ADAM: An Adaptive Directory Accelerating Mechanism for NVM-Based File Systems

Xin Cui, Linpeng Huang(✉), and Shengan Zheng

Shanghai Jiao Tong University, Shanghai, China
{cuixindd,lphuang,venero1209}@sjtu.edu.cn

Abstract. Byte-addressable non-volatile memory (NVM) offers fast, fine-grained random access to persistent storage, which revolutionizes the architecture design of file systems. Existing NVM-based file systems seldom optimize the directory-access performance despite that directory operations significantly impact application performance. These file systems still follow the traditional design of multi-level directory namespace which is inadequate for byte-addressable NVM and involves redundant access overhead.

In this paper, we propose an adaptive directory accelerating mechanism (ADAM) for NVM-based file systems. ADAM analyzes different directory states including read/write frequency and size and then builds adaptive full-name directory namespace (AFDN) areas as well as an evolving strategy. Compared with multi-level directory namespace, AFDN areas offer fast read access and low write latency. Besides, the evolving strategy helps AFDN areas maintain a stable performance during system runtime. We implement ADAM on NOn-Volatile memory Accelerated (NOVA) log-structured file system and build an efficient hybrid index for DRAM/NVMM architecture. Experiments show that NOVA with ADAM achieves up to 43% latency reduction and 76% throughput improvement, compared with original NOVA. Moreover, ADAM generally outperforms other state-of-the-art NVM-based file systems in various tests.

Keywords: Non-volatile memory · File system
Adaptive directory mechanism

1 Introduction

Emerging non-volatile memory (NVM) techniques promise to offer both byte-addressability and persistent memory storage, e.g., spin-torque transfer, phase change [13], resistive memories [19], and Intel and Micron's 3D-XPoint [11] technology, which revolutionize the architecture design of file systems. As a result, many file systems are proposed to exploit the performance benefit of NVM, such as NOVA [16], PMFS [6], SCMFS [14] and HMVFS [20]. These NVM-based file

J. Vaidya and J. Li (Eds.): ICA3PP 2018, LNCS 11334, pp. 578–592, 2018.
https://doi.org/10.1007/978-3-030-05051-1_40

systems significantly improve the performance through redesigning structures which are considered to be inefficient or unsuitable for NVM characteristics.

Directory operations have a large impact on application performance [9,10,12]. There are two main traditional designs of directory namespace: multi-level directory namespace and full-name directory namespace. Multi-level directory namespace supports fast renames but increases read latency because of recursive scans [18]. This kind of mechanism tightly couples directory metadata and inode since the directory index and update involve both directory names and inode numbers, thus it brings a large overhead in consistency guarantee. On the contrary, full-name directory namespace offers fast direct access but brings heavy write overhead in renames, because changes of the path name may cause a large amount of small random writes which are amplified [8] in traditional disk-based file systems.

It is a trade-off to choose one of the above mechanisms for a disk-based file system, however, both of which are not suitable for NVM-based file systems. For multi-level directory namespace, it is unnecessary to sacrifice directory read performance because the write amplification problem no longer exists in byte-addressable NVM. Besides, the tight coupling of directory metadata and inodes causes redundant access to inodes and large NVM writes which have a negative impact on access performance. For these reasons, full-name directory namespace is more appropriate for NVM. However, problems arise when implementing full-name directory namespace on NVM-based file systems since it naturally brings a large number of random writes when renaming directories. Current NVM-based file systems seldom consider these issues and implement their directory mechanisms following the improper traditional design of multi-level directory namespace.

In this paper, we propose ADAM, an adaptive directory accelerating mechanism, which provides fast read access as well as low write overhead. ADAM introduces a new design of directory layout called adaptive full-name directory namespace (AFDN). Each AFDN area defines a root directory with a full path name and sub-directories with adaptive path names relative to the root directory name. The adaptive path names not only avoid the rename overhead when the root directory name changes, but also improve both read and write performance of directories by decoupling directory metadata and inodes.

Moreover, ADAM classifies directory states based on read frequency, write frequency and directory size. We introduce an evolving strategy which defines three evolvements for AFDN areas: *splitting*, *merging* and *inheriting*. The evolving strategy ensures that each AFDN area is selected reasonably and keeps the division of AFDN areas always be adaptive to the directory changing states during system runtime.

We implement ADAM on NOVA, a hybrid DRAM/NVMM file system, and build an efficient hybrid index includes hash tables and radix trees.

