Wang Y, Jia WQ, Jiang DJ *et al.* A survey of non-volatile main memory file systems. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 38(2): 348-372 Mar. 2023. DOI: 10.1007/s11390-023-1054-3

A Survey of Non-Volatile Main Memory File Systems

Ying Wang (王　盈), *Member, CCF, ACM, IEEE*
Wen-Qing Jia (贾文庆), *Student Member, CCF, ACM, IEEE*
De-Jun Jiang (蒋德钧), *Member, CCF, ACM, IEEE*, and
Jin Xiong (熊　劲), *Senior Member, CCF, Member, ACM, IEEE*

State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China

Research Center for Advanced Computer Systems, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China

University of Chinese Academy of Sciences, Beijing 100049, China

E-mail: wangying2023@ict.ac.cn; jiawenqing@ict.ac.cn; jiangdejun@ict.ac.cn; xiongjin@ict.ac.cn

Received October 9, 2020; accepted March 12, 2023.

Abstract　Non-volatile memories (NVMs) provide lower latency and higher bandwidth than block devices. Besides, NVMs are byte-addressable and provide persistence that can be used as memory-level storage devices (non-volatile main memory, NVMM). These features change storage hierarchy and allow CPU to access persistent data using load/store instructions. Thus, we can directly build a file system on NVMM. However, traditional file systems are designed based on slow block devices. They use a deep and complex software stack to optimize file system performance. This design results in software overhead being the dominant factor affecting NVMM file systems. Besides, scalability, crash consistency, data protection, and cross-media storage should be reconsidered in NVMM file systems. We survey existing work on optimizing NVMM file systems. First, we analyze the problems when directly using traditional file systems on NVMM, including heavy software overhead, limited scalability, inappropriate consistency guarantee techniques, etc. Second, we summarize the technique of 30 typical NVMM file systems and analyze their advantages and disadvantages. Finally, we provide a few suggestions for designing a high-performance NVMM file system based on real hardware Optane DC persistent memory module. Specifically, we suggest applying various techniques to reduce software overheads, improving the scalability of virtual file system (VFS), adopting highly-concurrent data structures (e.g., lock and index), using memory protection keys (MPK) for data protection, and carefully designing data placement/migration for cross-media file system.

Keywords　non-volatile main memory (NVMM), file system, performance, scalability, crash consistency, data protection, crossmeida

1　Introduction

Emerging non-volatile memory (NVM)[1–4] is byte-addressable and non-volatile, which can be used as a memory-level storage device (non-volatile main memory, NVMM) to store file data. Compared with traditional block devices, such as solid state drives (SSDs) and hard disk drives (HDDs), NVMM provides lower latency and higher bandwidth. The performance bottleneck of file systems is no longer the slow storage devices, and software overhead becomes the major factor affecting the performance[5, 6]. This motivates a number of research efforts to optimize or redesign file systems for NVMM over the past decade.

However, designing a high-performance and cost-effective NVMM file system is non-trivial. Firstly, reducing software overhead is important. Traditional block-based file systems (such as ext3, ext4[7], xfs[8], zfs①, btrfs[9], f2fs[10] and [11–15]) adopt a deep software stack to optimize the file I/O on block devices, in-

Survey

This work is supported by the Major Research Plan of the National Natural Science Foundation of China under Grant No. 92270202, and the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No. XDB44030200.

①S.Microsystems. ZFS. http://www.opensolaris.org/os/community/zfs, Mar. 2022.

cluding page cache, block layer and I/O scheduler layer. This deep software stack provides caching and I/O scheduling to reduce data operations on slow disks and accelerate file accessing. However, NVMM is byte-addressable and provides lower access latency than block devices. The deep software stack can reduce the performance of NVMM file systems. Removing these software stacks directly, such as page cache, can improve the performance, but will reduce read performance. This is because the read latency of real NVMM hardware—Intel Optane DC persistent memory module (Optane PMM)[②] is longer than that of DRAM (dynamic random access memory). Therefore, re-architecting an NVMM aware software stack is crucial to NVMM file systems. Besides, existing index structures in traditional file systems, such as B-tree for the index file data block, are designed based on block devices and cannot take the advantage of NVMM being byte-addressable. Therefore, we should consider the problems above to reduce software overheads of NVMM file systems.

Secondly, the increasing number of CPU cores allows multiple threads to access file systems simultaneously and NVMM supports high concurrent accesses[16–18]. Therefore, the NVMM file system should support high concurrent operations. However, the traditional block-based file systems are designed based on deep software stacks, such as VFS and page cache[19–22], which limits the concurrency of the file system. Some studies improve concurrency by using partition or storing data in memory temporarily and then migrating to storage device. These studies bring additional software operations, such as data merging[23, 24] and garbage collection[20], which increase the latency on NVMM file systems. Therefore, NVMM file systems should be reconsidered to improve scalability and avoid long latency.

Thirdly, guaranteeing crash consistency is a fundamental requirement for file systems. Modern CPU and memory systems may reorder data store instructions to memory, which may result in crash inconsistency[18]. However, flushing data from the CPU cache to NVMM sequentially harms the performance[12]. Besides, traditional crash consistency techniques, such as copy-on-write and log-structuring[25], are designed for block devices that write the data at block granularity, which results in write amplification on NVMM file systems, and increases the amount of the data that needs to be written and further reduces performance. How to guarantee consistency and reduce the overhead of guaranteeing consistency is an issue that NVMM file systems need to consider.

Fourthly, data protection and NVMM endurance are needed for NVMM file systems. Since NVMM can be directly attached on the memory bus, threads can access NVMM as regular memory. Some bugs from unrelated threads may generate stray writes and result in data errors on NVMM file systems. Besides, media errors from NVMM can cause incorrect values. NVMM file systems need techniques to avoid errors and to detect and correct errors after they occur. Existing studies provide software (e.g., error-correction codes, checksum) and hardware (e.g., Intel memory protection keys) techniques to solve these problems. However, these techniques may degrade system performance or require hardware support. Choosing an efficient and appropriate write protection technique is important for NVMM file systems. Besides, NVMMs have limited endurance[17] and some cells may wear out faster than others. In NVMM file systems, different data has different update frequencies. Updating some cells or blocks frequently may cause permanent loss of data or even file corruption. Some studies[26, 27] focus on metadata of file systems, which are often updated in the fixed location. How to guarantee NVMM cells to be evenly worn is an issue that should be considered when designing NVMM file systems.

Finally, NVMM is more expensive than block devices[23, 28]. Building file systems only on NVMM is too expensive especially for a data center. Considering the cost, capacity, performance and accessing mode of different storage media and building a cross-media file system are more desirable. How to place data and perform data migration on NVMM and block devices (SSD and disk) to build a high-performance, cost-effective file system is a challenging issue.

There have been already a few surveys that summarize the impacts of NVMM on storage systems. Wu *et al.*[29] focused on Phase Change Memory (PCM) as NVMM, explored the challenges of adopting PCM as storage and main memory, and summarized and classified existing solutions. Chen[30] summarized device-level optimization techniques for NVMM. Mittal and Vetter[31] focused on using software and system-level

[②]Intel Corporation. Revolutionary memory technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html, Mar. 2022.

techniques to exploit the advantages and mitigate the disadvantages of NVMMs. However, these researches are based on assumptions about NVMM performance, which are different in certain aspects with the real NVMMs.

To reveal the performance of previous NVMM file systems on the real hardware Optane PMM, we survey the techniques of NVMM file systems and analyze their advantages and disadvantages in this paper. Izraelevitz *et al.*[16] tested the performance of Optane PMM and summarized the device characteristics, which provide us suggestions for designing NVMM file systems. Puglia *et al.*[32] derived the main concerns and challenges currently being studied and discussed in the academia and industry for NVMM file systems, as well as the trends and solutions being proposed to address them. However, they focused on theoretical contents rather than experimental results and techniques. We make a deeper introduction to the software techniques of NVMM file systems, analyze these techniques based on the performance of real hardware Optane PMM, and provide some suggestions.

In this paper, we survey 30 NVMM file systems and focus on the main problems on them. The main contributions of the paper are as follows.

1) We summarize the challenging issues of NVMM file systems and survey existing techniques, including reducing software overhead, improving concurrency, guaranteeing crash consistency, protecting file data, and building cost-effective cross-media NVMM file systems. Besides, we summarize the advantages and limitations of existing techniques.

2) We propose possible optimizations and research directions of NVMM file systems with real hardware NVMM.

The remainder of this paper is organized as follows. Section 2 presents the characteristics of NVMM and introduces file systems. Sections 3–6 survey the techniques to reduce software overhead, improve scalability, guarantee consistency, and protect data respectively. We discuss building a cross-media file system to reduce cost in Section 7. Finally, we conclude this paper in Section 8.

2 Background

2.1 Non-Volatile Main Memory

The emerging non-volatile memory (NVM) provides low latency and persistent storage, which can be used as a storage device to improve file system performance. The existing NVM devices include ReRAM[2], STT-RAM[3], Phase Change Memory (PCM)[1, 4] and 3D Xpoint③. Although these devices use different techniques, they have the following advantages: providing access latency close to DRAM, and supporting high concurrent accesses and persistent storage.

Currently, there are two ways to use NVM. One is for external memory-level storage devices, providing block access interfaces and connecting to the motherboard by using PCIe bus (such as Optane DC SSD, Optane SSD). The other is for memory-level storage devices (NVMM), connecting to the memory bus and providing load/store interfaces.

Table 1 shows the read and write (R/W) latency, R/W bandwidth, capacity and price of DRAM, NVMM (PCM, Optane PMM), SSD, and disk[5, 16, 28, 33–36] and the data is from Intel's website④ and jd.com in July 2019. PCM is used as the basic device for academic research because of its high density, high scalability and mature techniques. PCM has similar read latency, 3x– 5x write latency and 1/8 bandwidth of DRAM. A lot of researches[6, 18, 37–39] have considered

Table 1. Comparison of NVM Technologies with DRAM, SSD and Disk

Memory	R/W Latency	R/W Bandwidth (GB/s)	Volatility	Product Capacity	Price ($/GB)
DRAM	60 ns/60 ns	20	Yes	64 GB	4.49
PCM	50 ns–70 ns/150 ns–1 000 ns	7.8	No	–	–
Optane DC PMM	305 ns/81 ns	6–7/2–3	No	512 GB	5.34
Optane DC SSD	10 µs/10 µs	2–3	No	1.5 TB	1.28
NVMe SSD	120 µs/30 µs	2/0.5	No	8 TB	0.21
Disk	5 ms/5 ms	0.1	No	16 TB	0.03

Note: – means there is no data.

③Intel. Intel and Micron produce breakthrough memory technology. https://www.intc.com/news-events/press-releases/detail/324/intel-and-micron-produce-breakthrough-memory-technolog, Mar. 2022.

④https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html, Mar. 2022.

optimizing the file systems on PCM. However, due to lacking real PCM products, the existing work evaluates the performance using an emulator[18, 23, 40]. Recently, Intel has provided the real NVMM device Optane™ DC Persistent Memory Module (Optane PMM) based on 3D XPoint that can be used for file systems evaluations and replace the emulator. The performance of Optane PMM in Table 1 comes from experiments and existing work[16, 41]. We can see that Optane PMM shows 3x slower read latency and similar write latency to DRAM. Besides, its read and write bandwidth are about 1/3 and 1/8 of those of DRAM respectively. The performance change from PCM to Optane PMM motivates us to reconsider the design of NVMM file systems.

In Table 1, we can see that NVMM provides lower latency than SSD and disk. Besides, it provides a larger capacity than DRAM and is non-volatile. We can build a file system on NVMM to improve performance. However, NVMM is more expensive than SSD and disk, and we should build a cross-media file system on multiple storage media to reduce cost. In this paper, we only focus on file systems on NVMM, which can be used as memory and provide load/store interfaces.

To show the performance of Optane PMM and obtain some conclusions, we run some experiments and show results in this paper. We conduct all experiments on a server equipped with two sockets (NUMA nodes). Each socket contains one Intel Xeon Gold 6271 CPU, 128 GB DRAM and two 256 GB Optane PMMs. Each CPU has 24 cores and a shared 3.3 MB last level cache (LLC). All experiments are running in Centos 7.4.1708 with Linux kernel 4.18.8. We run all evaluations in NUMA node 0 to exclude the impact of NUMA architecture. We set Optane PMM in App Direct Mode[16], which is directly exposed to the CPU and operating system.

2.2 File System

The file system divides the data into file data and metadata. The application data is stored in the file data. The file system is not aware of the content of file data and only treats the file data as byte streams. Applications use write interface to store data and read interface to read data. Metadata records the file system and file information to support file data read and write operations.

File system metadata is mainly divided into two categories. One is the file system metadata, which records the global information of the file system, such as storage space and namespace. For example, we can use the namespace to find the target file. The other is file metadata[⑤], which records the attribute information and data location of each file. When accessing a file, the file system finds the target file by using the namespace, and locates the file data block by using file metadata.

Taking writing a file on ext4 as an example, the application needs to execute at least three system calls: *open*, *write* and *close*. The *open* operation finds the target file, checks the write permission and marks the file accessing. The *write* operation locates the target file data block, and then writes the data. In this process, the file system may need metadata information to allocate new space to write data. Finally, the application uses *close* to stop the file operations, and the file system can recycle the file cache in DRAM. During the entire operation, the file system needs access data and metadata in storage devices.

