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

Xin Cui
Shanghai Jiao Tong University
cuixindd@sjtu.edu.cn

Linpeng Huang
Shanghai Jiao Tong University
lphuang@sjtu.edu.cn

Shengan Zheng
Shanghai Jiao Tong University
venero1209@sjtu.edu.cn

ABSTRACT

Various new-designed file systems are proposed after the emerging of byte-addressable non-volatile memories(NVM). However, the directory architecture in NVM-based file systems is still the traditional design of multilevel namespaces, which is designed for disk-based file systems and is not suitable for NVM.

In this paper, we propose ADAM, an adaptive directory accelerating mechanism for NVM-based file systems, to offer fast directory read while reducing write overhead. ADAM provides a novel directory layout, an adaptive full directory-namespace, which is suitable for byte-addressable NVM and introduces a self-adaptive strategy to maintain a consistent well performance during system runtime. We label different states of directories in our novel designed state_map and optimize the directory access on the base of different states. We implement ADAM on NOVA and design an efficient hybrid index for Hybrid DRAM/NVM. Experiments show XXXXX.

KEYWORDS

NVM,directory accelerating, file system, adaptive, strategy

ACM Reference Format:
Xin Cui, Linpeng Huang, and Shengan Zheng. 1997. ADAM: An Adaptive Directory Accelerating Mechanism for NVM-Based File Systems. In *Proceedings of ACM Woodstock conference (WOODSTOCK'97)*, Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Emerging non-volatile memory (NVM) technologies such as spin-torque transfer, phase change, resistive memories and 3D XPoint technology promise offering both byte-addressable I/O performance and persistent memory storage, which revolutionize the architecture design of file systems. As a result, various novel file systems based on NVM have been proposed, such as PMFS[2], SCMFS[13], NOVA[15] and HMVFS[18]. NOVA[15] and HMVFS[18] are designed by combining both faster, volatile DRAM and slightly slower, but the persistent non-volatile memory, and promise to offer efficient data access.

The directory-lookup is a vital part of accelerating data access in file systems with a large impact on application performance[7, 8, 11]. The design of directories balances the trade-off between directory write and read. There are two mainstream designs for directories: multi-level directory-namespace, and full name directory namespace. Multi-level directory-namespace sacrifices read efficiency because of recursive lookups but supports fast write operations such as RENAMEs and CPs. On the contrary, full name directory-namespace offers fast lookups without recursive scans but brings heavy write overhead if a part of full name changes. For traditional disk file systems with block I/Os, the small change of the directory name causes heavy write overhead because of write amplifications[6] and large random write. Therefore, most of disk file systems such as ext4 and ext3, prefer multi-level directory-namespace. Besides, betrFs provides full name directory-namespace by design WOD[3] based on LSM tree[9], which combines small random write into large sequential write. But betrFs shows slow file deletion and renames[17].

Non-volatile memory brings opportunities to solve heavy write overhead problem of full name directory-namespace. The non-volatile memories offer short write latency, high efficient random access, and byte-addressability. Write amplification does not exist in NVM. Therefore, many new-designed file systems come out, in order to utilize byte-addressable NVM. However, these NVM-based file systems still design their directories following traditional multi-level directory-namespace, which is mainly to avoid heavy write overhead due to block-I/Os and apparently not suitable for byte-addressable NVM. Although nvm solves the problem of write amplification in of full name directory-namespace, strictly full naming still causes a large amount of random write if directory name changes. Too much random write increases latency and has a bad impact on system performance.

To address these problems, we propose ADAM, an adaptive directory accelerating mechanism which offers fast read as well as small write overhead for NVM-based file systems. We introduce adaptive full directory namespace (AFDN) and self-adaptive namespace moving strategy in our ADAM. The key ideas of ADAM is that: 1) we allocate AFND areas to store both directory inodes and file inodes. Each AFND area has a root directory inode which stores strictly full pathname, and other directories in the area store path name compared with the AFND area root directory; 2) Read/write access frequency for each directory and the directory size according to the number of subfiles is recorded in ADAM; 3) we analyze different states of directories based on access frequency and size, and then implement a novel state_map to label them; 4) In our ADAM, root directories have a great impact on the system performance. Because when root directory name changes, it would not cause changes on the path names of its subdirectories. ADAM selects root directories according to the states marked in state_maps. The directory which is write hot, read hot, or has a large number of subfiles is most

