Caching or Not: Rethinking Virtual File System for Non-Volatile Main Memory

Ying Wang Dejun Jiang Jin Xiong

SKL Computer Architecture, ICT, CAS University of Chinese Academy of Sciences
{wangying01, jiangdejun, xiongjin}@ict.ac.cn

Abstract

Virtual File System (VFS) conventionally provides an abstraction for multiple instances of underlying physical file systems as well as metadata caching, concurrency control and permission check, which benefits disk based file systems. However, in this paper we find VFS brings extra overhead when interacting with persistent memory (PM) file systems. We explore the opportunity of shortening VFS stack for PM file systems. We present ByVFS, an optimization of VFS to directly access metadata in PM file systems bypassing VFS caching layer. The results show ByVFS outperforms conventional VFS with cold cache and provides comparable performance against conventional VFS with warm cache. We also present potential issues when reducing VFS overhead.

1 Introduction

Emerging Non-Volatile Memory (NVM) technologies, such as PCM [2, 19], ReRAM[3], STT-RAM [11], and recent 3D XPoint [9] , allow to store persistent data in main memory. This motivates a number of efforts to build file systems on persistent memory (PM), which focus on minimizing software overhead [24, 7, 22], providing strong consistency with low overhead [4, 6, 25], supporting snapshot and fault tolerance [26] , and optimizing slow writes [17].

Compared to block-based devices, byte-addressable non-volatile memory is naturally suitable for small reads/writes. Accessing metadata is an important source of small reads/writes. For example, a major part of file references are dominated by metadata operations [29]. Metadata operations include accessing properties of individual files as well as whole file system (e.g. *statfs*, *stat*, *rename*).

Virtual File System (VFS) is a software abstraction layer that was first introduced to address the problem of accessing local file system and remote Network File

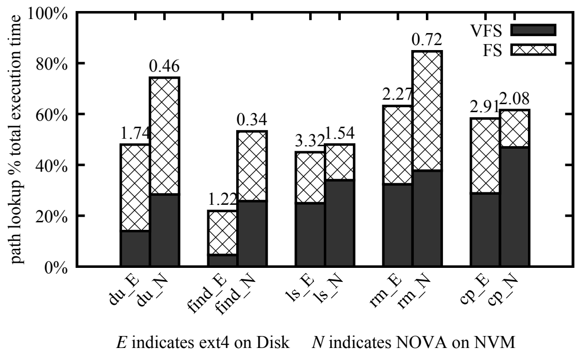
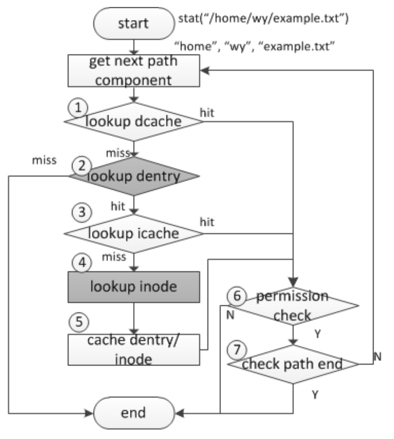


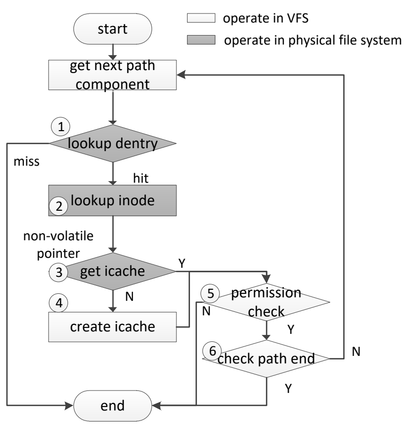
Figure 1: The percentage of time spent on path lookup with a cold cache. *VFS*: time of lookup spend in VFS. *FS*: time of lookup spend in physical file system. The numbers above the histograms are application total execution times in seconds.