We conclude our contributions as follows:

- We analyze the previous, unsuitable directory designs in NVM-based file systems and propose ADAM, an adaptive directory accelerating mechanism, which offers fast directory access as well as low write overhead.
- We introduce AFDN, adaptive full-name directory namespace which labels different directory states in the state-map and reduces directory access overhead by decoupling directories/files and inodes.
- We propose an evolving strategy to keep the most beneficial namespace division and a stable performance of AFDN areas during system runtime by involving three evolvements for AFDN areas.
- We implement an efficient index including radix trees in DRAM and multi-level hash tables in NVM. The hybrid index ensures the best utilization of both larger, persistent NVM and faster, volatile DRAM.
- We implement ADAM on NOVA [16] and evaluate the performance on several benchmarks. The results show that NOVA with ADAM achieves up to 43% latency reduction and 76% throughput improvement compared with original NOVA and generally outperforms other state-of-the-art NVM-based file systems.

The rest of this paper is organized as follows. We introduce the ADAM design in Sect. 3 and implementation details in Sect. 4. We discuss the evaluation results in Sect. 5. Finally, we provide related works in Sect. 6 and conclude our paper in Sect. 7.

2 Background and Motivation

2.1 Non-volatile Memory

Table 1 summarizes the characteristics of different memory technologies. NVM provides slightly shorted read latency to DRAM, while its write latency is apparently longer than DRAM. Similar to NAND Flash, the write endurance of NVM is limited especially for PCM. But unlike flash that is block-addressable, NVM is byte-addressable. Besides, NVM has high performance of random access like DRAM, which is better than traditional Flash.

Table 1. Comparison of different memory characteristics [7,17]

Category	Read latency	Write latency	Write endurance	Byte- addressable
DRAM	60 ns	60 ns	10^{16}	Yes
PCM	50–70 ns	150–1000 ns	10^{9}	Yes
ReRAM	25 ns	500 ns	10^{12}	Yes
NAND Flash	35µs	350µs	10^{5}	No

In hybrid DRAM/NVMM architecture, the drawback of NVM is the long write latency while the advantage of NVM is byte-addressability [1]. Thus, our work aims to accelerate the directory access by designing a new directory mechanism, which suits NVM byte-addressability and reduces the number of writes.

2.2 High Directory Access Overhead

The directory mechanism significantly impacts file system performance since directory operations are commonly involved in many applications such as *tar*, *git-clone*, and *git-diff*. However, there are three problems in current NVM-based file systems directory mecahnisms: (i) Current NVM-based file systems mainly follow the traditional directory design, which is suitable for the block-addressable disk but not byte-addressable NVM. Though NVM solves the write-amplification problem, known as the bottleneck of full-name directory namespace, the write overhead remains heavy when renaming the full path names, and slows down directory operations. (ii) The directory name is not unique and can not be considered as an independent index in current multilevel directory namespace. Therefore, directory metadata and inodes are tightly coupled, which increases the write overhead to keep consistency for inode and directory metadata. (iii) The states of directories, e.g., read/write-frequency and directory size, always change in system lifetime and have a different impact on system performance while it is not well utilized in current file systems.

To overcome the above problems, it is necessary for us to design an NVM-friendly directory mechanism which accelerates directory access. The new directory design needs to weaken the coupling between directories and inodes so not only do we reduce the write overhead of consistency guarantee but also we reduce the access latency to directories and files. Besides, it is beneficial to utilize different directory states and design an adaptive mechanism to make the system maintain the best performance during runtime.

3 Design

3.1 ADAM Layout

Figure 1 shows the design of ADAM layout which contains four parts: AFDN areas, Multi-level hashtables, AMT areas and DRAM cache trees.

AFDN Areas. ADAM initializes each AFDN area in a 2 MB block array. Each directory and file entry is initially aligned on a 128-Byte boundary. ADAM assigns new directory/file entries to the AFDN area in a round-robin order, so that directories/files are evenly distributed among AFDN areas.

Each AFDN area has a root directory with a full path name and plenty of sub-directories which stores an adaptive path name relative to the root directory name. Therefore, AFDN decouples directory metadata and inodes by using this kind of unique path names. Moreover, renaming a directory that is the root of an AFDN area causes no write overhead to the sub-directories. Every AFDN area has an area_ID which is the hash value of the root name. As shown in Fig. 2, an AFDN area contains two parts: directory/file entries and a state-map.

