19.0%. PMFS_Nnqp performs similar to PMFS_N because it can quickly locate inode by calculating offset.

For metadata writes(*open*/*unlink*/*mkdir*/*rmdir*), Metadata cache optimization (PMFS_N and NOVA_N) reduces the execution time by 75.3% and 36.8% on average compared to PMFS and NOVA respectively. This is because it performs lookup operations once and reduces cache (de-)construction/maintenance overhead in VFS. Besides, DirectFS recycles file metadata structures (dentry, inode and cachelet) asynchronously when performing *unlink* and *rmdir* operations. Since DirectFS also uses pre-allocation and atomic writes to execute metadata updates, it reduces the execution time by 28.9% on average compared to NOVA_N.

2) Macrobenchmarks: We use 5 widely-used command-line applications running in the Linux source directory [48] to evaluate the end-to-end performance: *find* (searching for "test.c" file), *du -s* (showing whole directory size), *rm -rf* (deleting all files) and *tar -xzf* (decompress and unpack code) and *gzip* compress code. Table III shows execution time of the tested applications.

In case of cold cache, DirectFS outperforms ext4-dax, PMFS and NOVA by 24.1%, 14.3% and 32.8% respectively. This is because DirectFS optimizes metadata lookup operations and reduces cache (de-)construction/maintenance overhead. Besides, DirectFS reduces metadata write operation by using pre-allocates and atomic write. Because PMFS does not build directory index, so it saves the overhead of rebuilding directory index. On the other hand, the average number of files under each directory in Linux source is 12. Thus PMFS performs better than NOVA and ext4-dax for *du* and *find* as shown in Table III.

In case of warm cache, ext4-dax, PMFS and NOVA can perform almost all metadata read operations in VFS. DirectFS provides comparable performance. DirectFS_nqp performs worse than DirectFS due to the lack of inode quick path. For *tar* and *gzip*, DirectFS achieves little improvement as the data I/O occupies most time of the application.

C. Metadata scalability

We show the scalability of metadata operations in DirectFS when multiple threads operating a shared directory. We evaluate file lookup, remove and rename. Since PMFS performs similar performance to NOVA and ext4-dax, we do not show the results of PMFS.

We first use mdtest [22], a metadata benchmark, to evaluate metadata scalability for multiple threads operating a shared directory in case of warm cache (we have similar scalability for cold cache). All threads operate 1 M files and perform