System files transparently[12]. During the past decades, VFS is evolved not only to support multiple file systems, but also provide metadata caching, concurrency control, and permission check. Existing kernel based PM file systems [4, 7, 17, 25, 26] are also attached under VFS for handling metadata. However, since NVMs provide sub-microsecond access latency, we find VFS brings extra overhead when operating metadata for PM file systems. Figure 1 shows the percentages of execution times of path lookup spent in VFS and physical file system respectively. We run several command-line applications on two file systems: NOVA file system (a state-of-the-art PM file system [25]) and traditional disk-based file system ext4[1]. As shown in Figure 1, since NVM is faster than hard disk, the total application execution times on NOVA are reduced by from 28.4% to 73.4% compared to ext4. However, the percentages of execution times of

[1]The experiment setup for application running and NVM emulation is presented in Section 4.



(a) The process of path lookup in conventional VFS.



(b) The process of path lookup in ByVFS.

Figure 2: *dcache*: cached dentry metadata in VFS. *icache*: cached inode metadata in VFS.

path lookup spent in VFS on NOVA increase by from 16.5% to 459.6%. This indicates that when serving PM file systems, VFS brings extra performance overhead.

In this paper, we argue that instead of using VFS caching for metadata, one can directly access metadata in physical file systems. We take path lookup as a case study, which is a critical step for many system calls (e.g. 10%-20% of system calls involve path lookup [8].). We remove VFS metadata caching for PM file systems and discuss the arising issues, such as supporting specific system calls, concurrency control, and compatibility with conventional VFS abstraction. We implement ByVFS, an optimization of VFS by removing dentry metadata caching in VFS. We use NOVA as the underlying PM file system. We evaluate ByVFS against conventional VFS using both micro-benchmarks and command-line applications. The results show that ByVFS improves path lookup performance by 7.1%-47.9% for system calls compared to conventional VFS with cold cache. Meanwhile ByVFS performs comparably to conventional VFS with warm cache. For certain command-line applications, ByVFS reduces application execution time by up to 26.9%. We also present potential issues when shortening VFS stack.

2 Background and Motivation

Virtual File System (VFS) was originally designed to support accessing local file system and remote network file system transparently. Currently, VFS not only provides an abstraction for interacting with multiple instances of physical file systems, but also provides caching, concurrency control, and permission check when accessing files.

VFS mainly caches three types of metadata, including *superblock*, *dentry*, and *inode*. A dentry metadata (named as dcache in this paper) mainly contains file name and corresponding inode number, while an inode metadata (named as icache in this paper) mainly contains file properties (e.g. inode number and file size).

Path lookup is a critical process involved in many file system operations. We take the system call *stat* as an example to illustrate the path lookup process as shown in Figure 2(a). When serving *stat("/home/wy/example.txt")*, VFS first directly obtains the icache of the root directory ("/") as it is loaded into VFS when system starts. Then VFS looks up the dcache of the next path component ("home" here) in VFS (step 1). If the dcahe is found, VFS executes operations such as permission check and symlink processing on the directory (step 6). Otherwise, one needs to look up the dentry in physical file systems (step 2). After step 2, VFS further looks up its corresponding icache in VFS (step 3). Note that, step 3 is executed intending to check whether there already exists icache for the path component. If there is no icache in VFS, one needs to look up the inode in physical file systems (step 4). Once the inode is found in either VFS or physical file systems, one can continue step 6. VFS caches the dentry or the inode found in step 2 or step 4 by creating and initializing its structure in VFS (step 5). After step 6, VFS checks whether it reaches the end of the whole path (step 7). Here, the final file "example.txt" has not been looked up yet. Thus VFS iterates such process until it reaches the end of path. Then *stat* returns the information about the file "example.txt".

From the above example, caching both dentry and inode metadata in VFS helps improving path lookup performance for disk-based file systems by avoiding frequently accessing slow disk. However, since NVMs have read latency similar to DRAM [13, 16], accessing

DRAM cache bring performance overhead compared to directly reading NVMs. We are thus motivated to explore the potential opportunity of directly accessing metadata in PM file systems while bypassing VFS caching.

3 ByVFS

In this section, we present the design issues of ByVFS, an optimization of Virtual File System for non-volatile memory. The key idea of ByVFS is to bypass the metadata caching layer in VFS and directly access metadata in physical file systems. Figure 2(b) shows the reduced stack of path lookup in ByVFS. ByVFS looks up dentry metadata for a path component directly in PM file systems. Once found, one can obtain the related inode metadata. ByVFS remains to cache inode to hide the long NVM write latency for frequent metadata updates.