State-Map. A state-map is a two-bit bitmap, in which each slot labels the state of an entry in the AFDN area. As shown in Fig. 2, the state-map is in the head of an AFDN area. A 2-bit slot in a state-map can represent four different states:

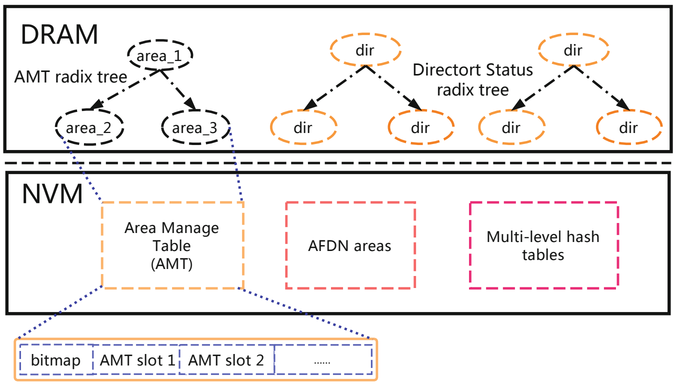


Fig. 1. ADAM layout

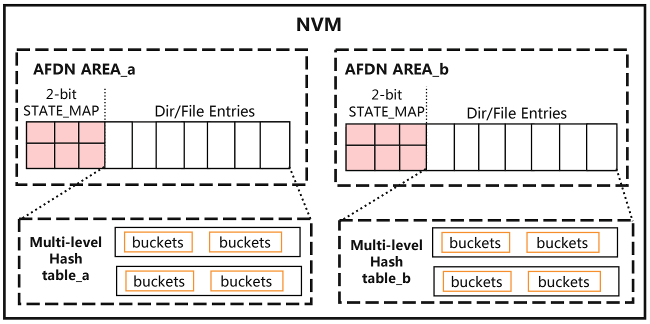


Fig. 2. AFDN areas and hash tables

00 represents an invalid entry in AFDN area, *01* represents a valid file or a *cold* directory, *10* and *11* represent *warm* directories and *hot* directories respectively. Hot labels represent higher access overhead. ADAM gains the best performance when *hot* directories are selected as root directories in order to reduce rename overhead and read latency. The computation and design details of the state labels are introduced in Sect. 3.3.

Multi-level Hash Tables. We build multi-level hash tables for each AFDN area to support fast directory/file locating as shown in Fig. 2. A hash entry contains a hash value of an adaptive path name and the position of an AFDN entry as a hash pair. To reduce the hash table size, each hash entry contains a valid bit, thus we can reuse invalid hash entries for new files and directories.

Area Manage Table (AMT). The AMT records the addresses of both AFDN areas and corresponding hash tables, and it is allocated and initialized as a 4 KB block as shown in Fig. 1. An AMT has two parts: a bitmap and 54-Byte slots. The bitmap manages the allocations and releases of the slots and each slot is cached in DRAM in order to reduce index latency.

DRAM Cache Trees. ADAM keeps two kinds of radix tree in DRAM: an AMT radix tree and status radix trees. The status radix tree is used to calculate the directory states and it records the read frequency, write frequency and the directory size for each accessed directory.c The leaves of the AMT radix tree store the same information as NVM AMT slots. The AMT radix tree is an important part of the hybrid index which is introduced in Sect. 3.2.

3.2 Hybrid Index

Since the index structure has a great influence on hybrid DRAM/NVMM storage systems [3], we pay close attention to the directory searching and design an efficient hybrid index.

The hybrid index contains three parts: AFDN hash tables, AMT and DRAM radix trees. The multi-level hash tables provide a good solution to hash conflicts and offer a fast access at the same time. We use a radix tree because there is a mature, well-tested, widely-used implementation in the Linux kernel. Throughout DRAM radix tree searching, we can easily and quickly find the right AFDN area and corresponding hash tables. According to Table 1, DRAM owns the fastest access speed which is suitable to store the most frequently accessed index structures. It is easy to see that the AMT has a high-frequency access with small space requirement for DRAM space.

Applications can quickly locate the directory and file with our hybrid index through the following step: (i) calculate the AFDN area_ID with the path name; (ii) searching the right leaf in the AMT radix tree; (iii) find the address of AFDN area and the corresponding hash table; (iv) locate the directory by the hash key-value pair.

Table 2. Directory states and state-map labels

States			Label	Bit-value
Size	Write	Read		
small	frequent	frequent	warm	10
		rare		
	rare	frequent	cold	01
		rare		
large	frequent	frequent	hot	11
		rare		
	rare	frequent		
		rare	warm	10
invalid			invalid	00

3.3 States of Directories

We consider read frequency, write frequency and size as three important states to label different directories. The state of a directory changes among different applications and is categorized into 5 groups: {*small size*, *large size*, *frequently read*, *rarely read*, *frequently write*, *rarely write*}. The write overhead is caused mainly by directory renames and creates. And we define 3 labels to classify different directories, they are {*hot*, *warm*, *cold*}. Different labels represent different access overhead, which means a *hot* directory brings the largest access overhead. Table 2 lists the mapping from directory states to labels.