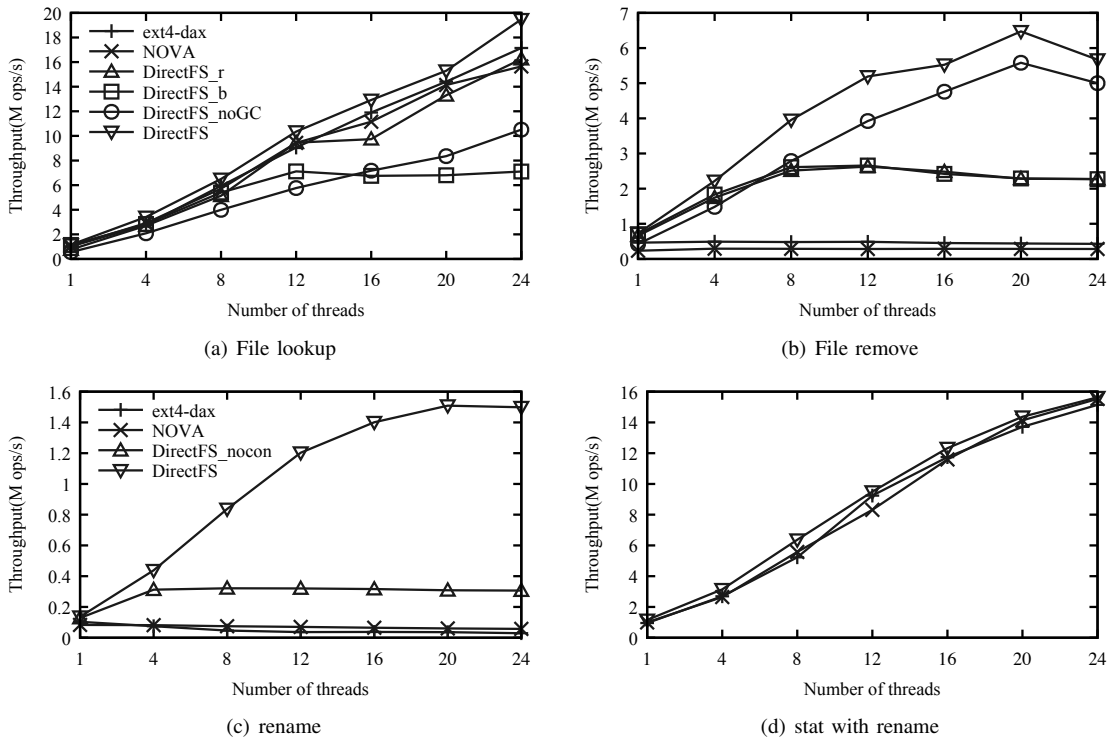


Fig. 7. Metadata scalability

TABLE III
THE EXECUTION TIME (S) OF COMMON COMMAND APPLICATIONS ON DIFFERENT FILE SYSTEMS. "_c" MEANS WARM CACHE.

Time(s)	ext4-dax	PMFS	NOVA	DirectFS_nqp	DirectFS
du	0.42	0.30	0.54	0.38	0.38
tar	8.12	8.93	7.66	7.65	7.63
find	0.25	0.18	0.26	0.17	0.16
gzip	22.10	22.96	21.91	21.41	21.30
rm	0.78	1.03	1.15	0.44	0.43
du_c	0.15	0.14	0.16	0.20	0.14
tar_c	7.60	8.43	7.13	7.14	7.06
find_c	0.10	0.09	0.10	0.09	0.09
gzip_c	21.81	22.94	21.67	21.37	21.26
rm_c	0.64	1.04	1.01	0.35	0.34

metadata operations. To evaluate the performance impact of global mindex, we implement radix tree (DirectFS_r) and B-tree (DirectFS_b) as global mindex separately. To show the GC performance of global mindex in DirectFS, we access 5 M files in the file system before running the evaluation. If the GC operation is not performed (DirectFS_noGC), the length of hash chain increases, reducing file lookup performance.

File lookup For lookup operation in warm cache, ext4-dax and NOVA access metadata in VFS and scale well as show in Figure 7(a). Both DirectFS and DirectFS_r adopt RCU-lock and thus achieve similar scalability. Since the lookup performance of radix tree is worse than that of hash table, DirectFS_r performs a little worse than DirectFS in terms of throughput. The lookup operation in DirectFS_b is protected by rwlock, and thus achieves poor scalability. DirectFS_noGC does not contain GC thread to recycle index entries that are no longer accessed in the global mindex, resulting poor lookup performance.

File remove Since VFS locks the whole directory to limit concurrent metadata access when operating the shared directory, ext4-dax and NOVA fail to scale file remove as shown in Figure 7(b). Compared to fine grained lock in hash table, radix tree and B-tree require locking a sub-tree. Thus, both DirectFS_r and DirectFS_b fail to scale file deletion. DirectFS_noGC shows poor performance due to long file lookup performance. Since DirectFS removes VFS directory lock and reduces logging contention by using atomic write, it provides high concurrency, it provides high scalability for file remove operations. Note that, the performance of DirectFS degrades after reaching 24 threads when deleting files. This is because we run a garbage collection thread in the background to recycle the deleted file (Section III-G). The garbage collection thread co-runs with file deletion threads on the same CPU processor of 24 cores. When reaching 24 deletion threads, the garbage collection thread introduces extra CPU contention and thus degrades the performance.