3.1 Handling dentry cache

In ByVFS, the dentry metadata is no longer cached. One directly looks up the dentry metadata of a path component in physical file systems (e.g. NOVA file system in this paper). One issue after removing dcache is the capability to support the system call *getcwd* (returns the full path of current working directory), since it is frequently used to obtain the name of parent directories. Conventionally, any dcache has a pointer referring to its parent directory. VFS can iterate dcache of all path components to obtain the full path. However, in ByVFS, dcache is removed, while the dentry metadata in physical file systems does not include pointers referring to the parent directory. Thus, we instead provide another iterating lookup approach by traversing the whole directory hierarchy to serve *getcwd*. For illustration purpose, we assume a process calls *getcwd* under the directory "/home/wy". The kernel holds a pointer referring to the inode of the current working directory for each process. Thus, one can first obtain the inode ($inode_{current}$) of the current working directory ("wy" directory here). Then, one can obtain the inode ($inode_{parent}$) of the parent directory ("home" directory here) through the dotdot hard link in current directory. By searching the inode number of $inode_{current}$ across sub-directory entries in the parent directory file, one can obtain the corresponding directory name (name is "wy" here). Meanwhile, as $inode_{parent}$ (for "home" directory here) is already obtained, the above process can be repeated until the root directory is reached. Finally, the system call *getcwd* returns the full path of working directory. We leave optimizing the efficiency of such iterating lookup as the future work.

On the other hand, dcache is conventionally used to ensure the correctness of concurrent access, and existing PM file systems, such as PMFS[7] and NOVA[25], rely on VFS for concurrency control. Since ByVFS removes dcache, the physical file systems are required to provide such guarantee, which is to be done in our future work.

3.2 Handling inode cache

Unlike removing dcache in VFS, ByVFS retains to cache inode metadata (icache) in VFS. This is because icache contains commonly-used file properties, such as file size and access time. These file properties are updated frequently. Considering NVM usually has long write latency [13, 16], persisting inode metadata through low-level file systems directly on NVM once inode metadata is modified degrades the whole system performance.

Conventionally, the dcache in VFS maintains a pointer to the icache of itself. VFS can access the icache once the dcache is found. However, in ByVFS, we need to record the memory location of icache as dcache is removed. Thus, we modify the low-level file system to add a non-volatile pointer in the inode structure, which points to icache. The pointer is created when ByVFS caches inode metadata. The pointer is set to NULL when the corresponding icache is released (e.g. evicted from VFS cache or system shutdowns normally). Note that, the pointer is non-volatile, and it becomes meaningless as long as system crashes as the pointed icache does not exist any more. We use version number to figure out meaningless pointers. We add a version number in both superblock and inode metadata of files. The version number in the superblock increases by one upon each mount. When ByVFS creates a icache for an inode, the version number in the inode is updated to the one in superblock. For each file lookup, the physical file system compares version numbers in both inode and superblock. The non-volatile pointer is considered to be meaningful if the two versions are the same. Otherwise, the physical file system re-creates a icache, and lets the pointer in the inode metadata point to the newly created icache. The physical file system updates the version number in the inode to the one currently in superblock.

3.3 Supporting multiple file systems

VFS is conventionally designed to provide abstraction for supporting physical file systems. To maintain the compatibility, we let the low-level PM file systems hold a flag **S_NVMFS_ROOT**. This flag is added into the inode of mounted directory once a PM file system is mounted. In such doing, once the flag is encountered when doing path lookup, the remaining lookup is performed in ByVFS. Otherwise, the conventional VFS lookup process is performed.

4 Evaluation

4.1 Experimental setup

We conduct all experiments on a server equipped with two Intel Xeon E5-2620 v3 processors. The memory size is 96 GB. We put 16 GB memory for DRAM and 80GB for emulating NVM. The OS is CentOS 7.0, Linux kernel 4.3.0. All experimental results are the average of at least 10 runs.