Three Important States: Read frequency is an important state because selecting a large, *frequently read* directory to be a root directory of a new AFDN area will greatly reduce the access latency to this new root and its sub-directories. Root directories or children of root directories only need one calculation to find which AFDN area the target file belongs to, while others need more recursive calculations for the correct AFDN area. The worst case involves the same number of calculations as the number of recursive scans in traditional multi-level directory namespace.

Since write overhead affects most to the performance of NVM, we consider write frequency as the prime factor in calculating directory labels.

Size is also an important factor because larger directories are more likely to generate a large amount of write overhead.

Calculation: To calculate the read frequency, ADAM first counts access times for each accessed directory and records the value in DRAM status tree. And then ADAM calculates the average read time of the AFDN area. Third, ADAM marks *frequently read* if the directory access time is more than the average value or marks *rarely read* if less than the average value.

Since rename brings the largest write overhead, ADAM marks a directory *frequently write* once it is renamed. A *rarely write* directory is the one which is never renamed during the runtime.

In size calculation, ADAM introduces $Size\_Max$ and $Size\_Min$ to determine the size state of a directory. Each directory that is not the root of its AFDN area has the size at most $Size\_Max$. Each area has size at least $Size\_Min$. In our implementation, $Size\_Max = Size\_Min = 512\,\text{KB}$ [18]. ADAM marks a directory *large* when its size is larger than $Size\_Max$, and *small* when it is smaller than $Size\_Min$.

According to the above calculating rules and the mapping ralation shown in Table 2, we update the values stored in NVM state-map every 5 s before executing our evolving strategy which is introduced in Sect. 3.4.

3.4 Evolving Strategy

For purpose of adapting to changing states during system runtime, we define three evolvements among AFDN areas: *splitting*, *merging* and *inheriting*, shown

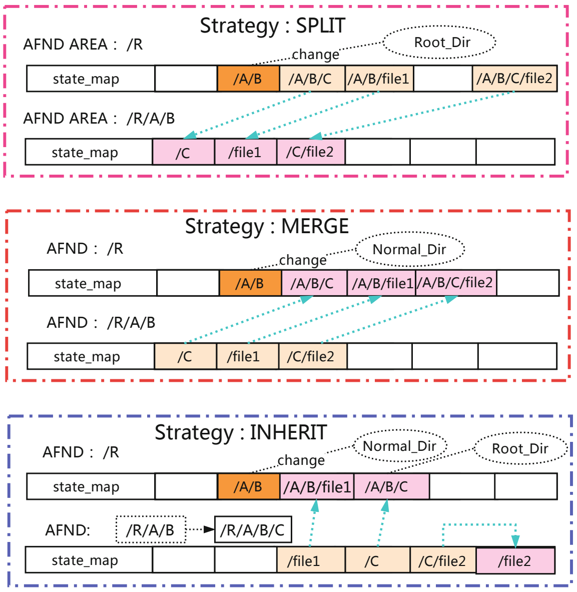


Fig. 3. Adaptive evolving strategy

in Fig. 3. ADAM executes these evolvements on the base of labels recorded in the state-map.

Splitting is the way to create new AFDN areas as well as to select corresponding root directories. When splitting happens, ADAM copies all the sub-files and sub-directories to the new allocated AFDN area and then changes their adaptive names. The root directory still stays in the original AFDN area. ADAM executes splitting when the system periodically checks labels in a state-map and finds a *hot* directory which brings a large access overhead.

Merging happens when an old AFDN area is small and seldom accessed. This kind of AFDN area wastes nearly a 2MB block array because the invalid entries take up the most space and the rest sub-directories and sub-files are seldom visited or renamed. Once we find a root directory satisfies the conditions we will merge the directories and files back to their parent AFDN area and then free this area to make it reusable.

Inheriting defines an AFDN evolvement that a strong sub-directory replaces its parent root directory. The strong sub-directory represents a kind of the directory which owns large size and contributes to the vast majority access to the AFDN area.