The existing studies optimize these operations to improve performance. The file system calls (such as *open* and *close*) are handled in the virtual file system (VFS, as shown in Fig.1(a)). Since the block device is not byte-addressable and metadata is usually updated by several or tens of bytes, VFS caches file system metadata in DRAM to accelerate the file lookup. Because the disk provides high access latency, the block-based file system builds page cache in DRAM to reduce data operations on disk. Besides, file systems use the block layer to support block devices and use the I/O scheduler layer to reorder requests to reduce long disk seek time.

2.3 Design Challenges of NVMM File Systems

We introduce the challenges of file systems on NVMM, including performance, data correctness and cost.

2.3.1 *Performance*

File systems should provide low latency. This includes avoiding additional software operations, opti-

[⑤]The file refers to the original file and the special file, such as the directory file.

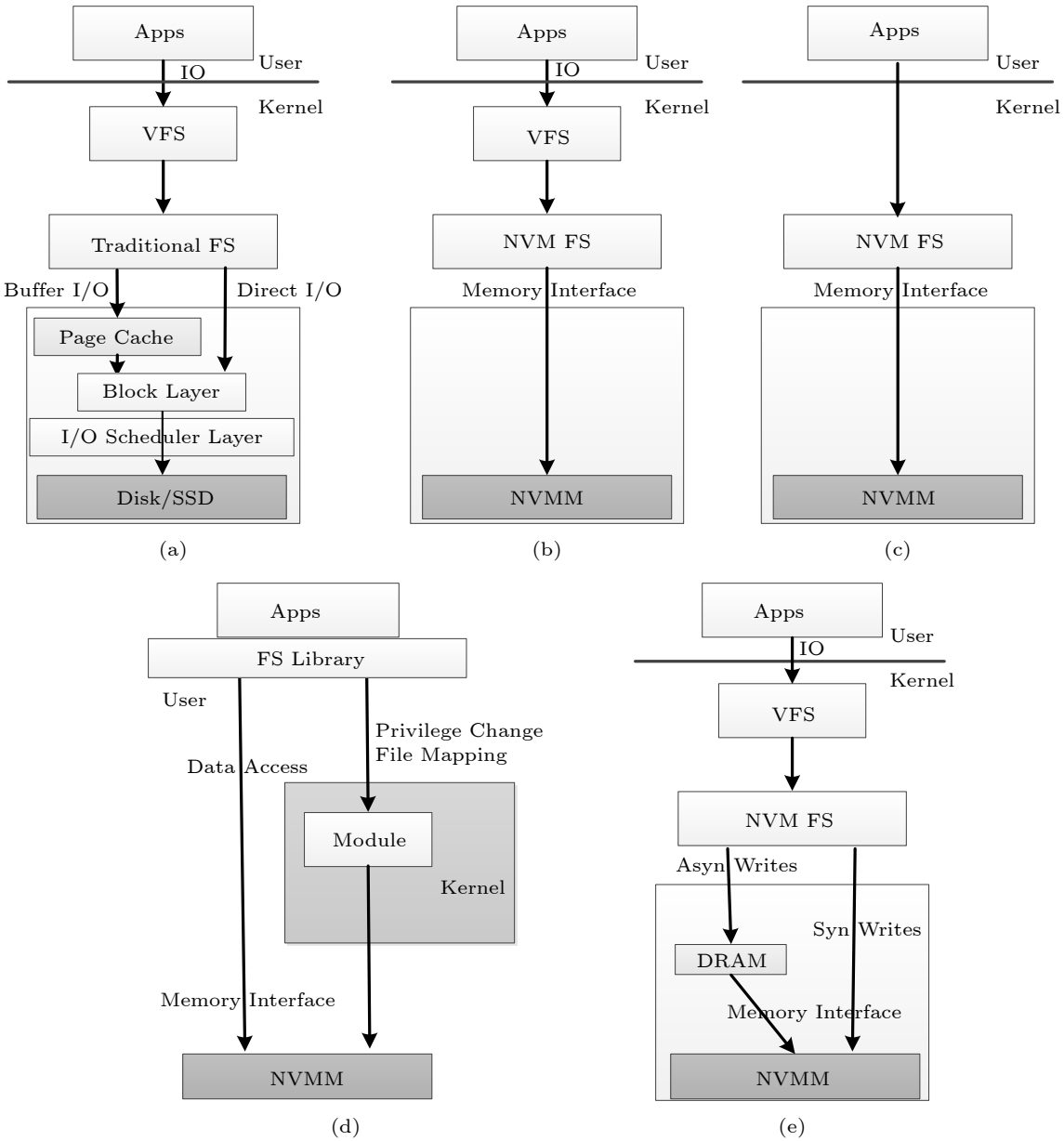

Fig.1. Architecture of several major NVMM file systems. (a) Traditional I/O architecture. (b) NVMM FS architecture. (c) Bypassing VFS architecture. (d) User-level architecture. (e) Using cache architecture.

mizing file system index structures and using DRAM cache. How to design software operations to obtain low latency for NVMM file systems is a challenge.

Besides, file systems should support high concurrency. With the development of multicore, parallelizing I/O operations is a key technique to improve storage performance. Nearly all applications implement concurrent I/O operations, including mobile[42] and desktop[43]. NVMM supports concurrency accessing, and requires file systems to avoid contention across the entire I/O path and support highly concurrent operations.

2.3.2 *Data Correctness*

File systems should guarantee the crash consistency and correctness of file data. Crash inconsistency is caused by performing incomplete write operations. For example, a file write operation needs to update file data and metadata. The system may crash when the data has been updated but the metadata is not updated. When one file system is remounted, only updating the data causes the file system being inconsistent. File systems need to guarantee crash consistency with minimal overhead.

In addition, file systems need to protect the data on NVMM. This is because stray writes and storage media errors can cause changes to the data that has been written to the file system. File systems should provide data protection techniques to avoid, detect and correct these data errors.

2.3.3 Cost to Build

Building file systems should also consider cost. NVMM provides high performance and is byte-addressable, building file systems only on it has a higher performance than block-based file systems. However, NVMM provides high cost and small capacity (Table 1). It is too expensive to use the NVMM file system on a data center. Therefore, cross-media file systems are a better choice, which should consider the performance, capacity and accessing mode of each storage media. For cross-media file systems, we should solve the data placement and the migration problem.

3 Reducing Software Overhead

NVMM is byte-addressable and provides persistent storage, and we can directly access file data in it by using load/store interfaces. Compared with block-based file systems with the deep software stack, NVMM file systems should reduce software overhead. In this section, we first introduce traditional software operations of block-based file systems. Then we introduce several major optimizations to reduce software overhead in NVMM file systems. Finally, we summarize and discuss these technologies based on the performance of the real hardware Optane PMM.

3.1 Software Stack of Block-Based File Systems

Fig.1(a) shows the traditional I/O architecture of kernel-level block-based file systems, which have the deep software stack. VFS caches file metadata. When accessing a file, one first accesses the metadata cache in VFS and then accesses data. If the accessing metadata is not found in VFS, one searches metadata in page cache or block devices and then updates VFS. Page cache caches all file data (including file metadata and data), which can avoid accessing slow block devices frequently[⑥]. If page cache does not contain the accessing data, one should access data in slow block devices. Since block devices only support read or write operations in fixed-size block granularity, a generic block layer is established below the file systems to convert data into blocks. Besides, the I/O scheduler layer reorders and merges I/O operations to reduce random I/O and improve spatial locality. [44] shows the complicated IO stack can occupy almost half of the whole execution time. NVMM is byte-addressable and provides access latency close to DRAM. Therefore, the software operations of page cache, the block layer and the I/O scheduler should be reconsidered for NVMM file systems.

3.2 Shortening I/O Stack

A number of studies shorten software stack to reduce software overhead. NVMM file systems, such as BPFS[37], SCMFS[6], PMFS[5] and NOVA[18], remove page cache, block layer (Fig.1(b)). Ext4[⑦] and xfs[⑧] also use the direct access (DAX) mechanism to directly access file data on NVMM. File data can be accessed directly and quickly by using memory interfaces (load/store).

The studies of [45, 46] argue that the metadata cache in VFS is also unnecessary for NVMM file systems (Fig.1(c)). They shorten the metadata path by bypassing VFS, which lets one directly access metadata. They can improve the performance of metadata write operations, such as creating or deleting a file.

3.3 Building NVMM-Aware Cache

NVMM has a higher latency than DRAM (Table 1). Compared with caching data (page cache) and metadata (VFS) in DRAM, accessing to NVMM directly reduces system performance[38]. For example, PCM has the higher write latency than DRAM and directly writing data to PCM is less performant than writing data in DRAM. However, using DRAM to cache data introduces double-copy overhead for all file operations[38] when cache misses occur. Therefore, one can build an NVMM-aware DRAM cache to improve performance in NVMM file systems (Fig.1(e)).

HiNFS[38] builds an NVMM-aware (PCM) write

[⑥]File systems also support direct I/O, which bypasses page cache and directly accesses data on block devices.
[⑦]Corbet J. Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/, Mar. 2022.
[⑧]Chinner D. xfs: Dax support. https://lwn.net/Articles/635514/, Mar. 2022.

buffer in DRAM to buffer the lazy-persistent file writes. It persists the buffered data to NVMM lazily to hide the long write latency of NVMM.

DirectFS[47] builds a small metadata cache in VFS to optimize metadata performance, which can keep original metadata read performance and optimize metadata write performance. Compared with removing VFS cache directly[45, 46], DirectFS can avoid harming metadata read performance.

3.4 Using User-Level File Systems

Some studies[23, 46, 48–53] suggest using user-level interfaces to improve the performance of NVMM file systems. They execute file read/write operations at user level and only use a kernel module to perform privileged operations, such as hardware privilege changes and memory mapping. These studies not only remove VFS and page cache, but further reduce the overhead of system calls and kernel interaction (Fig.1(d)).

3.5 Optimizing File System Indexing

File systems use index structures to find data locations quickly. The index structures in file systems can be divided into three categories. The first is the file index, which is used to support namespace. We can use it to find the target file. The second is the file data block index, which is used to locate file data block in a file. The last one manages free space in the file systems. These index structures are important for getting good file system performance. NVMM file systems can use efficient index structures to optimize file operations. In addition, NVMM has a higher read latency than DRAM, and one can build index structures in DRAM to accelerate data lookup.

3.5.1 *Using Efficient Index Structure*

File Index. File index is used to look up the target file. In file systems, this lookup process involves two index structures. One uses the file name to find the file dentry in the directory index. The other uses the inode number in the dentry to look up the file inode.

Block-based file systems use B-tree[9] and Htree[7] as the directory index. These tree structures are suitable for block devices but cannot utilize NVMM's high performance fully. For example, B-tree has high spatial locality, which reduces the seek operations of block devices. However, NVMM does not require seeking and supports byte-addressability. The advantages of B-tree in block devices are gone, which introduces a lot of consistency overhead[36, 54, 55] and reduces the performance[6, 56]. Aerie[52] and SoupFS[38] use the hash table as the directory index, which can take full advantage of byte-addressable NVMM.

Since the inode number is an integer and can be allocated consecutive numbers, file systems can store inodes in the inode table according to the inode number, such as ext4. By this way, file systems can calculate the offset of the target inode by using the inode number and then locate the inode directly. The inode table provides high lookup performance, and NVMM file systems can reuse it, such as PMFS[5].

File Data Block Index. File systems locate file data blocks by using the file data block index. We use the indirect index block in ext3 as an example. As shown in Fig.2(a), file systems read metadata from the inode, and get the virtual address of indirect index block 1 (ind1, ① in Fig.2(a)). Then, file systems transform the virtual address to the physical address by using the page table to get indirect index block 1 (② in Fig.2(a)). After that, file systems read the indirect index block 1 to get virtual address of the indirect block 2 (③ in Fig.2(a)). The above process can be repeated until the target block 1 is reached. Each time a block is accessed, and it is necessary to look up the file data block index structure and the page table. NVMM file systems can directly use a high-performance index structure to replace the file data block index. For example, NOVA[18] uses the radix tree as the file data block index structure.

In addition, NVMM has the same addressing mode as DRAM, and file systems can remove the file data block index to improve the performance. SCMFS[6] pre-allocates a continuous large address space (such as 10 GB) for each file to avoid using the file data block index. In this way, SCMFS can directly access data block by using offset. As shown in Fig.2(b), we can directly locate the virtual address of block 1 by adding offset (4 KB) on the start virtual address of file *A*. Then, it uses the page table to translate the virtual address into a physical address. The page table is accessed by hardware memory management unit (MMU), which has much higher performance than software operations.