NVM emulation As real NVM are not available for us yet, we implement a hardware emulator based on the NUMA architecture with processors 0 and 1. Applications run on processor 0 but access remote memory attached to processor 1. The NUMA architecture introduces extra memory read latency (30-40 ns in our hardware) compared to accessing local memory. This is used to emulate NVM read latency which is similar to DRAM [13, 16]. For NVM write, we use hardware performance monitoring module (PMU) [1] to count the number of LLC write miss for processor 0. Periodically (e.g. every 100ms), we add stalls according to counted write misses to processor 0 to emulate long nvm write (each write incurs 600ns in this paper).

ByVFS implementation and comparison We implement ByVFS by modifying the VFS in Linux kernel 4.3.0. Currently ByVFS only implements optimization for path lookup by removing dcache in VFS. We choose the state-of-the-art NOVA[25] as the low-level PM file system. To support maintaining cached inode metadata in ByVFS, we modify inode and superblock structures in NOVA. We compare ByVFS against the conventional VFS, while we use the originally open-sourced NOVA under the conventional VFS.

Benchmarks We use 6 system calls: *open/stat/access/unlink/mkdir/rmdir* as micro-benchmarks. which all involve path lookup. Although ByVFS support multi-level directory hierarchy, we only show the results when running the micro-benchmarks in a single-level directory with 100,000 files due to the space limitation. Applications benefit from conventional VFS caching in case of metadata cache hit. VFS cache evicts metadata in case of deleting files or reduced available memory, which results in metadata cache miss. Thus, we evaluate system calls against both cold VFS cache (run the experiment once) and warm VFS cache (run the experiment twice and drop the first one) for the conventional VFS. In addition, we use 5 widely-used command-line applications in the Linux source directory to evaluate the end-to-end performance: *find* (searching for "test.c" file), *du -s* (showing whole directory size), *ls -lR* (recursively listing all file attributes), *rm -rf* (deleting all files), *cp -r* (copying whole directory from tmpfs to the test file system).

4.2 Micro-benchmarks

Figure 3 shows the execution times of tested system calls. Due to space limitation, we show the comparison between cold VFS cache and warm VFS cache only for *access* and *unlink*. When running with cold VFS cache, ByVFS outperforms conventional VFS for all system calls. ByVFS reduces execute times by 12.7%/48.1%/28.4%/50.8%/7.4%/18.4% for *open/stat/access/unlink/mkdir/rmdir*. Since ByVFS directly looks up metadata in underlying NOVA, the time VFS_lookup is reduced a lot (black parts in histograms). The remaining VFS_lookup is spent in conventional VFS for searching the upper-level directories on top of the mount point of NOVA. Besides, ByVFS reduced the dcache overhead in VFS_others.

When running with warm VFS cache, the execution time of *unlink* decreases by 38.1% in ByVFS. This is because *unlink* involves deleting files, which includes metadata operations in both VFS (e.g. deleting dcache and icache) and physical file systems (e.g. deleting dentry and reclaiming inode). In ByVFS, the metadata operations in VFS are greatly reduced (e.g. no dcache related operations), which helps to decrease the execution time. Moreover, ByVFS allows the underlying NOVA to refer to the cached inode directly using a non-volatile pointer (steps 2 in Figure 2(b)) . This makes lookup performance in ByVFS comparable to the conventional VFS with warm cache. This can also be seen from the result of the lookup-intensive operation *access*, in which ByVFS performs slightly worse.

4.3 Command-line applications

Figure 4 shows the execution time of tested applications. ByVFS reduces the execution time by 26.9%/11.6%/21.4% for *du/find/rm* respectively. As shown in Figure 4, the lookup overhead for searching dentry metadata cache (time VFS_lookup) is reduced a lot in ByVFS. However, for *ls* and *cp*, ByVFS performs slightly worse than conventional VFS, in which the execution times of *ls* and *cp* increase by 6.4% and 3.8% respectively. The two applications benefit from high VFS cache hit ratios under conventional VFS. For example, *ls* repeatedly calls *stat*, *lstat* and *lgetxattr* against files within a directory, which results in warm VFS cache. Table 1 shows the hit ratios of VFS metadata cache for all applications. The hit ratios of *ls* and *cp* reach 95%. Thanks to the design of the non-volatile pointer in ByVFS, ByVFS still provides comparable performance with warm VFS cache.