For an inheriting evolvement, the operations follow the steps shown in Fig. 3. First, ADAM detects the directory $/C$, which is a strong sub-directory in AFDN $/R/A/B$ and meets the inheriting conditions. Second, ADAM merges $/C$ and its brother-directories/files back to the parent AFDN area $/R$ and sets their new adaptive path names. Third, ADAM changes the type of the old root directory

/A/B to normal directory and sets */A/B/C* as the new root directory. Fourth, ADAM changes the full path name of AFDN area from */R/A/B* to */R/A/B/C*. Fifth, ADAM resets the state-map for both AFDN */R* and AFDN */R/A/B/C*.

The evolving strategy keeps the system adaptive to different states and maintains the best and stable performance during runtime. In our implementation, ADAM checks the state-map of all accessed AFDN areas every 5 s at the background and then updates the corresponding state-map before executing the evolving strategy. The evolving strategy brings small space overhead for copying splitting directories entries. However, compared with the total size of files and directory when splitting happens, it is reasonable to sacrifice the small space overhead to accelerate directory access and keep the system stable during runtime.

4 Implementation

We implement ADAM on NOVA [16], a state-of-the-art log-structured NVM-based file system. Note that ADAM can also be implemented in other NVM-based file systems, such as PMFS [6] or HMVFS [20] to accelerate directory access and reduce write overhead.

Optimized Index. Considering that DRAM and NVM show different characteristics, the index in hybrid DRAM/NVMM file system should utilize the strength of both faster DRAM and larger NVM. It is not yet well-balanced in NOVA as it caches all directories and files in DRAM [16], which take too much space of DRAM. Inspired by HiKV [15], we implement an optimized hybrid directory index with large hash tables in NVM and frequently accessed radix trees in DRAM which only caches the index of AFDN areas to save DRAM space overhead. Moreover, compared with the linear linked list for directory log-entries in NOVA, hash tables can greatly accelerate name searching for directories and files.

Efficient Logging. NOVA involves large overhead for keeping the consistency between DRAM radix trees and NVM structures since it copies all the directories and files into DRAM. The consistency overhead includes checking the large radix tree and journalling for logs. ADAM reduces the scale needed to write logs. In our mechanism, we atomically update 2-bit slots of the NVM state-map by writing to a log buffer first. The log buffer size is fixed to the same size of a state-map, which is much smaller than keeping a large number of log entries for each cached directory and file in NOVA. Under this consideration, ADAM greatly reduced the consistency overhead compared with NOVA. Besides, we enforce write ordering by using instructions of *mfence* and *clflush* in ADAM which are widely implemented in NVM-based storage systems [5,16] to ensure consistency.

Lightweight Update. Since multi-level directory namespace tightly couples directories and inodes, the file systems which follows this traditional mechanism brings heavy write overhead in frequent directory/inode logging and updating. However, directories in ADAM with adaptive path names are able to be found and updated independently, which avoids the redundant updates to and caused by inode updates. Therefore, ADAM greatly reduces the frequency of updating and minimized the directory update overhead.

5 Evaluation

5.1 Experimental Setup

We evaluate the performance of ADAM with NOVA against PMFS, original NOVA, and EXT4-DAX. We conduct experiments on a commodity server with 64 Intel Xeon 2 GHz processors and 128G DRAM, running Linux 4.3.0 kernel, and we change the kernel to 3.11.0 for PMFS. For EXT4-DAX, we use ramdisk carved from DRAM and configure 64 GB as ramdisk to simulate NVM. For PMFS and NOVA, we reserve 64 GB of memory using the grub option memmap. User processes and buffer cache use the rest of the free DRAM space. Since NVM has similar read performance but apparently slower write performance compared with DRAM, we introduce extra latencies for NVM write to emulate the longer write latency of NVM. We consider the write latency as 500ns which is the average value of different NVMs shown in Table 1.

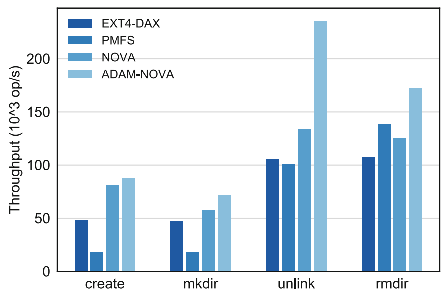
Table 3. Micro-benchmark characteristics

Name	Workload
filetest	(i) create (10^4) (ii) unlink (10^4)
dirtest	(i) mkdir (10^4) (ii) rmdir (10^4)

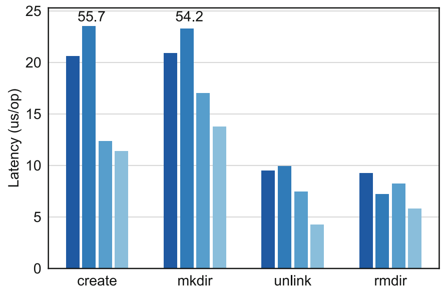
5.2 Write Performance Improvement

Micro-benchmarks. We use two single-threaded micro-benchmarks to evaluate the write performance of ADAM-NOVA, as shown in Table 3. The *filetest* create 10^4 files in one directory and then deletes all of them. The *dirtest* is similar to the *filetest* but the operated objects are directories instead of files. Our micro-benchmarks are write-sensitive since both of the create and delete operations for directories and files involve large write overhead. All the results are averaged over five runs.