SIMFS[56] has a similar idea to SCMFS—building a continuous virtual address space for each file. The

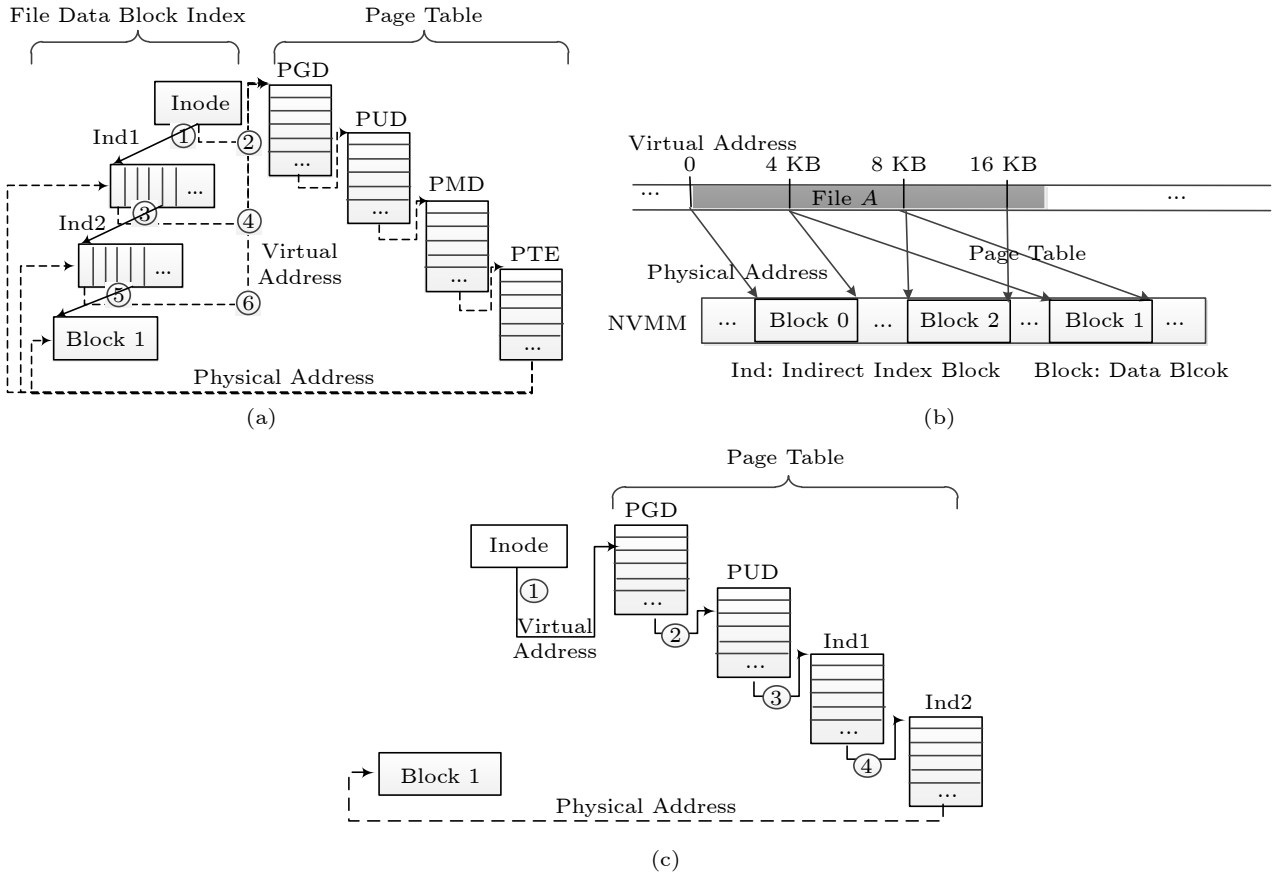

Fig.2. File data block index structure of (a) ext3, (b) SCMFS, and (c) SIMFS.

difference is that it build continuous space by using a file page table. Items in the file page table (PGD in Fig.2(c)) record the physical address of the next page table or target block. When a file is opened, SIMFS directly inserts the file's page table into the process page table. Since the file's page table occupies a contiguous virtual space, SIMFS can directly access the file data block by calculating the address offset.

Free Space Index. The free space index can be subdivided according to the type of free space, such as free data space and free inodes. Traditional block-based and memory (such as ramfs) file systems have provided a lot of index structures, which can provide high performance. NVMM file systems can use these index structures directly. For example, btrfs and NOVA[18, 39] use the red-black tree. This is because the red-black tree can sort the free lists by addresses, allow for efficient merging, and provide $O(\log n)$ deallocation.

3.5.2 *Index Structure Placement*

Since NVMM has a higher read latency than DRAM, building index structures in DRAM can provide higher read performance than NVMM. NOVA[18] builds directory index and file data block index in DRAM to perform index operations quickly. Strata[23] builds file data block index in DRAM. SoupFS[57] and NOVA[18] build free space index in DRAM. These operations benefit from the low read latency of DRAM.

3.6 Summary and Discussion

Table 2 classifies the optimization techniques of reducing software overhead on NVMM file systems. Now, we summarize and discuss them.

3.6.1 *I/O Stack*

The software overhead caused by the deep stack for block devices should be removed for NVMM file systems. This is because NVMM is byte-addressable, and these software operations are unnecessary and reduce performance. As shown in Table 2, most NVMM file systems remove the block layer.

Table 2. Optimization Techniques of Selected NVMM File Systems

Optimization Technique	Technique Detail	NVMM File System
Shortening I/O stack	Removing page cache and block layer	BPFS[37], SCMFS[6], SIMFS[56], FSMAC[58], NVMFS[59], TridentFS[60], Shortcut-JFS[61], DenseFS[62], pNOVA[63], EVFS[53], SPFS[46], PMFS[5], ext4-dax, xfs-dax, Aerie[52], HiNFS[38], HMVFS[64], NOVA[18], NOVA-Fortis[39], Strata[23], SoupFS[57], Ziggurat[28], SplitFS[50], ZoFS[48]
	Bypassing VFS	SPFS[46], ByVFS[45]
Building NVMM-aware cache	–	HiNFS[38], DirectFS[47]
Using user-level file systems	–	[49], EVFS[53], SPFS[46], Aerie[52], Strata[23], DevFS[51], SplitFS[50], ZoFS[48]
Using efficient index structures	Optimizing file index	SoupFS[57], Aerie[52]
	Optimizing file data block index	SCMFS[6], SIMFS[56], NOVA[18]
	Optimizing free space index	NOVA[18], NOVA-Fortis[39]
Building index structures in DRAM	–	Strata[23], NOVA[18], NOVA-Fortis[39], SoupFS[57]

3.6.2 *NVMM-Aware Cache*

Some studies argue that traditional cache is redundant and should be removed[5, 6, 37, 45, 57]. However, DRAM cache can also optimize NVMM file system performance. Fig.3 shows the bandwidth when performing single random read/write operations for the disk file system (ext4 with page cache) and the NVMM file system (ext4-dax and NOVA[18] without page cache) in Optane PMM. We get these results by using fio[⑨]. Compared with ext4, ext4-dax and NOVA improve the write bandwidth by reducing page cache. This is because Ext4 needs to write data in both page cache and Optane PMM. Optane PMM and DRAM have similar write latency (see Table 1). However, Optane PMM shows 3x higher read latency than DRAM. Removing page cache reduces the read bandwidth. As shown in Fig.3, ext4 outperforms ext4-dax and NOVA on read bandwidth by 70.2% and 76.2% respectively.

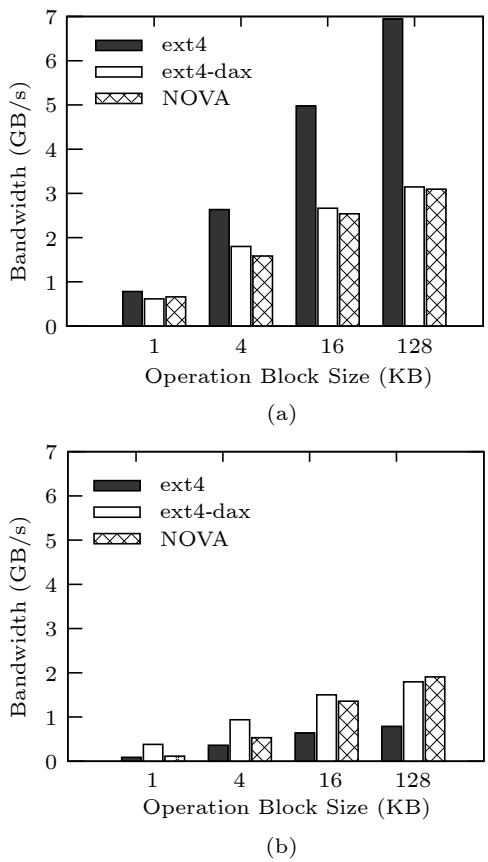

Fig.3. File system bandwidth when preforming random (a) read and (b) write operations.

Therefore, NVMM file systems should build an NVMM-aware DRAM cache to reduce the number of NVMM accesses. If NVMM has long write latency and low write bandwidth, building a write buffer for asynchronous writes is better. For example, PCM has a higher write latency than DRAM. HiNFS[38] builds a write buffer to use non-fsync writes to PCM. Compared with ext4-dax, HiNFS improves the system throughput by up to 184%.

If NVMM has long read latency (Optante PMM), building a read buffer for read-heavy workloads is better. As shown in Fig.3, ext4 with page cache has higher read bandwidth than ext4-dax and NOVA. For mixed read and write workloads, we can build cache according to the data characteristics. For example, DirectFS[47] builds a small NVMM-aware metadata cache to keep metadata read performance and optimize write performance. Fig.4 shows the metadata latency of DirectFS. Cold cache means that VFS does

[⑨]Fio-2.14. https://github.com/axboe/, Mar. 2022.

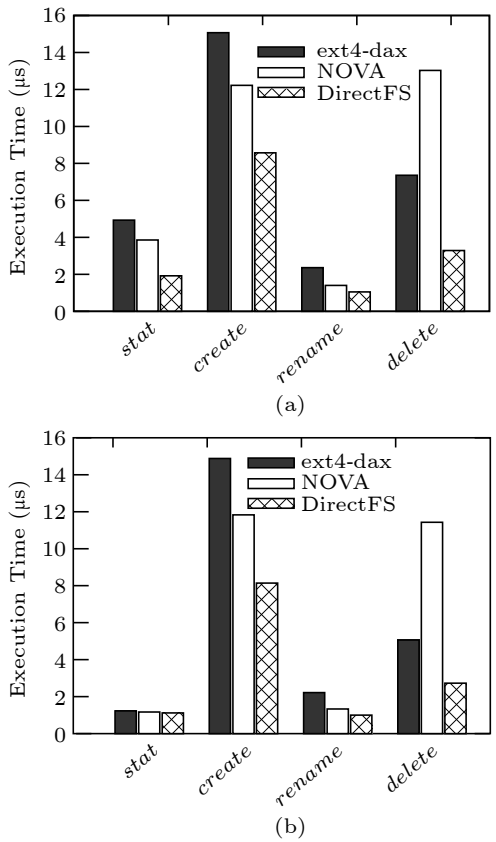

Fig.4. Latency of metadata operations. *stat*, *create*, *rename* and *delete* use system calls and perform reading, creating, renaming and deleting a file respectively. (a) Cold cache. (b) Warm cache.

not contain file metadata before accessing the file. Warm cache represents that VFS contains file metadata before accessing the file. For reading (*stat*) in cold cache and all writing operations (*create*, *rename* and *delete*), DirectFS improves the metadata performance by 48.1% on average compared with existing NVMM file systems ext4-dax and NOVA. For metadata reading in warm cache, DirectFS can keep the similar performance to existing NVMM file systems.

3.6.3 User-Level File Systems

User-level file systems can reduce kernel interaction and system call overhead. However, this results in applications running in their separate address spaces and introduces communication overhead when multiple applications access a shared file concurrently. Therefore, one can use the user-level file systems when files are rarely accessed by multiple applications. Otherwise, we should build kernel-level NVMM file systems. Kernel-level file systems can reduce the communication overhead between processes.

3.6.4 NVMM-Friendly Index

NVMM file systems should build efficient index structures. SCMFS[6] and SIMFS[56] use a memory management module to index file data blocks However, they have some limitations. SCMFS limits the size of each file. SIMFS needs to modify the page table frequently, which is complex and difficult to use. Therefore, we argue that using an efficient index structure is better, such as radix tree. Radix tree[10] is suited for the file data block index because the key of data block is block offset and is distributed densely. Radix tree occupies a small space and provides high performance for densely-distributed keys.

We suggest using a hash table[54, 55, 65] as the directory index. We can build a hash table for each directory or the entire file system shares a global hash table (the key of hash table consists of the file's parent directory inode number and the file name). This is because the hash table provides high performance for single key operations (*open*, *remove* and *stat* in file systems). Besides, it is unnecessary to support range query operations for directory indexes. File systems can use readdir (command *ls*) to scan all data blocks in the directory.

For other indexes in the NVMM file systems, such as the inode table and free space index, we can directly use the index structures of traditional file systems because these index structures can provide high performance. However, existing file systems use different index structures, for instance ext4[7] uses the inode table and btrfs[9] uses B-tree to index inodes. We need to know the characteristics of each index structure and make an appropriate choice. For example, the inode table can quickly locate the target inode by calculating the offset. But its size and storage location are fixed, which limits the number of files in the file system.

Based on the characteristics of Optane PMM, building the index structure in DRAM is better than in Optane PMM. This is because the read latency of Optane PMM is 3x higher than that of DRAM, and there are more lookup operations than update operations for index structures in file systems. More importantly, the operation granularity of Optane PMM is 256 B. Index structures often require modifying a

[10]Corbet. Trees i: Radix trees. https://lwn.net/Articles/175432/, Mar. 2022.

pointer (address, 8 B), which can also result in write amplification. Building index structures in DRAM requires the file system to support recovery after system crashes. Since directory index and free space index can be recovered by scanning all the metadata of the file system, they can be built in DRAM.

4 Improving File System Scalability

With hardware supporting highly concurrent operations, such as multi-cores and NUMA architectures, parallelizing I/O is a key technology to improve performance[43]. As shown in Fig.5, the system has two NUMA nodes, and each node contains DRAM, NVMM, disk and CPU. Each CPU has eight CPU cores. Multiple threads can concurrently access DRAM, NVMM, and disk on different NUMAs using different CPU cores. Besides, nearly all applications need concurrent I/O operations[66], including mobile database[42], desktop database[43], relational database[11], and NoSQL databases RocksDB[12] and MongoDB[13]. These all require file systems to provide high concurrent I/O support.