File rename We evaluate the scalability of *rename* operation by using FXMARK [36] in a shared directory. We also evaluate DirectFS_nocon without optimizing the scalability of consistency (Section III-F), and multiple threads compete for the lock to log. Figure 7(c) shows the results. DirectFS pre-allocates resources (e.g. journal and dentry) to each CPU core and adopts atomic writes to avoid using lock. Thus, DirectFS achieves better scalability than other file systems. Because DirectFS uses rename flag to block other metadata

operations (including other concurrent rename operations) during updating file status, it scales poorly after reaching 20 threads. Meanwhile, DirectFS_nocon stops scaling file rename after reaching 4 threads because of the lock of logging.

Figure 7(d) shows the performance of path lookup in warm cache when there is a concurrent rename thread executing on the same path. The rename lock of VFS does not limit the scalability of file lookup operations. This is because VFS caches all file metadata in case of warm cache, and the path lookup can be served in VFS. Therefore, ext4-dax and NOVA scale well in Figure 7(d). DirectFS instead adds rename flag for parent directory and suspends metadata lookup (Section III-E) to support concurrent rename. This introduces little overhead and DirectFS also scales well as shown in Figure 7(d).

D. Application scenarios

We now evaluate DirectFS under three application scenarios. We first evaluate DirectFS reading and writing 1 K small files (ranging from 1 KB to 32 KB). This is typical thumbnail/profile picture workload in Internet services (e.g. social network [5] and e-commerce). Note that, we run DirectFS using one thread here and multi-thread runs can execute in same directories without the scalability limitation (shown in Section V-C). This application scenario involves open/read/write/close metadata operations. Figure 8 shows the results. We do not show PMFS as it does not build directory index and performs poor in locating files. DirectFS improves the throughput by 35.6% and 38.3% compared to ext4-dax and NOVA for 1 KB files. When file size increases to 32 KB, DirectFS still improves throughput by 35.0% and 17.7% compared to ext4-dax and NOVA respectively. These improvements come from reducing cache (de)construction/maintain and lookup operations in physical file system. Besides, DirectFS uses pre-allocate and atomic write operations to improve metadata performance.

We then run filebench [1], a file system benchmark that simulates a large variety of workloads by specifying different models. We run fileserver and varmail and the test results are shown in Figure 9. Since these applications usually run with multi-thread, we test them with different threads. We set varmail with 10 K files, the average file size is 16 KB, IO size is 1 MB and the read/write ratio is 1:1. DirectFS obtains 27% improvement for single thread. This is because DirectFS reduces metadata cache update and lookup overheads. Besides, DirectFS optimizes the metadata update operations by using pre-allocates and atomic write. For 16 threads, DirectFS outperforms NOVA by 66.0%. This is because DirectFS further optimizes the scalability of metadata operations. We set fileserver with 10 K files, the average file size is 128 KB, IO size is 1 MB and the read/write ratio is 1:2. For single threads, DirectFS outperforms NOVA by 42.3%. As the number of threads increases, the throughput difference between DirectFS and NOVA becomes smaller, this is because the bandwidth of NVMM limits the performance.

E. Latency breakdown

Figure 10 shows the metadata execution time normalized to ext4-dax in case of cold VFS cache. Similar to Figure 1, we also divide metadata operations into *VFS_metadata* (metadata cache (de-)construction/maintenance) and *PFS_metadata* (including metadata read/write in physical file system, directory index or global mindex rebuilding, and logging). Since DirectFS executes one-time lookup, we do not divide lookup here. Note that, lookup in ext4-dax and NOVA includes operations in both VFS and physical file system. DirectFS adopts global hash based mindex to locate metadata. Hash index performs more efficiently than Htree (ext4-dax) and Radix tree (NOVA) in terms of point query. As shown in Figure 10, both mechanisms help DirectFS to reduce the lookup time for all metadata operations. DirectFS only needs to allocate and manage a small VFS cachelet. Thus, the latency of metadata cache (*VFS_metadata*) becomes smaller in DirectFS.