Figure 4 shows the experimental results of *filetest* and *dirtest*, with throughput and latency in (a) and (b) respectively. Among the assessed operations *create*, *mkdir*, *unlink*, and *rmdir*, ADAM-NOVA consistently achieves the best performance. In *create* and *mkdir* tests, compared with NOVA, ADAM-NOVA increases throughput by 8.6% and 24% and reduces latency by 7.7% and 19%. In *unlink* and *rmdir* tests, ADAM-NOVA outperforms NOVA by 76% and 37% in throughput and outperforms NOVA by 43% and 29% in latency.



(a) Throughput



(b) Latency

Fig. 4. Throughput and latency of file system operations

We attribute the write performance improvement to the reduction of NVM writes. ADAM offers each directory and each file an adaptive path name which makes them independent of their inodes. Thus, system calls is excuted without redundant access to inodes which increases the total access latency. On the contrary, directories/files are tightly coupled with inodes in original NOVA. In order to quickly update the directory entry, NOVA maintains two pointers in inodes which point to the coupling directory entries. Such pointers produce large write overhead once the old directory is changed or a new directory is appended.

Except for excessive write operations, PMFS has high latency and low throughput because the cost of write overhead grows linearly with the number of directory entries. This is notable for *create* and *mkdir* since one insertion to directory needs to scan all existing dentries to find an available slot. ADAM reduces the latency by 74% to 79% for *create* and *mkdir* compared with PMFS. EXT4-DAX leverages hashed B-tree to speed up directory access, thus it achieves better performance than PMFS for *create* and *mkdir*, and in these cases, ADAM outperforms EXT4-DAX by 34% to 44%.

5.3 Read Performance Improvement

Filebench. We evaluate the read performance of ADAM-NOVA with a real-world application by running a *Listdir* workload in Filebench. We choose the *Listdir* workload of 10000 directories and run it with 1 to 10 threads. We run these experiments multiple times and report the average results.

As shown in Fig. 5, the performance of *ListDirs* increases rapidly when thread $\leq$ 4. ADAM-NOVA performs better than original NOVA and EXT4-DAX. ADAM improves up to 9.7% throughput compared with original NOVA and outperforms EXT4-DAX by up to 41%.

ADAM shows faster read than original NOVA because (i) each directory in indexed by multilevel hash tables; (ii) each directory in NVM is read directly by adaptive path name without recursive scans. NOVA also performs well when thread increases, since it leverages in-DRAM radix trees to manage directories. However, these radix trees consume much more DRAM space which is optimized

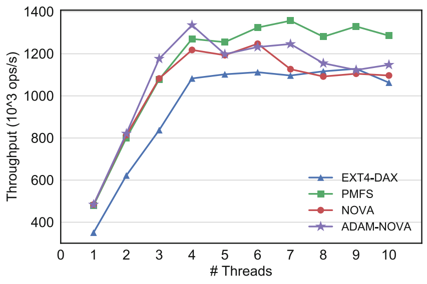


Fig. 5. Throughput of Filebench

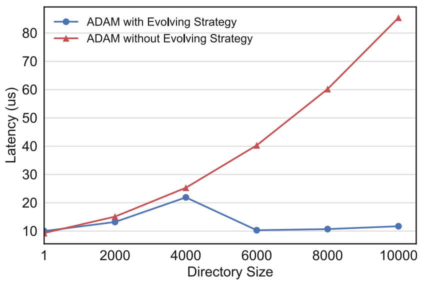


Fig. 6. Rename latency comparison

in ADAM by a hybrid index. EXT4-DAX performs the worst because the hashed B-tree is not suitable for the in-memory file system which is mainly used to optimize the disk I/O.