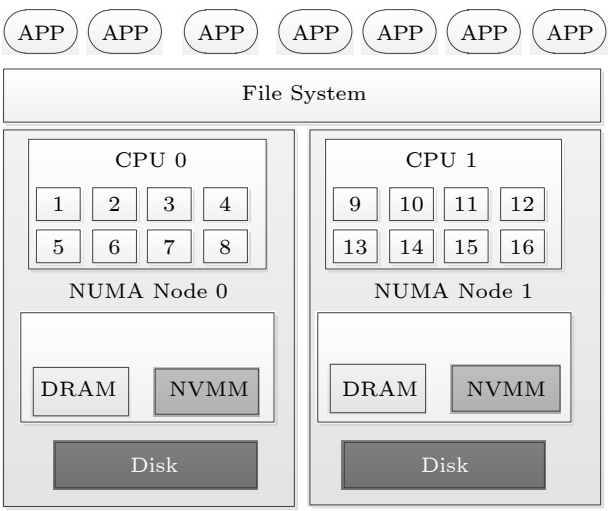

Fig.5. NUMA architecture with two NUMA nodes. APP represents application.

Traditional file systems are designed for slow block devices and use a deep software stack to operate files[19–22], limiting file system concurrency. Some file systems improve scalability by using partition[19, 20] or isolation[21, 22]. However, they are still limited by the software stack[66]. Besides, some studies improve concurrency by increasing additional software operations, such as data merging[23, 24] and garbage collection operations[20], but increase read and write latency. NVMM provides lower access latency than SSD and disk (Table 1), and additional software operations will add NVMM file system latency.

In this section, we analyze the main factors affecting concurrent operations and introduce the existing techniques based on the NVMM file system. We then summarize and discuss these techniques.

4.1 VFS Lock

Existing NVMM file systems[5, 6, 18, 37–39, 48, 50] use VFS to cache metadata. VFS uses read-write locks in each directory, limiting only one writer or multiple readers to run in a single directory. Fig.6 shows the impact of VFS on metadata scalability. To fully analyze the performance, we show the throughput when metadata is cached (*create*, *stat*) and when metadata is not cached (*stat_cold*) respectively. For metadata write operations (*create*), the read-write lock limits the increase of throughput. For metadata read operations when metadata is cached (*stat*), VFS uses RCU-walk[14] to support concurrent reads when metadata is cached, and it scales well with increasing threads. However, when metadata is not cached (*stat_cold*), the path lookup fails to scale due to the read-write lock contention in VFS. Therefore, if we keep VFS on NVMM file systems, we should improve the metada-

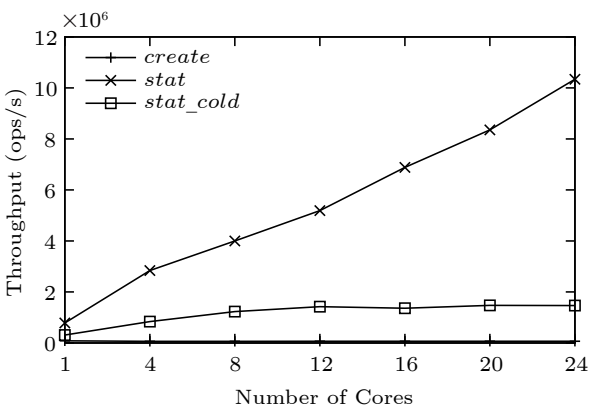

Fig.6. Metadata scalability on ext4-dax. ops/s means the number of operations per second.

[11] https://mariadb.org/, Mar. 2022.
[12] Facebook. Rocksdb. http://rocksdb.org/, Mar. 2022.
[13] https://www.mongodb.org/, Mar. 2022.
[14] Path walking and name lookup locking. https://www.infradead.org/~mchehab/kernel_docs/filesystems/path-walking.html, Mar. 2023.

ta scalability.

DirectFS[47] co-designs the metadata between VFS and a physical file system, using fine-grained flags and atomic write to remove directory locks in VFS, improving metadata scalability.

4.2 Consistency-Induced Contention

File systems require to guarantee consistency (see Section 5). Since the disk does not support concurrent access and has a long seek time, traditional file systems are designed with space locality. For example, ext4 and xfs only allow one thread to write journal, which causes multi-thread competition overhead[20].

Strata[23] and Aerie[52] create logs for each process, which reduces contentions between processes. However, they cannot solve the contention in threads within one process, such as the RocksDB and MySQL database running multiple threads in one process. Also, when multiple processes access the same file, they introduce a lot of inter-process communication overhead to synchronize the logs.

NOVA[18], NOVA-Fortis[39] and DevFS[51] allocate one log (journal) for each file. They prevent races between different files. To support operations on multiple files, such as *rename*, NOVA uses global journaling to support multiple file operations. To further reduce contention, NOVA allocates the global journal to each CPU core, and the global journal of each CPU core can record update information of all files.

4.3 Space Allocation Contention

Providing high concurrency of space allocation is important for file system scalability. Traditional disk-based file systems only use one space allocator, and all threads compete for the allocator. For example, btrfs[9] uses a red-black tree as a space allocator. The red-black tree only allows a single thread to modify concurrently.

NOVA[18] pre-allocates free space to each CPU core. When a thread allocates space, NOVA first allocates space of the CPU core where the thread is running on. This makes the threads run on different CPU cores conflict-free. Only when there is no enough space on the current CPU core, NOVA acquires space from other CPU cores. Compared with NOVA, SoupFS[57] adds objects to each CPU core, including dentries, inodes, B-tree nodes and hash table buckets. ZoFS[48] pre-allocates free space to each thread, avoiding conflicts between threads.

4.4 File Lock

pNOVA[63] and Ziggurat[28] use a fine-grained range lock for a file, avoid using mutex lock[5, 6, 18, 37–39] and allow a file to be written simultaneously by multiple threads.

4.5 Summary and Discussion

The factors limiting concurrent operations in NVMM file systems can come from the file systems themselves as well as other parts of the software stack, such as VFS. In this subsection, we make some suggestions for NVMM file systems. Table 3 shows the optimization aspects of the existing NVMM file systems for concurrent operations, including optimizing VFS lock and reducing consistency-induced contention, space allocation contention and locking range.

Table 3. Concurrency Techniques of Selected NVMM File Systems

Different Contention	NVMM File System
VFS lock	DirectFS[47]
Consistency-induced contention	NOVA[18], NOVA-Fortis[39], Strata[23], DevFS[51], Aerie[52]
Space allocation contention	NOVA[18], NOVA-Fortis[39], ZoFS[48]
File lock	pNOVA[63], Ziggurat[28]

Firstly, removing the VFS lock is necessary. DirectFS uses fine-grained flags and atomic writes to remove the directory lock in VFS. We use mdtest[15], a metadata benchmark, to show the metadata scalability of DirectFS (see Fig.7). Since VFS locks the whole directory, ext4-dax and NOVA fail to scale file create. DirectFS removes the VFS locks and improves metadata scalability.

Secondly, it is important to allocate multiple resources to reduce contentions in NVMM file systems, including consistency and space allocation induced contention. NOVA[18] allocates one log for each file, and each thread can write log without contention. Fig.8 shows the scalablity of NOVA and ext4-dax. We can see that NOVA (NOVA_D) has high scalability than ext4-dax (ext4-dax_D). Since NVMM has low bandwidth, there is little difference in perfor-

[15]https://github.com/MDTEST-LANL/mdtest, Mar. 2022.

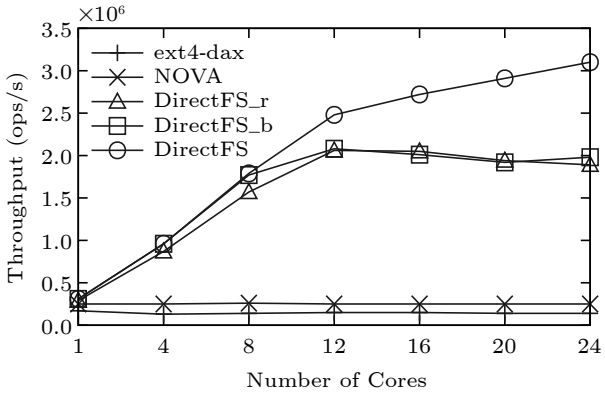

Fig.7. Metadata scalability of creating file.

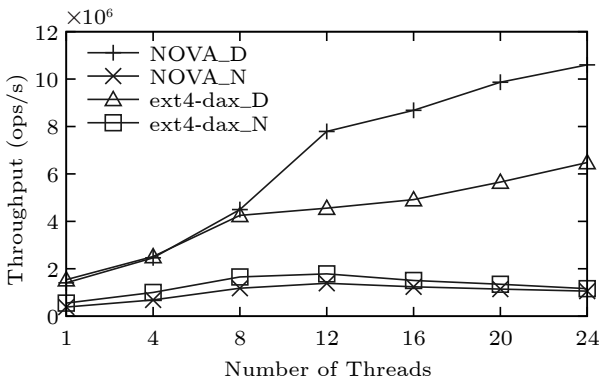

Fig.8. Impact of consistency scalability. N and D represent running on NVMM and DRAM respectively.

mance between NOVA (NOVA_N) and ext4-dax (ext4-dax_N) when being tested on NVMM.

Finally, using fine-grained locking and concurrent index can improve file system concurrency.

Using Fine-Grained Locking. The key to using fine-grained locking is to find contention operations and only lock them. For example, when creating a new file in directory A, the file system needs to allocate inodes and dentries, and then update the metadata of directory A. Traditional file systems lock the entire process. In fact, we only need to lock the process of modifying metadata of directory A. This is because the operations of allocating dentries and inodes do not interfere with one another. They can be executed concurrently.

Using Concurrent Index. The index structures of block-based file systems, such as B-trees and Htrees[67], do not support concurrent operations and have high consistency overhead on NVMM. We recommend considering memory index (hash table) and some new index structures[54, 55, 65] for NVMM file systems. Fig.7 shows the scalability of DirectFS when using hash table (DirectFS), radix tree (DirectFS_r) and B-tree (DirectFS_b) as the directory index. Since radix tree and B-tree only allow one thread updating, DirectFS_r and DirectFS_b cannot improve metadata throughput when the number of threads is up to 12. DirectFS uses the chain hash table as the directory index, which supports high scalability.

For some index structures that only need to be accessed by a single thread, simple data structures are better. For example, SoupFS[57] uses the linked list as the free objects inode. This is because these free objects have the same size and there is no thread contention. Linked list can efficiently support space allocation and deallocation operations for a single thread.

5 Guaranteeing Crash Consistency

File systems should remain consistent after system crashes, which requires that a single operation, such as *open* system call, is committed in all or none fashion. However, a single file system operation usually involves multiple steps and updates multiple places. For example, creating a file (*open* system call) requires building new file metadata and updating parent directory metadata atomically. Since the atomic write granularity of storage devices is sector (SSD and disk) or byte (NVMM), they cannot complete all the updates in multiple positions atomically. File systems require additional mechanisms to guarantee consistency.

NVMM is byte-addressable and supports direct access by the CPU load/store instructions. However, unanticipated cache line eviction may cause data to be written out of order. This causes partial data loss if the system crashes, resulting in an inconsistent file system. We can use cache flush instructions (*clflush, clwb, clflushopt*) and memory fence instructions (*sfence*, *mfence*, *lfence*) to perform sequential data write and ensure consistency on NVMM file systems. However, these instructions suffer from long execution time and serialize memory operations, reducing performance. Traditional consistency techniques are based on block devices, which write data with block granularity and result in write amplification. Write amplification further increases the use of cache flush and memory fence instructions, increasing the overhead of maintaining consistency.

In this section, we introduce and analyze consistency techniques, including copy-on-write, journaling, log-structuring, soft updates and snapshot. Besides, we give some suggestions on NVMM file systems.

5.1 Copy-on-Write

Copy-on-write (COW) copies the data before it is modified[68], which allocates new space and writes new data, and then atomically replaces the old data with new data. After the system crashes, one can only see old data or new data. However, COW can cause iterative updates.

Fig.9(a) shows how COW works when updating block 2 in file *A*. File *A*'s metadata stores in the inode table (In*A* in Fig.9(a)), and the inode uses indirect index blocks (Inds) to index data blocks (same to ext3 in Subsection 3.5.1). Block-based file systems first allocate a new block 2* and write new data to it (① in Fig.9(a)). Then one updates the pointer in indirect index block 2 (Ind2 in Fig.9(a)) from block 2 to block 2*. Although only updating several bytes, block-based file systems need to allocate a new block and rewrite the indirect index block 2 (② in Fig.9(a)). After that, COW updates the pointer in indirect index block 1 (Ind1, ③ in Fig.9(a)). The above process can be repeated until the inode block *A* is updated. This process quadruples the amount of data written in the file system. If a file contains more indirect index blocks in the search path, the write amplification will be larger.

BPFS[37] incorporates atomic in-place updates with COW to reduce write amplification. As shown in Fig.9(b), since file systems only require updating a pointer (8 B) in indirect index block 2 (Ind2 in Fig.9(b)), BPFS can update the pointer in NVMM atomically, avoiding iterative updates and reducing write amplification. However, atomic update does not support the operation of multiple data. For example, if one operation rewrites eight blocks, BPFS should update eight pointers in indirect index block 2 atomically. Since the atomic instructions do not support, BPFS needs to COW the indirect index block 2.