WOODSTOCK'97, July 1997, El Paso, Texas USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

possibly to becomes an AFND root.
Since the states of directories would be changed during system runtime, we propose a strategy to keep file systems maintain the most efficient performance. We call it self-adaptive namespace moving strategy. It involves three movements among AFDN areas: split, merge and inherit. The basis of implementing the strategy is the states labeled in state_map. The self-adaptive namespace moving strategy ensures that the AFND root is selected for reasonableness and minimizes the write overhead of AFND during system runtime.
We build an efficient hybrid index which is suitable for hybrid DRAM/NVMM storage. The central idea of the Hybrid index is to make full use of large capacity NVM and access faster DRAM. Thus, we build a hash table for each directory and file of AFND areas in NVM and a radix tree in DRAM for all AFND roots. We implement our ADAM based on NOVA[15], a hybrid DRAM/NVMM file system, and evaluate the performance using micro- and macro-benchmarks.
The contributions of this paper are:
• We identify the old, unsuitable directory design in NVM based file systems. And propose AFND, ADAM, an adaptive directory accelerating mechanism for NVM based file systems, which offers fast lookup as well as low write overhead.
• We design a novel state_map to efficiently store different states of directories.
• We proposed a self-adaptive namespace moving strategy to keep the AFND file system maintaining the best performance during system runtime. Our strategy ensures the reasonable root selections and minimizes the write overhead in case of the directory states are changed.
• We implement an efficient hybrid DRAM/NVM directory index: RADIX tree in DRAM and multi-level hash tables in NVM. This hybrid index ensures the best utilization of the persistence and large-capacity NVM and faster DRAM.
• We implement ADAM in NOVA, which might be the fastest nvm-based file system. We call it NOVA-A. The evaluation results show that XXXX.
The remainder of this paper is organized as follows. Section II provides background and motivation. Section III presents the design and implementation of AFND and SANS. The implementation of NOVA-A is described in Section IV. We discuss the evaluation results in Section VI. Finally, we provide related works in section VII and conclude our paper in Section VIII.

2 BACKGROUND AND MOTIVATION

2.1 Non-Volatile Memory

Computer memory has been evolving rapidly in recent years. A new category of memory, NVM, has attracted more and more attention in both academia and industry[16]. NVM provides features like byte-addressability, non-volatility,and higher-to-flash speed. Table 1 shows the performance characteristics of different memory technologies. NVM provides slightly longer read latency compared with DRAM, while its write latency is apparently longer than DRAM. Similar to NAND Flash, the write endurance of NVM is limited, especially for PCM. Thus, many hybrid DRAM/NVMM file systems have been proposed to support lower latency, high write endurance, and persistence. Balancing writes and reads in hybrid DRAM/NVMM system is critical for software system design. Besides, NVM has high performance for random accessing like DRAM, which is different from traditional Flash.

Table 1: Comparison of different memory characteristics[4, 10, 16]

Category	Read latency	Write latency	Write Endurance	Random accessing
DRAM	60ns	60ns	10^{16}	High
PCM	50~70ns	150~1000ns	10^{9}	High
ReRAM	25ns	500ns	10^{12}	High
NAND Flash	35μs	350μs	10^{5}	Low

2.2 Directory Structures of File Systems

Traditional Block-I/O based file systems such as F2FS, ext4 use multi-level directory-namespace, in order to reduce block I/O write overhead compared with full name directory-namespace. However, this design sacrifices lookup speed for involving recursive scans of multi directory-namespaces. Strictly full directory-namespace brings a large amount of write overhead because of the write-amplification in block-I/Os, which does not exist in hybrid NVM-based file systems. Btrfs 0.2[17] introduces zones to optimize WODs in BetrFS. The main idea of BetrFS 0.2 is that: first divides directories and files into different zones according to the size of each directory and file; and then, use optimized LSM tree to reduce I/O overhead mainly brought by renames. BetrFS 0.2 reduces a part of potential rename overhead in the consideration of size. However, it is not the most important factor that affects read and write during system runtime. Access frequency directly affects the system performance. For example, read/write frequent directory is more qualified to be a new zone than seldom access but large directory.
Although NVM solves the write amplification by offering byte-addressable I/O. Newly NVM-based file systems such as PMFS and NOVA still follow the traditional directory design of disk-based file system, which is apparently not suitable for NVM storage. Directly implementing strictly full name directory in the NVM-based file systems is not appropriate for it causes a large number of random writes and has a bad impact on low write endurant NVM. Another challenge is that files and directories show different characteristics during runtime. These characters provide chances to balance read and write overhead in order to enhance the system performance. But there is no consideration of analyzing adaptive states in existing NVM-based file systems like NOVA and PMFS with the same standard for every directory and file.