5.4 Benifits of Strategy

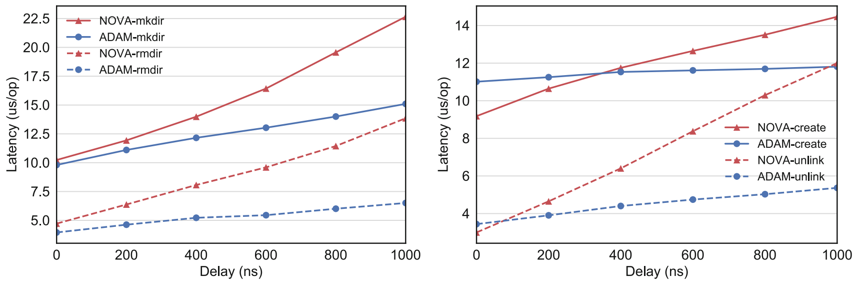
We evaluate the directory *rename* performance for ADAM with evolving strategy and ADAM without evolving strategy. Then we choose one directory named Dir-A in dirtest microbenchmark and create 1–10000 subfiles. Then we increasing the access time of Dir-A by accessing this directory multiple times to make it a read frequently directory. Note there is no difference to make a *large* directory *hot* by increasing the read frequency or changing the write frequency.

As shown in Fig. 6, the *rename* latency is similar for ADAM with evolving strategy and ADAM without evolving strategy when the number of subfiles is $\leq$ 4000. It is decreased in ADAM with evolving strategy when the number of subfiles is $>$ 4000. Because in our implementation, the size of a directory is considered *large* when the size of it takes more than 512 KB. ADAM with evolving strategy performs a low and stable rename latency compared with ADAM with evolving strategy. For ADAM without evolving strategy, the rename latency keeps increasing when the number of subfiles increases.

ADAM with evolving strategy performs lower rename latency because the Dir-A is split to be a root directory of a new AFDN when it is *large* and its label becomes *hot*. The rename of root directory involves no overhead to sub-files and sub-directories. For ADAM without evolving strategy, the rename of directory cause a large rename overhead for its sub-files and sub-directories. Therefore the evolving strategy shows great benefits to reduce the rename latency and keeps system stable during runtime.

5.5 Sensitivity to Different NVMs

Although NVM has similar read latency compared with DRAM, different NVM technologies have different write latencies which are all longer than DRAM. NVM built by PCM [13] and 3DX-Point [11] is expected to have 150ns to 1000ns write latency according to Table 1, while the write latency of DRAM is 60ns. We simulate different NVM technologies by inserting different delays [5] to evaluate the sensitivity of both ADAM-NOVA and original NOVA.



(a) Directory operations (b) File operations

Fig. 7. Latency with different delays

Figure 7 shows the latency of directory operations and file operations in our micro-benchmarks with different delays. In different cases of *mkdir*, *rmdir*, *create*, *unlink*, original NOVA shows drastically increasing latency, while ADAM-NOVA performs more stable because of the reduction of writes. Increasing the delay from 0 to 1000ns, the latency of original NOVA increases 1.

21% for *mkdir*, 193% for *rmdir*, 57% for *create*, 291% for *unlink* compared with its original values, while the corresponding increases of ADAM-NOVA are 54% for *mkdir*, 65% for *rmdir*, 7% for *create*, 56% for *unlink*.

ADAM performs better on involving fewer NVM writes. Because ADAM provides adaptive path names for each directory and file, which decouples directories/files and inodes. In contrast, the design of multi-level directory namespace in original NOVA tightly couples directories/files and inodes which involves a large number of redundant write for consistency guarantees during system calls. Therefore, ADAM performs better scalability to different NVMs.

To summarize, ADAM achieves a significant improvement for both read and write performance compared with original NOVA, PMFS, and EXT4-DAX. Besides, ADAM successfully reduces the rename overhead by executing the evolving strategy which keeps the systems adaptive to different states during system runtime. Moreover, ADAM is more suitable for NVM characteristics which are decided by different write latencies.

6 Related Work

NOVA is a state-of-the-art log-structured NVM-based file system, which performs well in file/directory access and atomic updates. However, NOVA implements their directory mechanism following the traditional multi-level directory namespace which is not suitable for NVM characteristics and involves long access latency. We implement ADAM in NOVA to optimize the problems and experiments show that we greatly reduce the access latency.

BetrFS 0.2 [18] uses write-optimized dictionaries [2] and proposes zones to optimize file system performance. However, BetrFS 0.2 is designed for disk-based storage and uses write-optimized dictionaries to combine the small random writes into large sequential writes which are not necessary for NVM with byte-addressable random access. Besides, the detection of zones is only based on the size of directories and files, which is not accurate in zone-root selection.