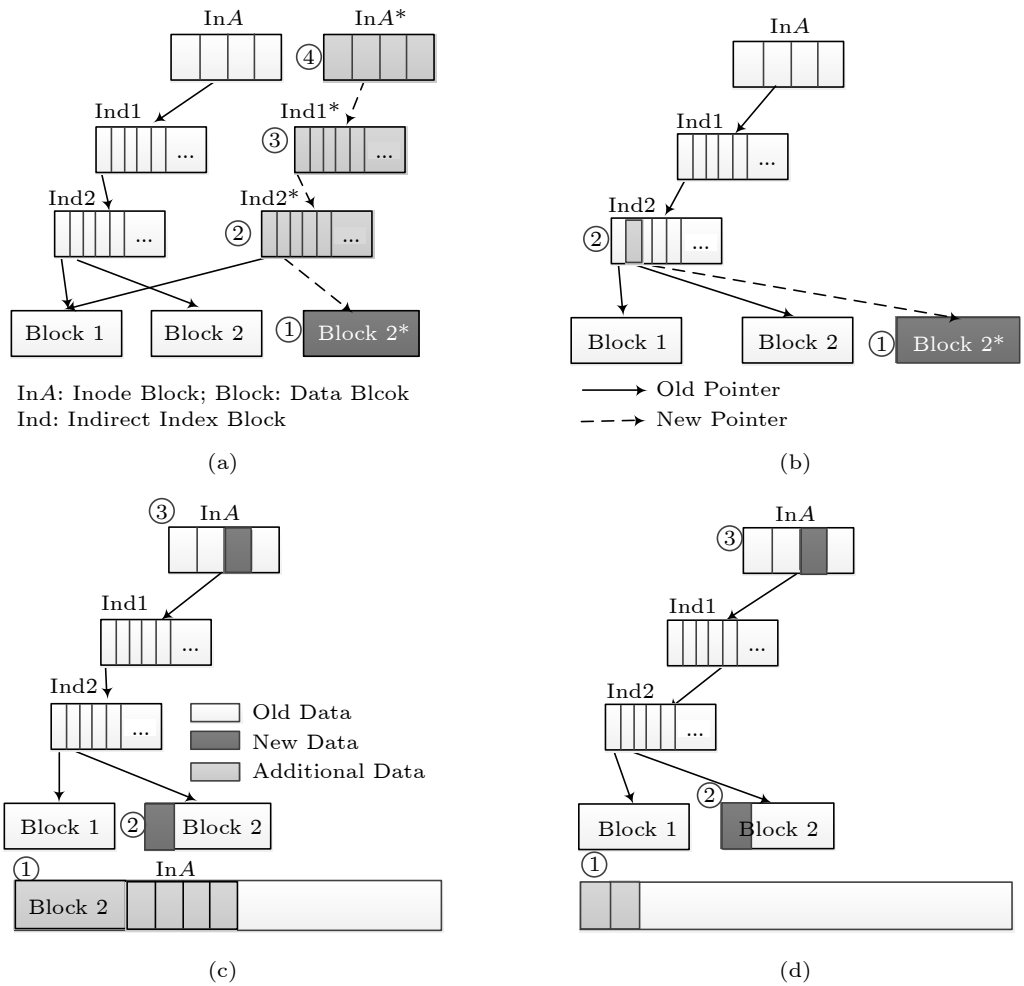

Fig.9. Consistency technology of COW and journaling. Inode block (In*A*) stores file inodes; indirect index block (Ind) stores directory index; block stores file data; new data: new written data in file system; additional data: extra data caused by writing new data. (a) Traditional COW technique. (b) NVMM COW technique. (c) Traditional journaling technique. (d) NVMM journaling technique.

NOVA[18] and PMFS[5] use COW to guarantee crash consistency of file data. For metadata, they use journaling and log-structuring (Subsections 5.2 and 5.3 show more details). This is because metadata updates are usually several bytes and COW needs copying the entire data block, introducing a lot of extra write data.

5.2 Journaling

A journaling file system uses journals to guarantee consistency[5]. For each operation, file systems first store old data or new data in the journal and then perform the operations. Once an improper system shutdown occurs, file systems can be repaired by replaying the journal. As shown in Fig.9(c), when updating data in block 2, block-based file systems first write old block 2 and file metadata (InA) in journal persistently (① in Fig.9(c)), and then update block 2 and file metadata (InA) in-place (② in Fig.9(c)). Journaling writes data twice: one writes to the journal and the other updates in-place[5], which causes write amplification. Besides, block-based file systems record journal at the block size, which further increases write amplification. Some file systems have reduced consistency guarantees, such as ext4 in the ordered or writeback mode, which only record metadata in journal to reduce write amplification.

PMFS[5] uses fine-grained undo journaling to guarantee metadata consistency, which only records the updated metadata. As shown in Fig.9(d), after rewriting block 2* by using COW (①), it only records the updated metadata in journal and then updates the pointer in-place (③). This technique takes advantage of two consistent techniques and avoids unnecessary writes.

5.3 Log-Structuring

A log-structured file system organizes the entire file system as a log. All file operations are sequentially appended to the end of the log in the block size[25, 69]. After the system fails, the file system scans the log to restore its consistency. To reduce the scan time of system recovery, the file system sets checkpoints periodically. All operations before one checkpoint are consistent. Therefore, the file system only scans the written data after the last checkpoint to recover the consistency. Fig.10 shows the process that updates block 2 in file A. File systems firstly append new block 2* in log (①). Then file systems update the pointer in indirect index block 2 (Ind2 in Fig.10) by appending a new indirect index block 2* in log (② in Fig.10). After that, file systems update indirect index block 1 (③ in Fig.10) and inode block A (④ in Fig.10). The write amplification is 4x. Although log-structuring generates similar write amplification to COW, it converts all updates in file systems to sequential access, reducing seek operations in disk and improving throughput. However, log-structuring scatters file data anywhere and results in poor read performance. Besides, log-structuring requires a large amount of contiguous free space, which results in severe garbage collection (GC) overhead[69].

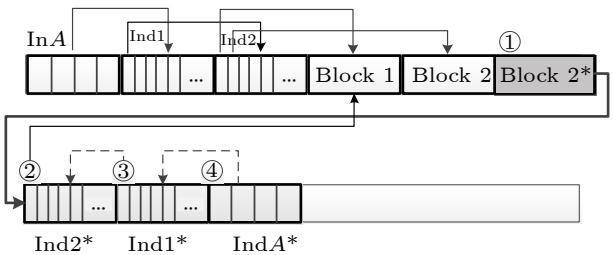

Fig.10. Consistency technique of log-structuring.

The byte-addressability of NVMM allows file systems to only record update data in log, which reduces the write amplification. Besides, it reduces the reliance on locality for read operations. GC becomes a major factor affecting performance. Strata[23] uses the log-structuring technique to record all file writes. Since Strata is a cross-media file system, it needs migrate data to SSD and disk periodically. Strata incorporates GC into the migration process to reduce the negative impact. NOVA[18] uses log-structuring to record metadata. Since metadata is small, NOVA only needs to recycle a small amount of space when performing GC. Besides, NOVA only provides 4 KB continuous space to record log. In this way, NOVA reclaims space in 4 KB block size and can avoid recycling the blocks that have a lot of valid data.

5.4 Soft Updates

Soft updates[16][70, 71] trace update dependencies to provide the metadata consistency. Unlike the techniques described above, soft updates record dependen-

[16]Aurora V. Soft updates, hard problems. https://lwn.net/Articles/339337/, Mar. 2022.

cies in DRAM and then write data to storage media in the background[57, 70, 71]. Therefore, it can achieve similar performance to memory-based file systems, such as tmpfs and ramfs.

However, performing soft updates on block devices can lead to false sharing and complicated dependencies. As shown in Fig.11(a), two threads perform rename operations in directory A respectively. For renaming file B to file C, the file system needs to add dentry of file C in block 2 before deleting the dentry of file B in block 1. For renaming file G to file E, the file system requires adding file E in block 1 before deleting file G in block 2. These two operations can run concurrently without interference, but result in false sharing when operating the same blocks. Delaying data updates on disk requires file systems to keep track of these dependencies to guarantee sequential writes for each operation. It is difficult to understand, implement and maintain soft updates in the mainstream file systems.

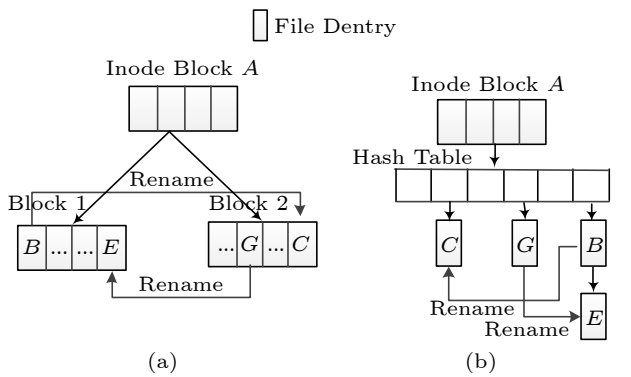

Fig.11. Technology of soft updates. (a) Traditional soft updates. (b) Soft updates of SoupFS.

NVMM is byte-addressable and can remove false sharing and simplify the complexity of soft updates. SoupFS[57] redesigns the directory structure to isolate file operations and reduce dependencies. As shown in Fig.11(b), the dentry of each file can be stored and updated separately. Besides, the hash table supports simultaneous updating by multiple threads. Therefore, the two rename operations can run without interference. By using these techniques, SoupFS solves the complicate dependency of soft updates, improves file system performance and guarantees consistency.

5.5 Snapshots

The snapshot records the state of the file system at a specific time and provides strong consistency guarantee. File systems can recover from snapshots with various error types[64]. To minimize the overhead of snapshot, file systems need to efficiently manage multiple version of snapshot and there is a large amount of duplicate data between versions. The most widely-used approach of snapshot is based on COW-friendly B-tree[64], which uses hierarchical reference count to record the usage of each block[68]. As shown in Fig.12(a), when the file system needs to modify block 1 in file C after the snapshot 2 is created, it needs to allocate a new block to store new data of block 1 and rebuilds the metadata of file C. Besides, the file system requires modifying the reference count of the related data block to support multiple snapshot version. In Fig.12(a), the reference counts of block 2 and other data blocks (data in Fig.12(a)) are updated. Although the hierarchical reference count can postpone explicitly counting references, the number of reference counts that requires to be updated is proportional to the fan-out of such a tree multiplied

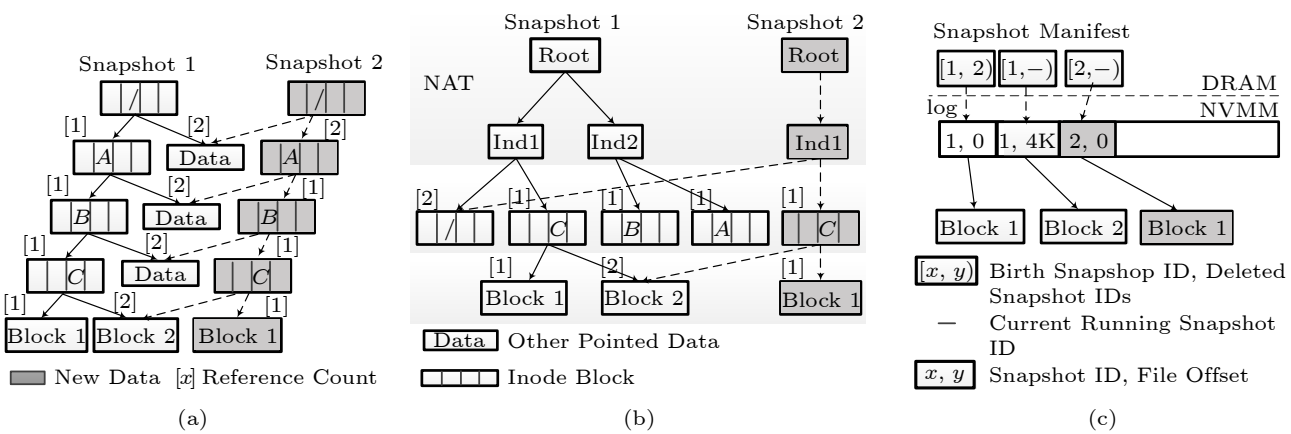

Fig.12. Technologies of snapshots. (a) Snapshots based on COW friendly B-tree. (b) Snapshots of HMVFS. (c) Snapshots of NOVA-Fortis.

by the height[72]. Besides, taking a global snapshot requires updating data from the leaf level to the file system root, including the height of the index tree in a file and the height of the directory hierarchy, which increases file I/O, wastes space and reduces file system performance.

HMVFS[64] solves the hierarchy update problem by building a node address tree (NAT) on NVMM. NAT is used to index multi-version metadata blocks, including the inode blocks, the indirect index block and the direct index blocks. For simplicity, we only show 2-level NAT and omit the indirect and direct index block in Fig.12(b). The data block is directly indexed by an inode. Since one NAT internal node can record 512 entries and one NAT leaf node records 256 entries (each entry can index 4 KB data block), the 4-level NAT can support 64 PB file system data. When creating a snapshot, HMVFS only requires rebuilding six blocks (four NAT blocks, one metadata block, and one data block) for a modified data block. Besides, HMVFS reduces the scope of modification of reference counts. In Fig.12(b), HMVFS only modifies the reference count of the block indexed by indirect block 1 (ind1 in Fig.12(b)), such as the inode block where C is located. Since NVMM is byte-addressable, HMVFS can update the reference count atomically and avoid write amplification.

