J. Parallel Distrib. Comput. 120 (2018) 355–368

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

HMVFS: A Versioning File System on DRAM/NVM Hybrid Memory

Shengan Zheng, Hao Liu, Linpeng Huang *, Yanyan Shen, Yanmin Zhu

Shanghai Jiao Tong University, China

HIGHLIGHTS

- A Hybrid Memory Versioning File System on DRAM/NVM is proposed.
- A stratified file system tree maintains the consistency among snapshots.
- Snapshots are created automatically, transparently and space-efficiently.
- Byte-addressability is utilized to improve the performance of snapshotting.
- Snapshot efficiency of HMVFS outperforms existing file systems by 7.9x and 6.6x.

ARTICLE INFO

Article history:
Received 9 January 2017
Received in revised form 12 October 2017
Accepted 27 October 2017
Available online 16 November 2017

Keywords:
Versioning
Checkpoint
Snapshot
File system
Memory

ABSTRACT

The byte-addressable Non-Volatile Memory (NVM) offers fast, fine-grained access to persistent storage, and a large volume of recent researches are conducted on developing NVM-based in-memory file systems. However, existing approaches focus on low-overhead access to the memory and only guarantee the consistency between data and metadata. In this paper, we address the problem of maintaining consistency among continuous snapshots for NVM-based in-memory file systems. We propose an efficient versioning mechanism and implement it in Hybrid Memory Versioning File System (HMVFS), which achieves fault tolerance efficiently and has low impact on I/O performance. Our results show that HMVFS provides better performance on snapshotting and recovering compared with the traditional versioning file systems for many workloads. Specifically, HMVFS has lower snapshotting overhead than BTRFS and NILFS2, improving by a factor of 9.7 and 6.6, respectively. Furthermore, HMVFS imposes minor performance overhead compared with the state-of-the-art in-memory file systems like PMFS.

1. Introduction

Emerging Non-Volatile Memory (NVM) combines the features of persistency as disk and byte addressability as DRAM, and has latency close to DRAM. As computing moving towards exascale, storage usage and performance requirements are expanding dramatically, which increases the need for NVM-based in-memory file systems, such as PMFS [8], SCMFS [34], and BPFS [4]. These file systems leverage byte-addressability and non-volatility of NVM to gain maximum performance benefit [26].

While NVM-based in-memory file systems allow a large amount of data to be stored and processed in memory, the benefits of NVM are compromised by hardware and software errors. Bit flipping and pointer corruption are possible due to the byte-addressability of NVM media. Moreover, the applications that handle increasingly large-scale data usually require long execution time and are more likely to get corrupted by soft errors. To recover from these failures, we need to reboot the system and restart the failed applications from the beginning, which will have a severe consequence on the run time of the workloads. Hence, it is becoming crucial to provision the NVM-based in-memory file systems with the ability to handle failures efficiently.

Journaling, versioning and snapshotting are three well known fault tolerance mechanisms used in a large number of modern file systems. Journaling file systems use journal to record all the changes and commit these changes after rebooting. Versioning file systems allow users to create consistent on-line backups, roll back corruptions or inadvertent changes of files. Some file systems achieve versioning by keeping a few versions of the changes to single file and directory, others create snapshots to record all the files in the file system at a particular point in time [27]. Snapshotting file systems provide strong consistency guarantees to users for their ability to recover all the data of the file system to a consistent state. It has been shown in [17] that multi-version snapshotting can be used to recover from various error types in NVM systems. Moreover, file data is constantly changing and users need a way to create backups of consistent states of the file system for data recovery.

* Corresponding author.
E-mail address: lphuang@sjtu.edu.cn (L. Huang).

https://doi.org/10.1016/j.jpdc.2017.10.022
0743-7315/© 2017 Elsevier Inc. All rights reserved.

However, the implementation of file system snapshotting is a nontrivial task. Specifically, it has to reduce space and time overhead as much as possible while preserving consistency effectively. To minimize the overhead caused by snapshotting, we require a space-efficient implementation of snapshot that is able to manage block sharing well, because the blocks managed by two consecutive snapshots always have a large overlap. Therefore, snapshot implementations must be able to efficiently detect which blocks are shared [7]. When a shared block is updated from an old snapshot, the new snapshot can be created only if the old snapshot is guaranteed to be accessible and consistent. Moreover, if a block has been deleted, the file system must be able to determine quickly whether it can be freed or it is still in use by other versions. To ensure the consistency among all snapshots, we need to maintain additional metadata structures or journals in the file system.