HiKV [15] introduces an efficient hybrid index for DRAM/NVMM KV-store [4] mixture systems. Our mechanism optimizes it in NVM-besed file systems by implementing radix trees in DRAM which suits well in Linux kernel and using multi-level hash tables to avoid hash conflicting.

7 Conclusion

In this paper, we propose ADAM, an adaptive directory accelerating mechanism, which suits byte-addressable NVM and offers fast read performance as well as low write latency for NVM-based file systems. ADAM is adaptive to different states of directories and keeps good, stable performance during system runtime. We also implement an efficient hybrid index to combine the best characteristics of DRAM and NVM. Our experiments show that NOVA with ADAM achieves up to 43% latency reduction and 76% throughput improvement compared with original NOVA and generally performs better than other state-of-the-art NVM-based file systems.

Acknowledgments. We thank anonymous reviewers for their insightful and helpful comments, which improve the paper. This work is supported by the National Key Research and Development Program of China (No. 2018YFB10033002) and the National Natural Science Foundation of China (No. 61472241).

References

1. Condit, J., et al.: Better I/O through byte-addressable, persistent memory. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, 11–14 October 2009, pp. 133–146 (2009). https://doi.org/10.1145/1629575.1629589
2. Conway, A., et al.: File systems fated for senescence? Nonsense, says science! In: FAST, pp. 45–58 (2017)
3. Debnath, B., Haghdoost, A., Kadav, A., Khatib, M.G., Ungureanu, C.: Revisiting hash table design for phase change memory. In: Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads, INFLOW 2015, Monterey, California, USA, 4 October 2015, pp. 1:1–1:9 (2015). https://doi.org/10.1145/2819001.2819002
4. Decandia, G., et al.: Dynamo: Amazon's highly available key-value store. In: ACM SIGOPS Symposium on Operating Systems Principles, pp. 205–220 (2007)

5. Dong, M., Chen, H.: Soft updates made simple and fast on non-volatile memory. In: 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, 12–14 July 2017, pp. 719–731 (2017). https://www.usenix.org/conference/atc17/technical-sessions/presentation/dong
6. Dulloor, S.R., et al.: System software for persistent memory. In: Proceedings of the Ninth European Conference on Computer Systems, p. 15. ACM (2014)
7. Lee, B.C., Ipek, E., Mutlu, O., Burger, D.: Architecting phase change memory as a scalable dram alternative. In: ACM SIGARCH Computer Architecture News, vol. 37, pp. 1–13 (2009)
8. Lee, E., Kim, J., Bahn, H., Lee, S., Noh, S.H.: Reducing write amplification of flash storage through cooperative data management with nvm. ACM Trans. Storage (TOS) **13**(2), 12 (2017)
9. Lensing, P.H., Cortes, T., Brinkmann, A.: Direct lookup and hash-based metadata placement for local file systems. In: Proceedings of the 6th International Systems and Storage Conference, p. 5. ACM (2013)
10. Lu, Y., Shu, J., Wang, W.: ReconFS: a reconstructable file system on flash storage. In: FAST, vol. 14, pp. 75–88 (2014)
11. Newsroom, I.: Intel and micron produce breakthrough memory technology (2015)
12. Tsai, C.C., Zhan, Y., Reddy, J., Jiao, Y., Zhang, T., Porter, D.E.: How to get more value from your file system directory cache. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 441–456. ACM (2015)
13. Wong, H.S.P., et al.: Phase change memory. Proc. IEEE **98**(12), 2201–2227 (2010)
14. Wu, X., Reddy, A.: SCMFS: a file system for storage class memory. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, p. 39. ACM (2011)
15. Xia, F., Jiang, D., Xiong, J., Sun, N.: HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In: 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, 12–14 July 2017, pp. 349–362 (2017). https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia
16. Xu, J., Swanson, S.: NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In: FAST, pp. 323–338 (2016)
17. Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K.L., He, B.: NV-tree: reducing consistency cost for NVM-based single level systems. FAST, vol. 15, pp. 167–181 (2015)
18. Yuan, J., et al.: Optimizing every operation in a write-optimized file system. In: FAST, pp. 1–14 (2016)
19. Zangeneh, M., Joshi, A.: Design and optimization of nonvolatile multibit 1T1R resistive RAM. IEEE Trans. Very Large Scale Integr. Syst. **22**(8), 1815–1828 (2014)
20. Zheng, S., Huang, L., Liu, H., Wu, L., Zha, J.: HMVFS: a hybrid memory versioning file system. In: 2016 32nd Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–14. IEEE (2016)