NOVA-Fortis[39] supports snapshots at file granularity. To avoid write amplification, it uses the log-structuring technique to store metadata and snapshot information. As shown in Fig.12(c), when modifying block 1, NOVA-Fortis allocates a new data block to write block 1 and records current snapshot ID and file offset (2, 0) in the log. It does not need other operations to support snapshots, avoiding write amplification. Besides, NOVA-Fortis builds a snapshot manifest cache in DRAM to accelerate the file access. The cache records a birth snapshot ID and a death snapshot ID for each log entry. As shown in Fig.12(c), log entry 1 belongs to snapshot 1, log entry 2 belongs to snapshot 2 to the latest snapshot and log entry 3 belongs to snapshot 2 to the latest snapshot. When accessing a file, one can quickly locate the snapshot data through cache.

5.6 Summary and Discussion

The above studies optimize traditional crash consistency techniques by reducing write amplification and write dependencies. Table 4 classifies the NVMM file systems according to their consistency techniques. Among these studies, we can see that there is no one technology that is suitable for all operations and file systems tend to use different techniques to guarantee crash consistency for data and metadata. For example, metadata is often modified by small writes. PMFS[5] uses journaling to guarantee metadata consistency. However, using journaling in data updates results in serious write amplification. Therefore, PMFS suggests using COW for data consistency. In this subsection, we summarize and discuss the use cases of these crash consistency technologies.

COW is suitable for file data updates. File data is usually updated at block granularity (e.g, 4 KB) and a write operation introduces at most 8 190 B write amplification (updating two bytes of data, and each byte occupies a data block). File data is written in large sizes frequently. The write amplification of 8 190 B data is acceptable. As shown in Table 4, there are seven NVMM file systems that use COW to ensure data crash consistency.

Log-structuring has similar write amplification to COW but with GC overhead. Unlike disk devices, the random read/write performance of NVMM is the same as sequential read/write with large granularity. As shown in Fig.13, Optane PMM shows similar performance for both sequential and random operations when performing large size operations (such as 4 KB). Therefore, log-structuring is not recommended to

Table 4. Consistency Techniques of Selected NVMM File Systems

Consistency Technique	Data Type	NVMM File System
COW	Metadata	BPFS[37], FSMAC[58], SPFS[46], Aerie[52]
	Data	BPFS[37], SPFS[46], PMFS[5], NOVA[18], Ziggurat[28], HiNFS[38], SplitFS[50]
Journaling	Metadata	HeRMES[73], [49], PMFS[5], ext4-dax, xfs-dax, Aerie[52], NOVA[18], Ziggurat[28], HiNFS[38], SplitFS[50]
	Data	TridentFS[60], Shortcut-JFS[61], [49]
Log-structuring	Metadata	HeRMES[73], pNOVA[63], NOVA[18], Strata[23], Ziggurat[28]
	Data	Strata[23]
Soft updates	Metadata and data	Conquest[74], SoupFS[57]
Snapshot	Metadata and data	HMVFS[64], NOVA-Fortis[39]

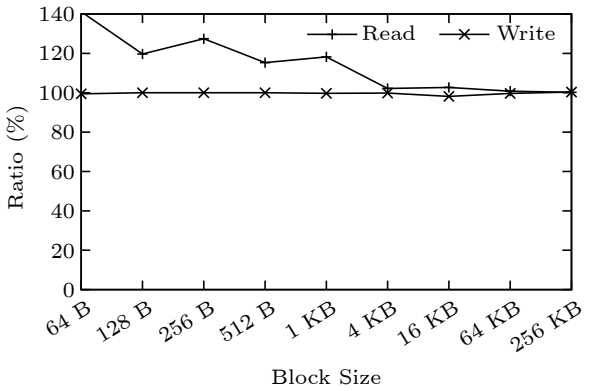

Fig.13. Ratio of random read/write latency to sequential read/write latency respectively.

guarantee file data consistency.

Journaling is better for metadata consistency. For file metadata, it only updates several bytes or up to hundreds of bytes. COW and log-structuring use block updates and may cause 4 KB write amplification. Although we can only record updated metadata to reduce write amplification in COW and log-structuring, different metadata operations result in different update sizes. It is difficult to index this updated metadata and increases lookup overhead. Therefore, journaling is suitable for metadata consistency.

Soft updates can guarantee file system crash consistency with low overhead. However, it may lose data after the system crashes. This is because it records the data into DRAM synchronously and delays updating the data to the storage media. Delayed updates to storage media may cause new data not to persist when the system crashes, resulting in data loss. Therefore, soft updates are not suitable for high reliable storage scenarios, such as data centers.

The snapshot provides a strong consistency guarantee. It records multi-version data and is used to backup data. We can use the technique of HMVFS[64] and NOVA-Fortis[39] to support snapshots. However, writing all snapshots in NVMM increases costs. This is because NVMM is expensive. Since old version snapshots are not accessed frequently, we can migrate old version snapshots into block devices to reduce cost.

6 Protecting Data and NVMM Endurance

NVMM is attached on the memory bus, and threads can access NVMM just like DRAM by using CPU load/store instructions. This can cause other stray writes[5] and produce data errors on NVMM file systems. In addition, NVMM may suffer from media errors, producing incorrect values. NVMM file systems need to avoid, detect and correct these errors. Besides, NVMM has limited endurance and updating some cells too frequently may cause wearing out. In this section, we introduce techniques to solve data protection issues, including software bugs and media errors, as well as techniques to keep NVMM cells wear evenly.

6.1 Software Bugs

Since most NVMM file systems map NVMM into the user or kernel address space, file data is vulnerable to stray writes (software bugs). For example, using a store instruction to access an invalid pointer may modify useful file data and cause permanent corruption. One possible solution is to mark all pages as read-only and upgrade them to be writable when updating data. However, changing the page tables to toggle write permission may take TLB shutdown and hurt performance. Some existing studies[5, 18, 39] leverage the specific CPU register to switch write permission, such as CR0.WP register in x86 architecture, which avoids the TLB shutdown overhead.

ZoFS[48] maps part of NVMM into the user space and data can be modified in the user space. It leverages memory protection keys (MPKs)[75] to represent the permission of one user space region. MPK adds a new 32-bit register PKPU and every two bits represent the permission of one region. The PKPU register is per-thread and it could prevent stray writes from other concurrent threads.

6.2 Media Errors

Like all storage media, NVMM suffers from media errors which may generate incorrect values. NOVA-Fortis[39] assumes memory systems take the responsibility to provide error-correcting codes (ECC) and the memory controller corrects correctable errors on NVMM. In addition, NOVA-Fortis keeps two copies and adds CRC32 checksums to protect and check metadata errors. For file data protection, NOVA-Fortis adopts the mechanism of RAID-4 parity and checksum.

6.3 NVMM Endurance

NVMM has the problem of limited endurance. Da-

ta writing to NVMM contains data and metadata. As for data, it is written in large blocks and may use copy-on-write, which helps the data to be evenly distributed. However, metadata takes a small part space and is updated frequently. Most of NVMM file systems[5, 18] store some metadata (superblocks, inodes, etc.) in a fixed position. Due to the limited endurance of NVMM, this may result in metadata corruption. March[26] and LMWM[27] focus on wear-leveling of inodes. March collects writes into a marched window and slides the window to spread the writes. The exchanging of inodes is implemented by changing the mapping between logical inodes and physical inode slots. LMWM finds that the internal of an inode also has different updating frequencies. Most NVMM file systems store an inode by two 64 B parts and the update frequency of the first part is far more than that of the second part. LMWM uses two 64 B slots to store one inode and control wear-leveling between many 64 B slots, eliminating the unbalance in the internal of one inode.

6.4 Summary

It is necessary to protect data to provide highly reliable file system even at the cost of hurting performance. Using MPK in the user level and the write mechanism in the supervisor mode is better to avoid stray writes. For media errors in NVMM, the mechanism used in DRAM such as checksum and ECC could be learned and applied. Inodes are frequently accessed and can cause wear-out easily. Designing file systems should consider this point.

7 Building Cross-Media File Systems

NVMM provides the lower latency than SSD and disk but has smaller capacity higher cost (see Table 1). Storing data only in NVMM cannot build the file system that is both cost-effective and large-capacity. A cross-media file system is a good choice to leverage the strengths of different storage mediums. We need to consider how data is placed and migrated across multiple storage mediums.

7.1 Data Placement

In cross-media file systems, NVMM can play two roles. One acts as a persistent cache, which stores the latest data, and then this data is migrated to other medium. The other uses NVMM to store part of data.

7.1.1 *NVMM as Persistent Cache*

NVMFS[59] uses NVMM to cache recently accessed data. This technique absorbs small random I/O on NVMM and then performs large sequential writes on SSD. Besides, NVMFS reduces GC overhead on SSD by grouping data with similar update periods in NVMM and writing the data into the same SSD blocks. AFCM[76] and PMW[77] use NVMM to build synchronization cache to reduce the write-back traffic to SSDs.

Strata[23] writes all data into NVMM and then migrates the data to SSD and disk asynchronously. This approach improves small write performance. In addition, migration operations can delete temporary writes, reorganize and compact data for efficient lookup, and batch data into large sequential operations. These operations are beneficial to SSD and disk. However, it is not friendly to sequential write operations. It brings a lot of data migration overhead and block devices can support efficient sequential operations. Directly writing this data to block devices is better. Besides, Strata distributes metadata at each storage level. This may cause that Strata needs to find all of the storage devices while processing the file, reducing file lookup performance.

7.1.2 *NVMM as Storage Medium*

Most studies use NVMM to store the data that is frequently accessed in the file system. They store metadata[58, 74, 78–80], small files[74] and hot data[23, 28, 59, 60] on NVMM. We introduce two representative studies.

FSMAC[58] places metadata on NVMM to improve performance. However, updating metadata on NVMM is faster than updating data on block devices. It may destroy the consistency between metadata and data. For example, when writing new data in file A, one requires updating metadata in NVMM and appending a new data block in SSD. After the system crashes, the file A has a modified file size in NVMM but the old file data in SSD. The file A is inconsistent and cannot recover. Write ordering can solve the problem, which writes file data into SSD before writing metadata. However, the speed mismatch between NVMM and SSD causes the file system to wait for the data to be persisted on SSD, reducing the performance. FSMAC establishes multiple versions of metadata and uses transaction to manage different versions of metadata. Before updating metadata, FSMAC creates a backup of the original version of the

metadata and starts a new transaction. The transaction is committed when the new metadata and data have been written to NVMM and SSD respectively. The original version metadata is deleted only after the transaction has been committed. If the system crashes before the transaction commits, FSMAC restores the original version metadata. Otherwise, FSMAC directly uses new metadata. However, FSMAC cannot guarantee the data of a file operation is persistent after the operation has returned to the application (same to soft updates in Subsection 5.4), and it only guarantees that data and metadata are consistent.

Ziggurat[28] sends small, synchronous writes to NVMM but asynchronous, larger writes to disk. Smaller and synchronous writes use the low-latency and byte-addressable NVMM, and they can be done quickly. Asynchronous and large writes can be cached in DRAM and then written back to disk in the background. This operation can reduce the write pressure of NVMM and utilize the high performance of DRAM.

7.2 Data Migration

Building a cross-media file system should consider data migration between different storage mediums. Although we can run data migration in the background, migrating data slowly can stall foreground operations. For example, the slow migration of data causes that the NVMM space is full. The foreground threads cannot obtain space from NVMM and can only wait. Besides, migration operations occupy device bandwidth, resulting in system performance jitters. We should consider these problems for cross-media file systems.

Strata[23] migrates data from NVMM to SSD and disk when the NVMM space is used beyond a threshold (30% is proposed in Strata). When NVMM runs out of space, the foreground threads must wait for the data migration before allocating new space.

Ziggurat[28] migrates data from NVMM to disk according to the utilization of NVMM. It implements a dynamic threshold based on the overall read-write ratio. If the write ratio is high, the threshold should be lower; otherwise the threshold should be higher. In addition, Ziggurat migrates data when it finds that it is inappropriate to store the data on the current storage medium. For example, Ziggurat migrates a file from NVMM to disk when it finds that the file is cold.

7.3 Summary and Discussion

A cross-media file system should take full advantage of the characteristics of each storage medium to place data and reduce overhead of migrating data. Table 5 shows the NVMM roles used by existing cross-media file systems. We can see that most file systems use NVMM as a storage medium. Now, we summarize and discuss these techniques when NVMM is used as the storage medium.

Table 5. Different NVMM Roles of NVMM File Systems

Role	NVMM File System
Persistent cache	NVMFS[59], Strata[23]
Storage medium	FSMAC[58], Ziggurat[28], Conquest[74], [78], [79], NVMFS[59], TridentFS[60], Strata[23]

7.3.1 *Data Placement*

Small and synchronous writes, such as metadata and log, require immediate persistence and should be placed on NVMM. Hot data also should be placed on NVMM except read-dominated and asynchronous writes. Placing read-dominated data and asynchronous write data on block devices and caching in DRAM are better for NVMM file systems. This is because DRAM shows the lower read latency than NVMM and is volatile. Besides, operating this data on block devices can reduce the access pressure of NVMM, improving performance. Cold data should be migrated to low-cost block devices, such as disks. Block devices offer higher capacity and lower cost than NVMM (see Table 1).