For physical metadata operations (*PFS_metadata* in Figure 10), both NOVA and DirectFS require re-building directory index or global mindex in DRAM when accessing existing files (small file reads, *du* and *rm*). Instead, ext4-dax only requires reading persistent directory index into page cache. As a result, both NOVA and DirectFS spend more time on physical metadata operations than ext4-dax for small file reads and *du*. The global hash based mindex in DirectFS rebuilds faster than radix tree does in NOVA. This results in less physical metadata operation times in DirectFS than NOVA. For file remove operations (*rm*), DirectFS only marks deleted dentries as invalid. Instead, NOVA and ext4-dax needs to reclaim dentry when deleting file. This helps DirectFS to reduce physical metadata operation time. Therefore, DirectFS performs less physical metadata operation times than ext4-dax for *rm*. For writing small files, all three file systems require updating directory index. However, ext4-dax requires first writing metadata into page cache and then flushing to NVMM, which adds extra latency to physical metadata operations. DirectFS builds global mindex which is faster than NOVA. Besides, DirectFS optimizes metadata update by using pre-allocate and atomic write. Thus, DirectFS consumes less time on physical metadata operations.

VI. Related work

NVMM based file system Recently, a number of research efforts build file systems on NVMM. Most focus on optimizing data path by removing block layer and page cache [9], [10], [16], [37], [45], [54]–[56], [60]. Among these works, NOVA-Fortis [56] further provides strong consistency. HiNFS [37] adds write buffer to reduce the impact of long NVMM write latency. In contrast, DirectFS mainly focuses on optimizing metadata path. SoupFS [13] targets to reduce the critical path latency by delaying almost all synchronous metadata updates. SPFS builds a single-level persistent memory file system by completely removing metadata cache but turns out to performing worse than existing NVMM file systems [61]. In contrast, DirectFS optimizes metadata cache and outperforms state-of-the-art NVMM file systems.

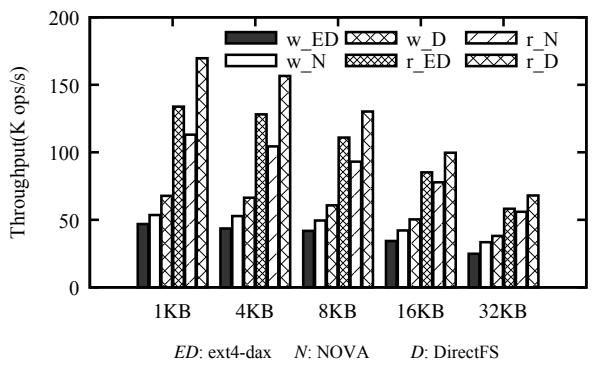


Fig. 8. Throughput of operating small files in terms of files processed per second

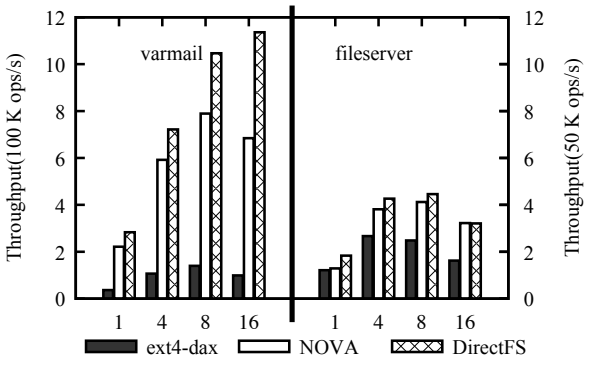


Fig. 9. The throughput of varmail and fileserver in terms of files processed per second