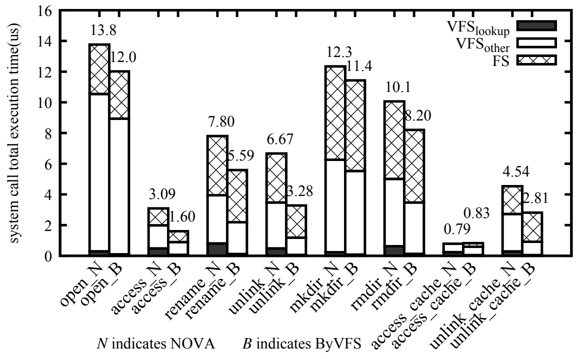


Figure 3: Execution times of system calls. *VFS_lookup*: the time of looking up dcache and icache in VFS. *VFS_others*: the time of others operations during path lookup in VFS, such as parsing path name and checking access permission. *FS*: the time spend in physical file system. The numbers above the histograms are total execution time of each system call in microseconds. $access_{cache}$ and $unlink_{cache}$: refer to the execution with warm VFS cache. Others are performed with cold VFS cache. *open* refers to creating a new file.

Table 1: VFS cache hit ratios in conventional VFS

du	find	ls	rm	cp
14.1%	69.1%	95.8%	55.5%	95.3%

5 Related Works

Persistent memory based file system. Recently, there is a number of research efforts to build file systems based on persistent memory. BPFS[4], Aerie[22], PMFS[7], SCMFS[24], and SIMFS[20] focus on minimizing software overhead by bypassing block layer and buffer cache, optimizing index structure. Bypassing buffer cache is different from bypassing metadata cache in VFS in this paper, which only helps to reduce the software stack of data I/O instead of metadata. PMFS[7] and NOVA[25] provide strong consistency with low overhead for PM file systems. NOVA-Fortis[26] further provides snapshot and fault tolerance for PM file systems. HinFS[17] focuses on hiding long NVM write latency by adding write buffer. ByVFS differs from these work by focusing on optimizing metadata operation for PM file systems. Aerie[22] also bypasses VFS by building user-level PM file system but focuses on providing flexible architecture, allowing applications to optimize file system interfaces. ByVFS instead mainly reduces extra overheads of VFS caching for metadata operations.

Optimizing metadata operations. Metadata operations in file systems usually result in small and random writes, causing write amplification to block devices. A number of research work focus on addressing the issue through buffering directory tree [15], using object stores [18], and providing write-optimized index [10, 27, 28]. Path lookup is another source of performance degradation involving metadata operations. [5, 21] add a cache on top of VFS to reduce the overhead of component-by-component translation. [5, 23, 14, 10, 27, 28] implement full path lookup in physical file systems to improve access latency in case of VFS cache miss. This paper also targets to optimize metadata operation, but specifically focuses on the overhead for PM file systems.

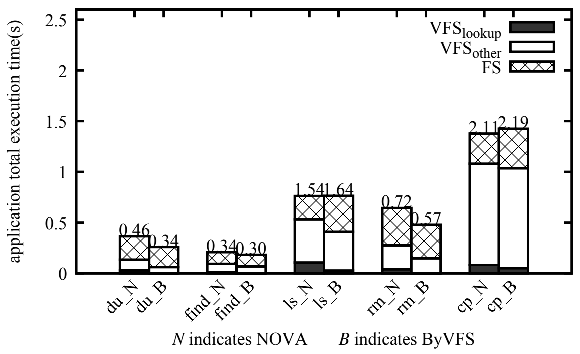


Figure 4: Execution times of command-line applications with cold cache. The time division is the same as Figure 3. The numbers above the histograms are the application execution times in seconds.

6 Conclusion and Future Work

This paper presents ByVFS, an optimization of VFS by removing dentry caching to reduce extra overhead for PM file systems. Moreover, we figure out two potential optimization opportunities: the concurrency control originally guaranteed in VFS is now shifted to low-level file systems, which is not well supported in existing PM file systems. The other is efficiently and completely supporting various system calls, which may require designing highly efficient index structures in persistent memory. We leave both issues as our future work.