We validate these strategies with the performance of real hardware Optane PMM. Fig.14 shows the normalized latency of Optane PMM and Optane SSD against DRAM (more details of evaluation configuration in Subsection 2.1). We evaluate Optane PMM by mmapping a 300 GB NVMM file (ext4-dax on Optane PMM) into the application address space and then perform read/write operations within the mmap address directly. For Optane SSD, we obtain test results by directly reading and writing data on it. We can see that the read and write latency of Optane PMM is much lower than that of SSD. Therefore, it is beneficial to putting data that requires immediately persistence on NVMM, such as metadata and log. For read latency, Optane PMM provides about 3x latency than DRAM (Fig.14(a)). Therefore, it is more suitable to cache the read-dominated data in DRAM. As the number of threads increases, the la-

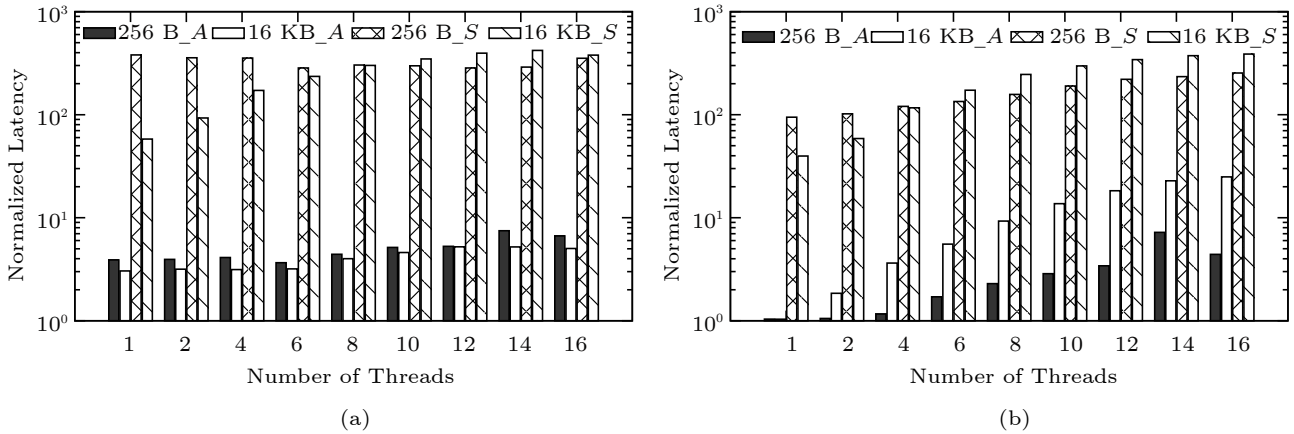

Fig.14. Normalized latency of Optane PMM and Optane SSD against DRAM. 256 B and 16 KB represent the access block size. *A* and *S* represent Optane PMM and SSD respectively. (a) Read latency. (b) Write latency.

tency of Optane PMM, especially for write latency (Fig.14(b)), increases compared with DRAM. Therefore, putting all hot data in Optane PMM increases accessing pressure and results in performance degradation. We should place data and operations according to the load capacity of devices. File systems can use block devices and DRAM to reduce the access pressure of Optane PMM.

7.3.2 *Data Migration*

When migrating data from NVMM to low storage tier on cross-media file systems, one should decide the utilization of NVMM. A fixed utilization threshold cannot be applied to all file system scenarios. For example, a higher threshold is not suitable for write-dominated workloads, because the space can be used by intensive file writes, which causes the foreground write threads to have no space available and stalling. A lower threshold is not suitable for read-dominated workloads because frequent migration operations can cause file system performance jittery. Besides, reads have to load more data blocks from block devices, reducing read performance. Therefore, a dynamic threshold is better for cross-media file systems. We should set a high threshold for read-dominate workloads and a low threshold for write-dominate workloads.

In addition, once the type of data access is determined, such as cold or hot data, we should migrate the data to the appropriate storage medium for high performance.

8 Conclusions

NVMM provides low latency, byte addressing capability, and persistence, changing the storage hierarchy and providing opportunities to improve file system performance. In this paper, we analyzed new challenges for NVMM file systems, including software overhead, scalability, consistency guaranteeing, data correctness protection and cross-media management. After analyzing the techniques of existing studies, we provided a few suggestions based on real hardware Optane PMM from the following five aspects.

Reducing Software Overhead. We suggested that one should adopt various techniques to reduce software overhead for building high-performance NVMM file systems. These techniques mainly include shortening IO stack, building NVMM-aware cache, using user-level file systems or kernel file systems in different situations, and building NVMM-friendly index.

Improving File System Scalability. We suggested improving scalability of VFS for the kernel-level NVMM file system. Moreover, one should use fine-grained locks and highly-concurrent index structures in the NVMM file system.

Guaranteeing Crash Consistency. We suggested that one should carefully choose crash consistency techniques for different situations. For example, journaling is suitable for guaranteeing metadata consistency, and meanwhile copy-on-write works well for guaranteeing crash consistency for file data updates.

Protecting Data and NVMM Endurance. We suggested using the MPK technique and checksum to protect data and endurance in NVMM file systems.

Building Cross-Media File Systems. We suggested that one should take advantage of different storage mediums when building cross-media file systems. Specifically, careful data placement as well as data migration should be conducted for achieving high performance.

References

[1] Akel A, Caulfield A M, Mollov T I, Gupta R K, Swanson S. Onyx: A prototype phase change memory storage array. In *Proc. the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, Jun. 2011. DOI: 10.5555/2002218.2002220

[2] Baek I G, Lee M S, Seo S, Lee M J, Seo D H, Suh D S, Park J C, Park S O, Kim H S, Yoo I K, Chung U I, Moon J T. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Proc. the 2004 IEEE International Electron Devices Meeting*, Dec. 2004, pp.587–590. DOI: 10.1109/IEDM.2004.1419228.

[3] Kawahara T. Scalable spin-transfer torque RAM technology for normally-off computing. *IEEE Design & Test of Computers*, 2011, 28(1): 52–63. DOI: 10.1109/MDT.2010.97.

[4] Raoux S, Burr G W, Breitwisch M J, Rettner C T, Chen Y C, Shelby R M, Salinga M, Krebs D, Chen S H, Lung H L, Lam C H. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 2008, 52(4/5): 465–479. DOI: 10.1147/rd.524.0465.

[5] Dulloor S R, Kumar S, Keshavamurthy A, Lantz P, Reddy D, Sankaran R, Jackson J. System software for persistent memory. In *Proc. the 9th European Conference on Computer Systems*, Apr. 2014, Article No. 15. DOI: 10.1145/2592798.2592814.

[6] Wu X J, Reddy A L N. SCMFS: A file system for storage class memory. In *Proc. the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2011, Article No. 39. DOI: 10.1145/2501620.2501621

[7] Mathur A, Cao M M, Bhattacharya S, Dilger A, Tomas A, Vivier L. The new ext4 filesystem: Current status and future plans. In *Proc. the 2007 Linux Symposium*, Jun. 2007, pp.21–34.

[8] Sweeney A, Doucette D, Hu W, Anderson C, Nishimoto M, Peck G. Scalability in the XFS file system. In *Proc. the USENIX 1996 Annual Technical Conference*, Jan. 1996. DOI: 10.5555/1268299.1268300.

[9] Rodeh O, Bacik J, Mason C. BTRFS: The Linux B-tree filesystem. *ACM Trans. Storage*, 2013, 9(3): Article No. 9. DOI: 10.1145/2501620.2501623.

[10] Lee C, Sim D, Hwang J Y, Cho S. F2FS: A new file system for flash storage. In *Proc. the 13th USENIX Conference on File and Storage Technologies*, Feb. 2015, pp.273–286. DOI: 10.5555/2750482.2750503.

[11] Campello D, Lopez H, Useche L, Koller R, Rangaswami R. Non-blocking writes to files. In *Proc. the 13th USENIX Conference on File and Storage Technologies*, Feb. 2015, pp.151–165. DOI: 10.5555/2750482.2750494.

[12] Chidambaram V, Sharma T, Arpaci-Dusseau A C, Arpaci-Dusseau R H. Consistency without ordering. In *Proc. the 10th USENIX Conference on File and Storage Technologies*, Feb. 2012. DOI: 10.5555/2208461.2208470.

[13] Jannen W, Yuan J, Zhan Y *et al.* BetrFS: A right-optimized write-optimized file system. In *Proc. the 13th USENIX Conference on File and Storage Technologies*, Feb. 2015, pp.301–315. DOI: 10.5555/2750482.2750505.

[14] Yuan J, Zhan Y, Jannen W, Pandey P, Akshintala A, Chandnani K, Deo P, Kasheff Z, Walsh L, Bender M A, Farach-Colton M, Johnson R, Kuszmaul B C, Porter D E. Optimizing every operation in a write-optimized file system. In *Proc. the 14th USENIX Conference on File and Storage Technologies*, Feb. 2016. DOI: 10.5555/2930583.2930584.

[15] Zhan Y, Conway A, Jiao Y Z, Knorr E, Bender M A, Farach-Colton M, Jannen W, Johnson R, Porter D E, Yuan J. The full path to full-path indexing. In *Proc. the 16th USENIX Conference on File and Storage Technologies*, Feb. 2018, pp.123–138. DOI: 10.5555/3189759.3189771.

[16] Izraelevitz J, Yang J, Zhang L, Kim J, Liu X, Memaripour A, Soh Y J, Wang Z X, Xu Y, Dulloor S R, Zhao J S, Swanson S. Basic performance measurements of the Intel Optane DC persistent memory module. arXiv: 1903.05714, 2019. https://arxiv.org/abs/1903.05714, Mar. 2023.

[17] Qureshi M K, Srinivasan V, Rivers J A. Scalable high performance main memory system using phase-change memory technology. In *Proc. the 36th Annual International Symposium on Computer Architecture*, Jun. 2009, pp.24–33. DOI: 10.1145/1555754.1555760.

[18] Xu J, Swanson S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. the 14th USENIX Conference on File and Storage Technologies*, Feb. 2016, pp.323–338. DOI: 10.5555/2930583.2930608.

[19] Kang J B, Zhang B L, Wo T, Hu C M, Huai J P. MultiLanes: Providing virtualized storage for OS-level virtualization on many cores. In *Proc. the 12th USENIX Conference on File and Storage Technologies*, Feb. 2014, pp.317–329. DOI: 10.1145/2801155.

[20] Kang J B, Zhang B L, Wo T, Yu W R, Du L, Ma S, Huai J P. SpanFS: A scalable file system on fast storage devices. In *Proc. the USENIX 2015 Annual Technical Conference*, Jul. 2015, pp.249–261. DOI: 10.5555/2813767.2813786.

[21] Lu L Y, Zhang Y P, Do T, Al-Kiswany S, Arpaci-Dusseau A C, Arpaci-Dusseau R H. Physical disentanglement in a container-based file system. In *Proc. the 11th USENIX Conference on Operating Systems Design and Implementation*, Oct. 2014, pp.81–96. DOI: 10.5555/2685048.2685056.

[22] Psaroudakis I, Scheuer T, May N *et al.* Scaling up concurrent main-memory column-store scans: Towards adaptive NUMA-aware data and task placement. *Proceedings of the VLDB Endowment*, 2015, 8(12): 1442–1453. DOI: 10.14778/2824032.2824043.

[23] Kwon Y, Fingler H, Hunt T, Peter S, Witchel E. Strata: A cross media file system. In *Proc. the 26th Symposium on Operating Systems Principles*, Oct. 2017, pp.460–477. DOI: 10.1145/3132747.3132770.

[24] Bhat S S, Eqbal R, Clements A T, Kaashoek M F. Scaling a file system to many cores using an operation log. In *Proc. the 26th Symposium on Operating Systems Principles*, Oct. 2017, pp.69–86. DOI: 10.1145/3132747.3132779.

[25] Rosenblum M, Ousterhout J K. The design and imple-

mentation of a log-structured file system. *ACM Trans. Computer Systems*, 1992, 10(1): 26–52. DOI: 10.1145/146941.146943.

[26] Chang H S, Chang Y H, Hsiu P C, Kuo T W, Li H P. Marching-based wear-leveling for PCM-based storage systems. *ACM Trans. Design Automation of Electronic Systems*, 2015, 20(2): Article No. 25. DOI: 10.1145/2699831.

[27] Yang C S, Liu D, Zhang R Y, Chen X Z, Nie S, Wang F S, Zhuge Q F, Sha E H M. Efficient multi-grained wear leveling for inodes of persistent memory file systems. In *Proc. the 57th ACM/IEEE Design Automation Conference*, Jul. 2020. DOI: 10.1109/DAC18072.2020.9218626.

[28] Zheng S A, Hoseinzadeh M, Swanson S. Ziggurat: A tiered file system for non-volatile main memories and disks. In *Proc. the 17th USENIX Conference on File and Storage Technologies*, Feb. 2019, pp.207–219. DOI: 10.5555/3323298.3323318.

[29] Wu C W, Zhang G Y, Li K Q. Rethinking computer architectures and software systems for phase-change memory. *ACM Journal on Emerging Technologies in Computing Systems*, 2016, 12(4): Article No. 33. DOI: 10.1145/2893186.

[30] Chen A. A review of emerging non-volatile memory (NVM) technologies and applications. *Solid-State Electronics*, 2016, 125: 25–38. DOI: 10.1016/j.sse.2016.07.006.

[31] Mittal S, Vetter J S. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. Parallel and Distributed Systems*, 2016, 27(5): 1537–1550. DOI: 10.1109/TPDS.2015.2442980.

[32] Puglia G O, Zorzo A F, De Rose C A F, Perez T, Milojicic D. Non-volatile memory file systems: A survey. *IEEE Access*, 2019, 7: 25836–25871. DOI: 10.1109/ACCESS.2019.2899463.