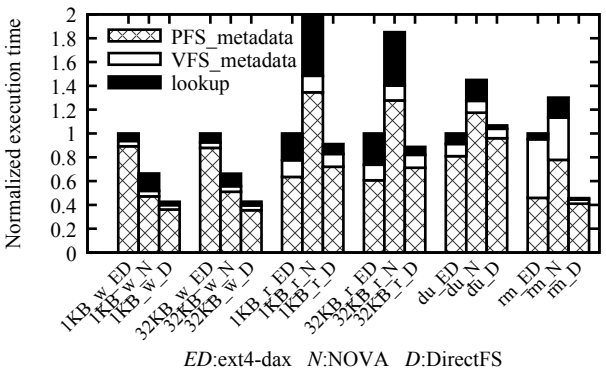


Fig. 10. Latency breakdown

On the other hand, a few research works build user-level file systems to directly access data in user space avoiding trapping into the kernel [12], [23], [26], [29], [50]. These file systems do not rely on VFS to cache metadata. However, Strata [29] itself provides read-only metadata cache in user space, and still needs to maintain metadata cache consistency between user space process and Strata kernel. Aerie [50] manages metadata through a trusted user-level FS service, which suffers from extra inter-process communication overhead. SplitFS [23] and ZoFS [12] do not optimize metadata operations. The approaches of DirectFS can be used in these works. DevFS [26] also provides file system lib in user space without interacting the kernel but targets to build file system inside solid state drive (SSD).

Metadata optimization A number of research works optimize metadata operations. For example, TableFS [41] uses log-structured merge tree to accelerate metadata writes. ReconFS [35] buffers directory tree in DRAM to reduce SSD write amplification. BetrFS [21] provides write-optimized index to reduce random metadata operation overhead. In contrast, DirectFS is built on byte-addressable NVMM without small write issues on block device. To reduce path lookup overhead, previous works [14], [49] replace component-by-component translation by adding a cache on VFS. Existing NVMM file systems mainly optimize metadata operations in physical file systems [9], [13], [16], [54], [55]. Unlike these works, DirectFS mainly focuses on building a small metadata cache and uses unified management to reduce metadata cache overhead. ByVFS [51] removes dcache in VFS to improve path lookup performance. However, it retains icache in VFS and does not take metadata scalability into account. Full path lookup proposed in previous works [14], [21], [33], [52], [58], [59] can also be applied to DirectFS.

File system scalability To improve scalability, SpanFS [25] partitions physical file system to domains to reduce data contention, IceFS [34] groups files or directories into physically isolated containers, and meanwhile Multilane [24] partitions the entire IO stack, including VFS, physical file system and driver. ScaleFS [47] builds an in-memory file system to support concurrent data structure and then merges in-memory updates to on-disk file system. NOVA [55], [56] pre-allocates free spaces and maintains separate logs for each file to improve concurrency. Previous work [46] allows concurrent updates on data structures and parallelizes I/O operations for journaling file systems. DirectFS is different from these works in that it targets NVMM file system and handles scalability issue by carefully resolving resource contentions. pNOVA [28] accelerates the parallel writing and reading for individual files. The proposed techniques can be used in DirectFS to improve single file I/O concurrency.

Index structure Previous works mainly focus on reducing consistency overhead and space consumption, and improving concurrency control for using different indexes on NVMM [7], [11], [32], [38], [40], [57], [62]. Existing NVMM file systems employ radix tree [55], hash table [13], [50], and Htree [10] for each directory index. Instead, DirectFS adopts chained hash table as global mindex to support fast lookup and improve scalability.

VII. Conclusion

Metadata is an important part of file systems. In this paper, we revisit virtual file system to find two main sources that limit metadata performance and scalability. We thus explore to build a metadata optimized file system for NVMM-DirectFS.