3 DESIGN

Our design of ADAM in NVM-based file system aims to enhance directory read efficiency and minimize its write overhead. In this section, we first present the overview design of our adaptive directory accelerating mechanism. We then describe AFND, the layout of directory namespace. Finally, we describe our self-adaptive namespace moving strategy, which keeps file system maintaining high data access efficiency, even though the state of directories changes during file system lifetime.

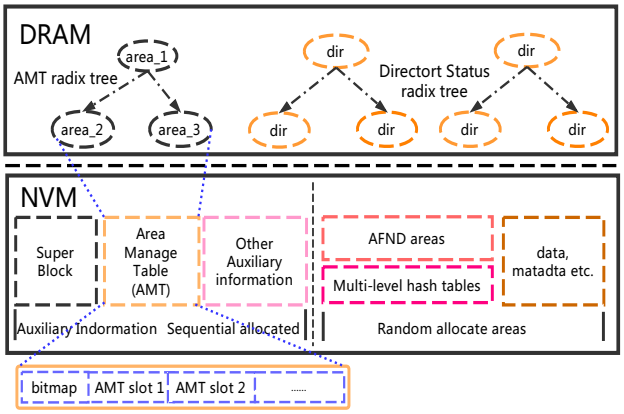

Figure 1: Architecture of ADAM-NOVA

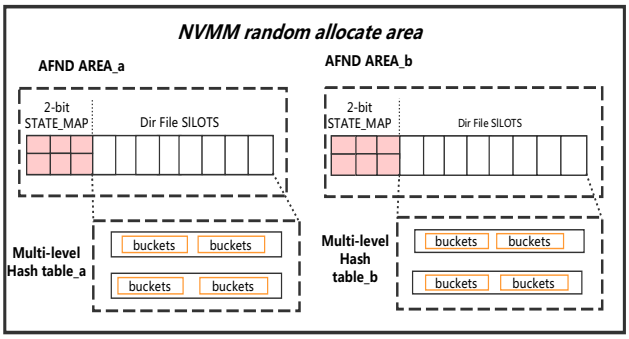

Figure 2: AFND areas and Hash tables

3.1 ADAM Overview

To accelerate directory access and suit to byte-addressable non-volatile memory, we proposed ADAM to replace the old directory mechanism in NVM-based file systems. We design ADAM based on three observations: First, existing new-designed NVM-based file systems follow the traditional directory mechanism, which is suitable for block-I/O based file systems, but not the byte-addressable NVM. Second, while nvm solves write-amplification problem of full name directory namespace, too many writes still increase system latency. Third, during runtime, directories show different characters in read-frequency, write-frequency, and size, which should be considered into file system optimization.

Based on these observations, we conceptually develop ADAM from two aspects: storage layout and efficient strategy. We introduce AFND area, an adaptive full namespace directory areas to manage the storage of directories. Each AFND area not only stores the entries of both directories and files but also records the state of directories with a novel design-state_map. Based on the state recorded in AFND, we develop a self-adaptive namespace moving strategy to manage movements of AFND areas. Our strategy ensures the consistent performance during system lifetime. Figure 1 shows the architecture of our directory layout.

3.2 AFND Area

AFND area contains two parts: directory and file metadata storage area and state_map area. We balance the directory area trade-off[17] and initialize each AFND area as a 512KB block array. Each directory and file entry is initially aligned on a 128-Byte boundary. The AFND area root directory records area_ID. Each AFND area_ID is marked by the hash value of its root directory strictly full name and indexed by hybrid index. In AFND storage part, the locality benefits are less important in NVMM-based storage, because both NVMM and DRAM support high qualified random access. ADAM append new directory entry to each zone in a round_robin order, so that directory entries are evenly distributed among AFND areas.