The state-of-the-art and widely used approach of implementing consistent and space-efficient snapshot is Rodeh's hierarchical reference counting mechanism [22], which is based on the idea of Copy-on-Write (CoW) friendly B-tree. Rodeh's method modifies the traditional B-tree structure and makes it more CoW-friendly. This method was later adopted by BTRFS [24]. However, such design introduces unnecessary overhead to in-memory file systems due to the following two reasons.

Firstly, the key of taking fast and reliable snapshots is to reduce the amount of metadata to be flushed to persistent storage. CoW friendly B-tree updates B-tree with minimal changes to reference counts, which is proportional to fan-out of the tree multiplied by height. Nevertheless, the total size of the file system is growing rapidly, which leads to a larger fan-out and higher tree to write-back. GCTree [7] refers to this kind of I/O as an update storm, which is a universal problem among B-tree based versioning file systems.

Secondly, directory hierarchy is used as tree structure base in most B-tree based file systems, which leads to an unrestricted height. If we want to take a global snapshot, a long pointer chain will be generated for every CoW update from the leaf-level to the root. Some file systems [20,7] confine changes within the file itself or its parent directory to avoid the wandering tree problem, which increases the overhead of file I/O and limits the granularity of versioning from the whole file system to single file.

To address these problems, we propose the **Hybrid Memory Versioning File System (HMVFS)** based on a space-efficient in-memory **Stratified File System Tree (SFST)** to maintain consistency among continuous snapshots. In HMVFS, each snapshot of the entire file system is a branch from the root of file system tree, which employs a height-restricted CoW friendly B-tree to reuse the overlapped blocks among versions. Unlike traditional directory hierarchy tree with unrestricted height, HMVFS adopts node address tree as the core structure of the file system, which makes it space and time efficient for taking massive continuous snapshots. Furthermore, we exploit the byte-addressability of NVM and avoid the write amplification problem such that exact metadata updates in NVM can be committed at the granularity of bytes rather than blocks. We have implemented HMVFS by applying SFST to our ongoing project: HMFS, a non-versioning Hybrid Memory File System. We evaluate the effectiveness and efficiency of HMVFS. The results show that HMVFS has lower overhead than BTRFS and NILFS2 in snapshotting and recovering, improving by a factor of 9.7, 6.6 (snapshotting) and 1.8, 3.3 (recovering), respectively. The contributions of this work are summarized as follows.

• We have solved the consistency problem for NVM-based in-memory file systems using snapshotting. File system snapshots are created automatically and transparently. The lightweight design brings fast, space-efficient and reliable snapshotting into the file system.

• We design a stratified file system tree (SFST) to represent the snapshot of whole file system, in which tree metadata is decoupled from tree data. Log-structured updates to files balance the endurance of NVM. We utilize the byte-addressability of NVM to update tree metadata at the granularity of bytes, which provides performance improvement for both file I/O and snapshotting.

The remainder of this paper is organized as follows. Section 2 provides the background knowledge. Section 3 shows our overall design and Section 4 describes our implementation. We evaluate our work in Section 5, present related work in Section 6 and conclude the paper in Section 7.

2. Background

2.1. CoW friendly B-tree

Copy-on-Write (CoW) friendly B-tree is used by BTRFS [24] to allow BTRFS to better scale with the size of storage and create snapshots in a space-efficient way. The main idea is to use standard B+-tree construction but (1) employ a top-down update procedure, (2) remove leaf-chaining, and (3) use lazy reference-counting for space management.

Fig. 1 shows an example of how insertion and deletion change the reference counts of the *nodes* in B-tree. In this section, *node* refers to the node in the abstract B-tree, which is different from the node block mentioned in the rest of the paper. Each rectangle in the figure represents a block in B-tree and each block is attached with a pair (*block_id*, *reference_count*). Originally, the reference count records the use count of the block, i.e. if a block is used by n versions, its reference count is set to n. A block is *valid* if its reference count is larger than zero. This reference counting method can be directly applied to the file system to maintain the consistency among continuous snapshots. However, if the file system keeps reference counts for every block, massive reference counts will have to be updated when a new snapshot is created, which results in a large overhead. To reduce the update overhead, Rodeh came up with an improvement to the original CoW friendly B-tree, which transforms a traversing reference count update into a lazy and recursive one. Most updates in the reference counts of *nodes* are absorbed by their parent *nodes*. Such improvement reduces the time complexity of updates from $O(\#totalblocks)$ to $O(\#newblocks)$. This method is adopted by B-tree based file systems like BTRFS as their main data consistency algorithm.