In DirectFS, VFS cachelet is first co-designed with VFS and NVMM file system to reduce conventional VFS cache management overhead meanwhile retaining file lookup performance. DirectFS then adopts a global hash based metadata index to manage both VFS cachelet and metadata in NVMM file system. This helps to avoid duplicated index management in conventional VFS and physical file system. Besides, DirectFS adopts fine-grained flags and atomic write for concurrency control and consistency guarantee to improve metadata scalability. We implement DirectFS in Linux kernel and evaluate it against state-of-the-art NVMM file systems. The results show DirectFS improves metadata access performance as well as scales for concurrent operations.

ACKNOWLEDGES

We thank the anonymous reviewers and our shepherd Bing Xie for their insights and valuable comments. We also thank Liuying Ma, Shukai Han and Wanling Gao for their suggestions. This work is supported by National Key Research and Development Program of China under grant No.2016YFB1000302, Strategic Priority Research Program of the Chinese Academy of Sciences under grant No. XDB44000000, Beijing Natural Science Foundation under grant No. L192038, and Youth Innovation Promotion Association CAS.

REFERENCES

[1] Filebench 1.4.9.1. https://github.com/filebench/filebench/wiki.

[2] A. M. Caulfield A. Akel, R. K. Gupta T. I. Mollov, and S. Swanson. Onyx: A protoype phase change memory storage array. *In Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems, HotStorage'11, pages 2–2*, 2011.

[3] Suparna Bhattacharya Avantika Mathur, Mingming Cao, Alex Tomas Andreas Dilger, and Laurent Vivier. The new ext4 filesystem: current status and future plans. *In Proceedings of the Linux Symposium, Vol. 2.*, 2007.

[4] IG Baek, MS Lee, S Seo, MJ Lee, DH Seo, D-S Suh, JC Park, SO Park, HS Kim, IK Yoo, et al. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*, pages 587–590, 2004.

[5] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 47–60. USENIX Association, 2010.

[6] C. Cowan C. Wright, J. Morris S. Smalley, and G. K. Hartman. Linux security modules: General security support for the linux kernel. *In USENIX Security Symposium*, 2002.

[7] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[8] Dave Chinner. xfs: Dax support. https://lwn.net/Articles/635514/, 2019.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Coetzee Derrick. Bpfs: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. ACM, 2009.

[10] Jonathan Corbet. Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, September 2014.

[11] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386. USENIX Association, 2018.

[12] Mingkai Dong, Heng Bu, Yi Jifei, Dong Benchao, and Chen Haibo. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 478–493, New York, NY, USA, 2019. ACM.

[13] Mingkai Dong, Institute of Parallel Haibo Chen, and Shanghai Jiao Tong University Distributed Systems. Soft updates made simple and fast on non-volatile memory. *ATC*, 2017.

[14] Dan Duchamp. Optimistic lookup of whole nfs paths in a single operation. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 10–10. USENIX Association, 1994.

[15] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. Persistent memory file system. https://github.com/linux-pmfs/pmfs.

[16] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15. ACM, 2014.

[17] Shuai Ma Fan Yang, Junbin Kang and Jinpeng Huai. A highly non-volatile memory scalable and efficient file system. *2018 IEEE 36th International Conference on Computer Design*, 2018.

[18] A. Grunbacher. Posix access control lists on linux. *USENIX ATC, 2003.*, 2003.

[19] Intel. *Intel and Micron produce breakthrough memory technology*, https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/ edition.

[20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[21] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 301–315. USENIX Association, 2015.

[22] Joinbent. mdtest. https://github.com/MDTEST-LANL/mdtest.

[23] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 494–508, New York, NY, USA, 2019. ACM.

[24] Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai. Multilanes: Providing virtualized storage for os-level virtualization on many cores. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 317–329. USENIX, 2014.

[25] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. Spanfs: A scalable file system on fast storage devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 249–261. USENIX Association, 2015.

[26] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true direct-access file system with devfs. In *16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 241–256. USENIX Association, 2018.

[27] Linux kernel. Path walking and name lookup locking. https://elixir.bootlin.com/linux/v4.3/source/Documentation/filesystems/path-lookup.txt.