During file system runtime, the statement of directories might be changed. ADAM identifies different statements of directories based on read frequency, write frequency and directory size. We define three labels of different states for directories hot, warm and cold. Different access frequency and size correspond to different states which are described in table 1.

State_map is a two-bits bitmap, which records corresponding states of entry slots in AFDN area which is shown in table 1. Compared with other data-intensive data structures[], State_map not only saves storage space but also incorporates the role of a bitmap for marking valid/invalid states of slots. We make a trade-off with read frequency, write frequency and size to calculate directory state. Read frequency is an important statement component. If the whole AFND area is read hot, then it would save a lot of time for read operations and lookups.

Since write overhead is a prime factor of system performance, we consider write frequency as the most important factor in calculating statements of directories. In strictly full naming design, if one directory name changes, then all pathnames that contain this directory must be overwritten. However, in AFND, the pathnames of all files and directories stored in the same AFND area are the adaptive names compared with its root directory. If the name of a root directory changes, only few root directories which contain the name would be affected. The rest directories and files in AFND remain the same, which will minimize the write overhead.

Size is also one of the factors in calculating the statement of directories. Because larger directories are most likely to generate a large amount of write overhead. We show the rules of calculating directory statement in table 1. The real-time access frequency and the number of subfiles are recorded in dram radix tree in order to fully utilizing the short latency of DRAM. The state_map in NVM updates from time to time.

AFND defines the layout of both directory-inodes and file-inodes stored in NVM. In order to update status for each directory, each AFND area owns a novel two-bits bitmap called state_map ahead as figure 3 shows. State_map records entry valid information and

Table 2: Different States in 2-bit State_Map

Bit-Value	Label	Objects	State
00	Invalid	Directories	Invalid
		Files	Invalid
01	Cold	Directores	Small Size and Low Read Frequency
		Files	Valid
10	Warm	Directory	Small Size and Write Access
			Large Size and Low read frequency
11	Hot	Directory	Large Size and Write Access
			Large Size and High read frequency

status information of every slot, which is the base of self-adaptive namespace strategy.

We divide files and directories into three types: NORMAL_FILE, NORMAL_DIRECOTRY, and ROOT_DIRECTORY. In our mechanism, the AFND root directory has two strengths compared with normal directories: 1)it is fast to find the subfiles of root directories. 2) The name changes of root directories have the least effect on the write overhead of the entire system. Therefore, the division of AFND areas has a great influence on the system performance. ADAM introduce a self-adaptive namespace moving strategy to manage AFND areas selection. To enhance read performance and reduce write overhead, directories which are read hot, write hot and large should become the AFND area. The AFND area with a cold and small root directory should be merged back to AFNF areas. We also define the boundaries of directories and AFND areas(not decide whether to write this part or not).

The characters of directories change during system lifetime. We choose the number of subfiles and subdirectories, read-frequency and write-frequency as the most important characters. It is necessary to move AFND areas to maintain a good performance. Reasons are following: 1) new large directories with frequent access is suitable to be selected as root directories; 2) for small and seldom accessed AFND areas, invalid slots take up large NVM space but are seldom visited; 3) If a subdirectory is stronger than its parent root directory, which means the vast majority access to the AFND area are to visit this strong subdirectory. We call this kind of subdirectory, Strong Subdirectory.

Due to these situations, we define three movements among various AFND areas: split, merge and inherit, shown in figure 3. The system executes these movements on the base of states recorded in state_map.

Split has two state: positive split and negative split. Splitting is the way to create new AFND areas and select corresponding root directories. When split happens, all the subfiles and subdirectories are copied to the new AFND area. The Root directory is still staying in the original AFND area with new AFND area_ID. Positive splitting happens when the system periodically checks the state_map and finds that the directory with high access frequency and large size. AFND areas root directories chosen by positive splitting are either with high read frequencies and large size or with high write-frequencies and large size. Therefore, positive split both improves the efficiency of finding files or directories and reduces the overhead of writing. The MAX size of each AFND area is fixed and initialized to 512KB[17]. When a subdirectory becomes unusually large and there are not enough free slots in AFND area, we introduce negative split. In negative splitting, we tend to select large size directories as new root directories. By splitting large size directories, we not only free slots in the AFND area, but also reduced the amount of potential write overhead.