[33] Lee B C, Ipek E, Mutlu O, Burger D. Architecting phase change memory as a scalable dram alternative. In *Proc. the 36th Annual International Symposium on Computer Architecture*, Jun. 2009, pp.2–13. DOI: 10.1145/1555754.1555758.

[34] Chang M F, Wu J J, Chien T F, Liu Y C, Yang T C, Shen W C, King Y C, Lin C J, Lin K F, Chih Y D, Natarajan S, Chang J. 19.4 embedded 1mb ReRAM in 28nm CMOS with 0.27-to-1v read using swing-sample-and-couple sense amplifier and self-boost-write-termination scheme. In *Proc. the 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Feb. 2014, pp.332–333. DOI: 10.1109/ISSCC.2014.6757457.

[35] Chen R H, Shao Z L, Liu D, Feng Z Y, Li T. Towards efficient NVDIMM-based heterogeneous storage hierarchy management for big data workloads. In *Proc. the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2019, pp.849–860. DOI: 10.1145/3352460.3358266.

[36] Yang J, Wei Q S, Chen C, Wang C D, Yong K L, He B S. NV-tree: Reducing consistency cost for NVM-based single level systems. In *Proc. the 13th USENIX Conference on File and Storage Technologies*, Feb. 2015, pp.167–181. DOI: 10.5555/2750482.2750495.

[37] Condit J, Nightingale E B, Frost C, Ipek E, Lee B, Burger D, Coetzee D. Better I/O through byte-addressable, persistent memory. In *Proc. the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, Oct. 2009, pp.133–146. DOI: 10.1145/1629575.1629589.

[38] Ou J X, Shu J W, Lu Y Y. A high performance file system for non-volatile main memory. In *Proc. the 11th European Conference on Computer Systems*, Apr. 2016, Article No. 12. DOI: 10.1145/2901318.2901324.

[39] Xu J, Zhang L, Memaripour A, Gangadharaiah A, Borase A, Da Silva T B, Swanson S, Rudoff A. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proc. the 26th Symposium on Operating Systems Principles*, Oct. 2017, pp.478–496. DOI: 10.1145/3132747.3132761.

[40] Volos H, Magalhaes G, Cherkasova L, Li J. Quartz: A lightweight performance emulator for persistent memory software. In *Proc. the 16th Annual Middleware Conference*, Nov. 2015, pp.37–49. DOI: 10.1145/2814576.2814806.

[41] Yang J, Kim J, Hoseinzadeh M, Izraelevitz J, Swanson S. An empirical guide to the behavior and use of scalable persistent memory. In *Proc. the 18th USENIX Conference on File and Storage Technologies*, Feb. 2020, pp.169–182. DOI: 10.5555/3386691.3386708.

[42] Jeong D, Lee Y, Kim J S. Boosting quasi-asynchronous I/O for better responsiveness in mobile devices. In *Proc. the 13th USENIX Conference on File and Storage Technologies*, Feb. 2015, pp.191–202. DOI: 10.5555/2750482.2750497.

[43] Harter T, Dragga C, Vaughn M, Arpaci-Dusseau A C, Arpaci-Dusseau R H. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proc. the 23rd ACM Symposium on Operating Systems Principles*, Oct. 2011, pp.71–83. DOI: 10.1145/2043556.2043564.

[44] Lee G, Shin S, Song W, Ham T J, Lee J W, Jeong J. Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs. In *Proc. the USENIX 2019 Annual Technical Conference*, Jul. 2019, pp.603–616. DOI: 10.5555/3358807.3358858.

[45] Wang Y, Jiang D, Xiong J. Caching or not: Rethinking virtual file system for non-volatile main memory. In *Proc. the 10th USENIX Workshop on Hot Topics in Storage and File Systems*, Jul. 2018.

[46] Zhou D, Pan W, Xie T *et al.* A file system bypassing volatile main memory: Towards a single-level persistent store. In *Proc. the 15th ACM International Conference on Computing Frontiers*, May 2018, pp.97–104. DOI: 10.1145/3203217.3203277.

[47] Wang Y, Jiang D J, Xiong J. Revisiting virtual file system for metadata optimized non-volatile main memory file system. In *Proc. the 36th International Conference on Massive Storage Systems and Technology*, Oct. 2020.

[48] Dong M K, Bu H, Yi J F *et al.* Performance and protection in the ZoFS user-space NVM file system. In *Proc. the 27th ACM Symposium on Operating Systems Principles*, Oct. 2019, pp.478–493. DOI: 10.1145/3341301.3359637.

[49] Sha E H M, Jia Y, Chen X Z, Zhuge Q F, Jiang W W, Qin J J. The design and implementation of an efficient

user-space in-memory file system. In *Proc. the 5th Non-Volatile Memory Systems and Applications Symposium*, Aug. 2016. DOI: 10.1109/NVMSA.2016.7547176.

[50] Kadekodi R, Lee S K, Kashyap S, Kim T, Kolli A, Chidambaram V. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proc. the 27th ACM Symposium on Operating Systems Principles*, Oct. 2019, pp.494–508. DOI: 10.1145/3341301.3359631.

[51] Kannan S, Arpaci-Dusseau A C, Arpaci-Dusseau R H, Wang Y G, Xu J, Palani G. Designing a true direct-access file system with DevFS. In *Proc. the 16th USENIX Conference on File and Storage Technologies*, Feb. 2018, pp.241–255. DOI: 10.5555/3189759.3189782.

[52] Volos H, Nalli S, Panneerselvam S *et al.* Aerie: Flexible file-system interfaces to storage-class memory. In *Proc. the 9th European Conference on Computer Systems*, Apr. 2014, Article No. 14. DOI: 10.1145/2592798.2592810.

[53] Yoshimura T, Chiba T, Horii H. EvFS: User-level, event-driven file system for non-volatile memory. In *Proc. the 11th USENIX Conference on Hot Topics in Storage and File Systems*, Jul. 2019. DOI: 10.5555/3357062.3357083.

[54] Chen S M, Jin Q. Persistent B^+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 2015, 8(7): 786–797. DOI: 10.14778/2752939.2752947.

[55] Oukid I, Lasperas J, Nica A, Willhalm T. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proc. the 2016 International Conference on Management of Data*, Jun. 2016, pp.371–386. DOI: 10.1145/2882903.2915251.

[56] Sha E H M, Chen X Z, Zhuge Q F, Shi L, Jiang W W. A new design of in-memory file system based on file virtual address framework. *IEEE Trans. Computers*, 2016, 65(10): 2959–2972. DOI: 10.1109/TC.2016.2516019.

[57] Dong M K, Chen H B. Soft updates made simple and fast on non-volatile memory. In *Proc. the USENIX 2017 Annual Technical Conference*, Jul. 2017, pp.719–731. DOI: 10.5555/3154690.3154758.

[58] Chen J X, Wei Q S, Chen C, Wu L K. FSMAC: A file system metadata accelerator with non-volatile memory. In *Proc. the 29th Symposium on Mass Storage Systems and Technologies*, May 2013. DOI: 10.1109/MSST.2013.6558440.

[59] Qiu S, Reddy A L N. NVMFS: A hybrid file system for improving random write in nand-flash SSD. In *Proc. the 29th Symposium on Mass Storage Systems and Technologies*, May 2013. DOI: 10.1109/MSST.2013.6558434.

[60] Huang T C, Chang D W. TridentFS: A hybrid file system for non-volatile RAM, flash memory and magnetic disk. *Software Practice and Experience*, 2016, 46(3): 291–318. DOI: 10.1002/spe.2299.

[61] Lee E, Yoo S, Jang J E, Bahn H. Shortcut-JFS: A write efficient journaling file system for phase change memory. In *Proc. the 28th Symposium on Mass Storage Systems and Technologies*, Apr. 2012. DOI: 10.1109/MSST.2012.6232378.

[62] Weiss Z, Arpaci-Dusseau A C, Arpaci-Dusseau R H. DenseFS: A cache-compact filesystem. In *Proc. the 10th USENIX Conference on Hot Topics in Storage and File Systems*, Jul. 2018. DOI: 10.5555/3277332.3277334.

[63] Kim J H, Kim J, Kang H, Lee C G, Park S, Kim Y. pNOVA: Optimizing shared file I/O operations of NVM file system on manycore servers. In *Proc. the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, Aug. 2019. DOI: 10.1145/3343737.3343748.

[64] Zheng S A, Huang L P, Liu H, Wu L Z, Zha J. HMVFS: A hybrid memory versioning file system. In *Proc. the 32nd Symposium on Mass Storage Systems and Technologies*, May 2016. DOI: 10.1109/MSST.2016.7897079.

[65] Lee S K, Lim K H, Song H, Nam B, Noh S H. WORT: Write optimal radix tree for persistent memory storage systems. In *Proc. the 15th USENIX Conference on File and Storage Technologies*, Feb. 2017, pp.257–270. DOI: 10.5555/3129633.3129657.

[66] Min C, Kashyap S, Maass S, Kang W, Kim T. Understanding manycore scalability of file systems. In *Proc. the USENIX 2016 Annual Technical Conference*, Jun. 2016, pp.71–85. DOI: 10.5555/3026959.3026967.

[67] Phillips D. A directory index for ext2. In *Proc. the 5th Annual Linux Showcase & Conference*, Nov. 2001. DOI: 10.5555/1268488.1268508.

[68] Rodeh O. B-trees, shadowing, and clones. *ACM Trans. Storage*, 2008, 3(4): Article No. 2. DOI: 10.1145/1326542.1326544.

[69] Seltzer M, Bostic K, Mckusick M K, Staelin C. An implementation of a log-structured file system for UNIX. In *Proc. the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, Jan. 1993. DOI: 10.5555/1267303.1267306.

[70] Ganger G R, McKusick M K, Soules C A N, Patt Y N. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. Computer Systems*, 2000, 18(2): 127–153. DOI: 10.1145/350853.350863.

[71] McKusick M K, Ganger G R. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proc. the USENIX 1999 Annual Technical Conference*, Jun. 1999. DOI: 10.5555/1268708.1268732.

[72] Dragga C, Santry D J. GCTrees: Garbage collecting snapshots. *ACM Trans. Storage*, 2016, 12(1): Article No. 4. DOI: 10.1145/2857056.

[73] Miller E L, Brandt S A, Long D D E. Hermes: High-performance reliable MRAM-enabled storage. In *Proc. the 8th Workshop on Hot Topics in Operating Systems*, May 2001, pp.95–99. DOI: 10.1109/HOTOS.2001.990067.

[74] Wang A I A, Kuenning G, Reiher P, Popek G. The *conquest* file system: Better performance through a disk/persistent-RAM hybrid design. *ACM Trans. Storage*, 2006, 2(3): 309–348. DOI: 10.1145/1168910.1168914.

[75] Park S, Lee S, Xu W, Moon H, Kim T. Libmpk: Software abstraction for Intel memory protection keys (Intel MPK). In *Proc. the USENIX 2019 Annual Technical Conference*, Jul. 2019, pp.241–254. DOI: 10.5555/3358807.3358829.

[76] Chen Y M, Lu Y Y, Chen P, Shu J W. Efficient and consistent NVMM cache for SSD-based file system. *IEEE Trans. Computers*, 2019, 68(8): 1147–1158. DOI: 10.1109/TC.2018.2870137.

[77] Yang C S, Zhuge Q F, Chen X Z, Sha E H M, Liu D, Zhang R Y. Optimizing synchronization mechanism for block-based file systems using persistent memory. *Future Generation Computer Systems*, 2020, 111: 288–299. DOI: 10.1016/j.future.2020.04.024.

[78] Chen C, Yang J, Wei Q S, Wang C D, Xue M D. Optimizing file systems with fine-grained metadata journaling on byte-addressable NVM. *ACM Trans. Storage*, 2017, 13(2): Article No. 13. DOI: 10.1145/3060147.

[79] Chen C, Yang J, Wei Q S, Wang C D, Xue M D. Fine-grained metadata journaling on NVM. In *Proc. the 32nd Symposium on Mass Storage Systems and Technologies*, May 2016. DOI: 10.1109/MSST.2016.7897077.

[80] Matsui C, Sun C, Takeuchi K. Design of hybrid SSDs with storage class memory and NAND flash memory. *Proceedings of the IEEE*, 2017, 105(9): 1812–1821. DOI: 10.1109/JPROC.2017.2716958.

Ying Wang received her B.E. degree in software engineering from University of Zhengzhou, Zhengzhou, in 2015. She is currently an assistant professor in Institute of Computing Technology, Chinese Academy of Sciences, Beijing. Her research interests include file and storage systems.

Wen-Qing Jia received his B.E. degree in computing science and technology from University of Chinese Academy of Sciences, Beijing, in 2019. Now he is a Ph.D. student of Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests are in file and storage systems.

De-Jun Jiang received his B.S. degree in electronic engineering from Beihang University, Beijing, in 2004, M.S. degree in software engineering from Tsinghua University, Beijing, in 2009, and his Ph.D. degree in computer science from Vrije Universiteit, Amsterdam, in 2012. He is currently an associate professor of Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His current research interests include storage system and architecture, operating system, and distributed system. He is a member of CCF, ACM, and IEEE.

Jin Xiong received her B.S. degree from Sichuan University, Chengdu, in 1990, her M.S. degree and Ph.D. degree in computer science from University of Chinese Academy of Sciences, Beijing, in 1993 and 2006, respectively. She is currently a professor at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. Her research interests include storage systems and file systems. She is a senior member of CCF and a member of ACM and IEEE.