[28] June-Hyung Kim, Jangwoong Kim, Hyeongu Kang, Chang-Gyu Lee, Sungyong Park, and Youngjae Kim. pnova: Optimizing shared file i/o operations of nvm file system on manycore servers. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, pages 1–7, New York, NY, USA, 2019. ACM.

[29] 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 '17, pages 460–477. ACM, 2017.

[30] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug 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. ACM, 2009.

[31] Changman Lee, Dongho Sim, Joo-Young 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'15, pages 273–286. USENIX Association, 2015.

[32] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. Wort: Write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 257–270. USENIX Association, 2017.

[33] Paul Hermann Lensing, Toni Cortes, and André Brinkmann. Direct lookup and hash-based metadata placement for local file systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 5:1–5:11. ACM, 2013.

[34] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 81–96. USENIX Association, 2014.

[35] Youyou Lu, Jiwu Shu, and Wei Wang. Reconfs: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 75–88. USENIX Association, 2014.

[36] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85. USENIX Association, 2016.

[37] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 12:1–12:16. ACM, 2016.

[38] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386. ACM, 2016.

[39] Jonathan Walpole Paul E. McKenney. What is rcu, fundamentally? https://lwn.net/Articles/262464/, 2007.

[40] Yu Hua pengfei Zuo. A write-friendly hashing scheme for non-volatile memory systems. *Proceedings of the33st Symposium on Mass Storage Systems and Technologies(MSST 2017)*, 2017.

[41] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156. USENIX, 2013.

[42] Ohad Rodeh, Josef Bacik, and Mason Chris. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[43] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[44] M. Breitwisch S. Raoux, G. Burr, R. Shelby C. Rettner, Y. Chen, D. Krebs M. Salinga, H. L. Lung S.-H. Chen, and C. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development, 52(4.5):465–479*, 2008.

[45] E. H. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang. A new design of in-memory file system based on file virtual address framework. *IEEE Transactions on Computers*, 65(10):2959–2972, Oct. 2016.

[46] Yongseok Son, Sunggon Kim, Heon Y. Yeom, and Hyuck Han. High-performance transaction processing in journaling file systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 227–240. USENIX Association, 2018.

[47] Rasha Eqbal Srivatsa S. Bhat, M. Frans Kaashoek Austin T. Clements, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. *In Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17).*, 2017.

[48] Linus torvalds. Linux kernel source tree. https://github.com/torvalds/linux.

[49] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 441–456. ACM, 2015.

[50] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael 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. ACM, 2014.

[51] Ying Wang, Dejun Jiang, and Jin Xiong. Caching or not: Rethinking virtual file system for non-volatile main memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, 2018.

[52] Brent Welch. A comparison of three distributed file system architectures: Vnode, sprite, and plan 9. *Sprite, and Plan 9. Computing Systems*, 7:175–199, 1994.

[53] Wikipedia. Cache replacement policies. https://en.wikipedia.org/wiki/Cache_replacement_policies.

[54] Xiaojian Wu and A. L. Narasimha Reddy. Scmfs: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11. ACM, 2011.

[55] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies*, FAST'16, pages 323–338. USENIX Association, 2016.

[56] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496. ACM, 2017.

[57] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng 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. USENIX Association, 2015.

[58] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Optimizing every operation in a write-optimized file system. In *14th USENIX Conference on File and Storage Technologies*, FAST'16, pages 1–14. USENIX Association, 2016.

[59] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The full path to full-path indexing. In *16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 123–138. USENIX Association, 2018.

[60] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, Boston, MA, 2019. USENIX Association.

[61] Deng Zhou, Wen Pan, Tao Xie, and Wei Wang. A file system bypassing volatile main memory: Towards a single-level persistent store. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, CF '18, pages 97–104. ACM, 2018.

[62] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476. USENIX Association, 2018.