Merge happens when an AFND area becomes too small and the access frequency becomes too low. If the number of valid slots in this AFND area is less than 1/5 of the entire region, then we assume that it satisfies the merging size. We check the root directory from its state_map to determine that whether the AFND area satisfies the low access frequency request of merging. AFND areas that exhibit features including small size and low access frequency take up a significant amount of NVM space and they can neither accelerate the read access of directories and files nor contribute to the reduction of write overhead. Usually, the size of directories is reduced when deletion happens. Therefore, we check AFND area when large deletion operations happen and merge areas if it meets the necessary conditions of merging. During merging, we copy all the file-inodes and directory-inodes back to their parent AFND area. And then free this AFND area and make it reusable.

Inherit define an AFND movement that a Strong Subdirectory replaces its parent root directory. During our experimenting, we find there is a big challenge against splitting. We assume AFND area A with a root directory P, and it has a strong subdirectory S. In ADAM we check the state_map in A and then find S is the write-frequency and large directory that satisfies the splitting requirement.

Thus, we split S as a new AFND area B. After moving all subfiles and subdirectories belong to S, the parent root directory P becomes small directory and seldom visiting one, which meets the requirement of merging. We then merge AFND area A and free it. The whole steps are Check A-> Find S-> Allocate B-> Splitting S-> Merging A-> Free B. The overhead of managing NVM space is very high under these circumstances. To solve this problem, we propose inherit. When we find a strong subdirectory, we copy the rest of subfiles and subdirectories back to its parent AFND area and let this strong subdirectory replace the original root directory. The steps of inherit are Check A->Find S-> Small Copies. Inherit reduces the number of allocations/releases in NVM at a finer granularity and improves the system performance.

3.3 Self-Adaptive Namespace Moving Strategy

4 IMPLEMENTATION

We implement ADAM in NOVA, a log-structured file system based on Hybrid DRAM/NVMM. Note that ADAM can also be implemented in other NVM-based file systems such as PMFS[2] or HMVFS[18] to accelerate directory accessing. This section describes the implementation details of ADAM on NOVA.

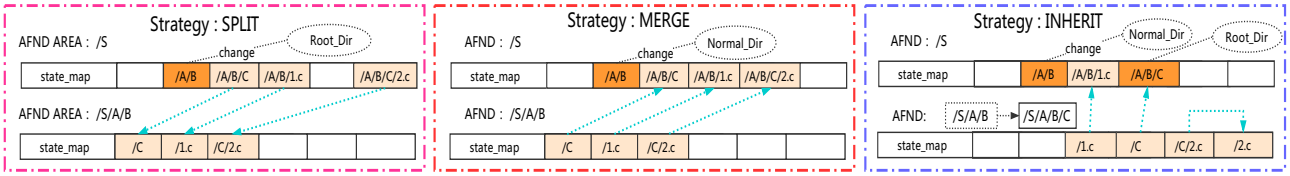

Figure 3: Self-Adaptive Namespace Strategy

4.1 Hybrid Index

We pay close attention to directory index for hybrid DRAM/NVM storage. In hybrid DRAM/NVM file systems, DRAM has lower latency but low capacity, whereas NVM has large capacity but slightly larger access latency. Therefore, the index in hybrid memories file system should utilize the strength of both lower latency DRAM and larger capacity NVM. However, it is not well balanced in NOVA. NOVA build a radix tree in DRAM for each directory inode[15], which takes too much space of DRAM. This index structure is obviously unrealistic in practical applications when there is a large number of directories. Encouraged by HiKV, we implement a more efficient hybrid directory index structure in NOVA, which is shown in figure 3.

AFND hash table We build a multi-level hash table for each AFND area. Multi-level hash tables are able to solve the hash conflicts, which is well-test in F2FS[5]. Unlike the uncertain level of hash tables in F2FS, five-level hash tables are enough for an AFND area in our experiment, because each AFND has a fixed number of slots for directory-inodes and file-inodes. Each hash entry contains a hash value of every directory inode and the position of AFND slot as a hash pair. To reduce the hash table size, each hash entry contains a valid bit and we reuse invalid hash entries for new files and directories. Hash tables greatly improve the speed of name searching of directories and files.