7 Acknowledgements

We would like to thank the anonymous reviewers for their feedback and guidance. This work is supported by National Key Research and Development Program of China under grant No. 2016YFB1000302, National Science Foundation of China under grant No. 61502448, and Alibaba Cloud Computing Limited.

References

[1] Intel 64 and ia-32 architectures software developers manual. Volume 3B: System Programming Guide, Part 2 http://www. developer.intel.com.

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

[3] Baek, I., Lee, M., Seo, S., Lee, M., Seo, D., Suh, D.-S., Park, J., Park, S., Kim, H., Yoo, I., et al. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International* (2004), pp. 587–590.

[4] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Derrick, C. Bpfs:better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), SOSP '09, ACM, pp. 133–146.

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

[6] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. Persistent memory file system. https://github.com/linux-pmfs/pmfs.

[7] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems* (2014), EuroSys '14, ACM, pp. 15:1–15:15.

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

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

[10] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M., Farach-Colton, M., Johnson, R., Kuszmaul, B. C., and Porter, D. E. Betrfs: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (2015), FAST'15, USENIX Association, pp. 301–315.

[11] kawahara, t. Scalable spin-transfer torque ram technology for normally-off computing. *IEEE Des. Test 28*, 1 (Jan. 2011), 52–63.

[12] Kleiman, S. R. Vnodes: An architecture for multiple file system types in sun unix. In *USENIX Association: Summer Conference Proceedings, Atlanta 1986, pp. 1-10.*

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

[14] Lensing, P. H., Cortes, T., and Brinkmann, A. Direct lookup and hash-based metadata placement for local file systems. In *Proceedings of the 6th International Systems and Storage Conference* (2013), SYSTOR '13, ACM, pp. 5:1–5:11.

[15] Lu, Y., Shu, J., and Wang, W. Reconfs: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST'14, USENIX Association, pp. 75–88.

[16] M. F. Chang, J. J. W., T. F. Chien, Y. C. L., T. C. Yang, W. C. S., Y. C. King, C. J. L., K. F. Lin, Y. D. Chih, S. N., and 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 Proceedings of 2014 IEEE International Solid-State Circuits Conference, ISSCC'14, pp. 332–333* (2014).

[17] Ou, J., Shu, J., and Lu, Y. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys '16, ACM, pp. 12:1–12:16.

[18] Ren, K., and Gibson, G. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013), USENIX, pp. 145–156.

[19] S. Raoux, G. Burr, M. B., C. Rettner, Y. Chen, R. S., M. Salinga, D. K., S.-H. Chen, H. L. L., and Lam, C. Phase-change random access memory: A scal- able technology. *IBM Journal of Research and Development, 52(4.5):465–479* (2008).

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

[21] Tsai, C.-C., Zhan, Y., Reddy, J., Jiao, Y., Zhang, T., and Porter, D. E. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, ACM, pp. 441–456.

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

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

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

[25] Xu, J., and Swanson, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (2016), FAST'16, USENIX Association, pp. 323–338.

[26] Xu, J., Zhang, L., Memaripour, A., Gangadharaiah, A., Borase, A., Da Silva, T. B., Swanson, S., and Rudoff, A. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, ACM, pp. 478–496.

[27] Yuan, J., Zhan, Y., Jannen, W., Pandey, P., Akshintala, A., Chandnani, K., Deo, P., Kasheff, Z., Walsh, L., Bender, M., Farach-Colton, M., Johnson, R., Kuszmaul, B. C., and Porter, D. E. Optimizing every operation in a write-optimized file system. In *Proceedings of 14th USENIX Conference on File and Storage Technologies* (2016), FAST'16, USENIX Association, pp. 1–14.

[28] Zhan, Y., Conway, A., Jiao, Y., Knorr, E., Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R.,

PORTER, D. E., AND YUAN, J. The full path to full-path indexing. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (2018), FAST'18, USENIX Association, pp. 123–138.

[29] ZHANG, S., CATANESE, H., AND WANG, A. A.-I. The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (2016), FAST'16, USENIX Association, pp. 15–22.