Area manager table(AMT). We sequentially allocated AMT in the auxiliary area of NOVA in NVM. AMT is initialized as a 4KB block and is divided into 54-Byte slots. We use an AMT bitmap to manage slots allocating and releasing. AMT manages the information of all the AFND areas and related hash tables. Each AMT slot corresponds to an AFND area and records its area_ID as well as hash table position. We mark each AMT slot by area_ID as the searching key. Therefore, users can easily find the AFND area and the hash table through AMT.

DRAM index. AMT is the most commonly accessed area for searching directory inodes, which is suitable for high read efficiency DRAM. Thus, we keep a radix tree in DRAM and the leaves of the radix tree store information as AMT slots stores. 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 AFND. The way to keep consistency between radix tree and AMT is same as Nova does. Besides, we also implement a status radix tree for each AFND area and is pointed by AMT radix tree leaves. The status radix tree records the access frequency state and size state for each accessed directory. We periodically convert the state stored in status radix tree to the value stored in state_map. We use a status tree in Dram to store the often changed directory states during system runtime. It would not take too much space of DRAM.

With our hybrid indexing, users can easily and quickly find the directory inode and file inode by the following step: 1) first calculate the AFND area_ID using adaptive full name; 2) find the correct AMT slot; 3)find the position of AFND area and corresponding hash table; 4)find the position of directory inode by the hash key-value pair. Compared with the original index design in Nova, our hybrid index greatly saves DRAM space, and also enhance the lookup efficiency through a hash table.

4.2 Scalable Name Storage

In our mechanism, the space overhead is a challenge for storing full name. Strictly full naming involves large duplicate path name storage which caused large space overhead. Our adaptive full naming has dramatically reduced the space overhead, compared with strictly full naming. However, the space challenge still remains in our mechanism.

First, adaptive full names of all files and directories contribute to a large space overhead. Second, each AFND slot has the boundary of 128 Bytes which is enough for common directories and files. But if the depth of directory is very deep or the path name is very long, an AFND slot is probably not enough to store the adaptive full name. In order to solve these challenges, we implement scalable name store. In our mechanism, the adaptive full name of directories is frequently accessed, such as calculating AFND area_ID when splitting, merging and inheriting movements happen. Thus, all directories store their adaptive full name in the slot. However, for normal_files it is not necessary to store their adaptive full name. Therefore, we store their own name and the parent directory position in the 128-byte slot. By reducing the file name length, we save a large NVM space and reduce the corresponding write overhead. When the name is too long to store under the boundary of 128 Bytes, we extend this slot by finding a free slot in this AFND area and make a pointer pointed to this extended slot in the original directory or file slot. If the extended is still not enough, then we extend another invalid slot and make a pointer, and so on.

By implementing this scalable naming method, we not only reduce the space overhead, but also satisfied the large name storage requirement.

4.3 Consistence And Garbage Collection

In our mechanism, only the small but frequently accessed data, such as AMT entries and state of directories, is stored in DRAM, in order to enhance the performance of file system. This part of data is updated in place at runtime with a journal log implemented in journal area of NOVA. The journal log records the movement and data in a 4KB buffer before movements start. NOVA prefer FAST GC with its original directory design, which is not conflicting with our mechanism.

5 EVALUATION

6 RELATED WORK

BetrFs 0.2 Since both multi-level directory-namespace and full name directory-namespace have strengths and shortages in balancing read and write overhead. Researchers have been conducted on how to utilize the fast-read of full name directory-namespace and low-overhead-write of multi-level directory namespace. BetrFS 0.2[17] builds zones for directories and files using write-optimized dictionaries[1] and improves the file system performance based on betrFS[3]. They propose zones inspired by Dynamic subtree partitioning[12] and implement it on BetrFS. The division of zones mainly refers to the size of files and directories. The files or directories chosen as roots of zones would potentially reduce the risk of large write overhead.

BetrFS 0.2 proposes a mechanism to maintain a consistent rename and scan performance trade-off in the consideration of possible size changes during system lifetime. The mechanism includes split and merge, and they define the Max size (ZoneMax) and Min size (ZoneMin) to decide when to split and when to merge. However, there are several problems remaining in BetrFS 0.2. 1) BetrFS 0.2 is designed for disk-based storage, which means the shortage of block I/Os remains in BestrFS. A small number of writes would cause large write overhead because of the write amplification problem. BetrFS 0.2 uses write-optimized dictionaries to combine the small random writes into large sequential writes which not suitable for NVM with high qualified random access. 2) The detection of zones is only based on the size of both directories and files, which is not accurate in zone_root selection. Size is only the potential reason that affects system read and write overhead. In the contrast, access frequency directly reflects read/write preference of the system. There are no considerations of access frequency in BetrFS 0.2. 3) The split and merge in BetrFS 0.2 might cause redundant NVM allocate, release and write. Therefore, in our mechanism, we introduced inherit to figure out this problem. 4) The space overhead from storing names is large. Our ADAM optimized this problem in implementing ADAM-NOVA.

HiKV. Hybrid DRAM/NVM based file systems provide a good architecture to fully use the strength of faster but volatile DRAM and non-volatile but slightly slower NVMM. It is reasonable to design an efficient index for hybrid DRAM/NVMM file system. In order to reduce the system latency, original nova overuse the capacity of DRAM via putting radix trees for each file and directory. It enhances system performance but takes up to much DRAM space. HiKV introduces a reasonable and efficient hybrid index for hybrid DRAM/NVM KV-store systems. HiKV[14] builds hash tables of KV pairs in NVM and build a B-tree in DRAM to quickly search for hash tables. We improve this hybrid index and implement it in the nvm-based file system. However, B-tree is mainly used to reduce I/Os in disk-based storage system. B-tree is mainly designed disk I/O, which obviously is not suitable for main memories. Our mechanism implements radix trees in dram which is more suitable for Linux kernel and uses multi-level hash tables to avoid hash conflicting. Compared with HiKV, our mechanism optimizes space management and remains high-qualified indexing.

7 CONCLUSION

This paper proposes ADAM, an adaptive directory accelerating mechanism designed for NVM-based file systems to enhance the directory read/write. ADAM includes AFND, an adaptive full namespace directory design and SANS, a self-adaptive namespace strategy. AFND introduces a novel state_map to record changing access-status and offers a reasonable management of both directory-inodes and file-inodes. SANS makes file system keep a efficient and persistent performance during runtime. We also implement a hybrid index with a good balance of DRAM and NVMM. Out measurements show XXXXXX.

REFERENCES

[1] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A Bender, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, et al. 2017. File Systems Fated for Senescence? Nonsense, Says Science!. In *FAST*. 45–58.

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

[3] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. 2015. BetrFS: Write-optimization in a kernel file system. *ACM Transactions on Storage (TOS)* 11, 4 (2015), 18.

[4] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 2–13.

[5] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage.. In *FAST*. 273–286.

[6] Eunji Lee, Julie Kim, Hyokyung Bahn, Sunjin Lee, and Sam H Noh. 2017. Reducing write amplification of flash storage through cooperative data management with NVM. *ACM Transactions on Storage (TOS)* 13, 2 (2017), 12.

[7] Paul Hermann Lensing, Toni Cortes, and André Brinkmann. 2013. Direct lookup and hash-based metadata placement for local file systems. In *Proceedings of the 6th International Systems and Storage Conference*. ACM, 5.

[8] Youyou Lu, Jiwu Shu, and Wei Wang. 2014. ReconFS: a reconstructable file system on flash storage.. In *FAST*, Vol. 14. 75–88.

[9] Patrick OâĂŹNeil, Edward Cheng, Dieter Gawlick, and Elizabeth OâĂŹNeil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[10] Kosuke Suzuki and Steven Swanson. 2011. The non-volatile memory technology database (nvmdb). *Micron* 296 (2011), 297.

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

[12] Sage A Weil, Kristal T Pollack, Scott A Brandt, and Ethan L Miller. 2004. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 4.

[13] Xiaojian Wu and AL Reddy. 2011. SCMFS: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 39.

[14] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. 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, July 12-14, 2017*. 349–362. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[15] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories.. In *FAST*. 323–338.

[16] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single

Level Systems.. In *FAST*, Vol. 15. 167–181.

[17] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A Bender, et al. 2016. Optimizing Every Operation in a Write-optimized File System.. In *FAST*. 1–14.

[18] Shengan Zheng, Linpeng Huang, Hao Liu, Linzhu Wu, and Jin Zha. 2016. Hmvfs: A hybrid memory versioning file system. In *Mass Storage Systems and Technologies (MSST), 2016 32nd Symposium on*. IEEE, 1–14.