Fig. 1(a) illustrates an original state of a simplified CoW friendly B-tree with reference counts. In Fig. 1(b), a leaf *node* D is modified (i.e. a new D' is written to storage). We perform path traversal from block D to the root of the tree (D→F→P), all the visited blocks are duplicated and written to the new locations. As a consequence, we have a new path D'→F'→Q. We set the reference count (`count`) of the newly created *nodes* (i.e. D', F', Q) to 1, and for the *nodes* that are directly referred by the *nodes* in the new path, we increase their `count` by 1, i.e. the reference counts of *node* C and E are updated to 2. Although the reference counts of *node* A and B have not changed, they are accessible by the newly created tree Q.

In Fig. 1(c), we consider the scenario that tree P is about to be deleted. We perform a tree traversal from *node* P and decrease the reference counts of the visited *nodes*. If `count` is larger than 1, the *node* is shared with other trees, thus we decrease `count` by 1 and stop downward traversal. If `count` is equal to 1, the *node* only belongs to tree P. We set the `count` to zero and continue the traversal. The *nodes* with zero count are considered *invalid*.

2.2. Hybrid Memory File System (HMFS)

HMFS is a log-structured non-versioning in-memory file system based on the hybrid memory of DRAM and NVM. HMFS utilizes log-structured writes to NVM to balance the endurance, and metadata is cached in DRAM for fast random access and update. HMFS also supports execute-in-place (XIP) in NVM to reduce the data copy

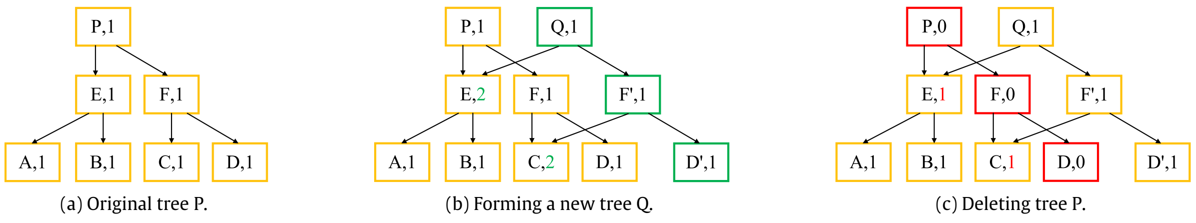


Fig. 1. Lazy reference counting for CoW friendly B-tree. (A,x) implies block A with reference count x.

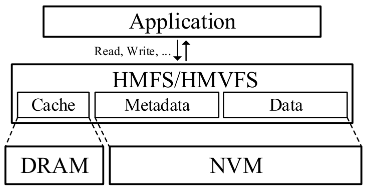


Fig. 2. Architecture of HMFS/HMVFS.

overhead of extra DRAM buffer cache. The architecture of HMFS is illustrated in Fig. 2.

As is shown in Fig. 3(a), HMFS splits the entire volume into two zones that support random and sequential writes, and are updated at the granularity of bytes and blocks respectively. Each zone is divided into fixed-size segments. Each segment consists of 512 blocks, and the size of each block is 4 kB. There are 5 sections in the random write zone of HMFS which keeps the auxiliary information of the blocks in the main area. The location of each section is fixed once HMFS is formatted on the volume.

Superblock (SB) contains the basic information of the file system, such as the total number of segments and blocks, which are given at the time of formatting and unchangeable.

Segment Information Table (SIT) contains the status information of every segment, such as the timestamp of the last modification to the segment, the number and bitmap of valid blocks. SIT is used for allocating segments, identifying valid/invalid blocks and selecting victim segments during garbage collection.

Block Information Table (BIT) keeps information of every block, such as the parent node-id and the offset.

Node Address Table is an address table for node blocks. All the node blocks in the main area can be located through the table entries.

Checkpoint keeps dynamic status information of the file system for recovery.

There are two types of blocks in the sequential write zone in HMFS. **Data blocks** contain raw data of files and **node blocks** include inodes or indirect node blocks, they are the main ingredients to form files. Fig. 4 shows the inner structure of each file. (Blue for node blocks and green for data blocks.) An inode block contains the metadata of a file, such as inode number, file size, access time and last modification time. It also contains a number of direct pointers to data blocks, two single-indirect pointers, two double-indirect pointers and one triple-indirect pointer. Each indirect node block contains 1024 node-ids (NID) of indirect or direct node blocks, each direct node block contains 512 pointers to data blocks. NID is translated to node block address with node address table. The cascade design of file structure can support file size up to 2 TB and is still efficient for small files (less than 4 kB) with inline data.

The idea of node and data blocks is inspired by F2FS [15]. In the design of traditional log-structured file system (LFS), any update to low-level data blocks will lead to a series of updates in direct node, indirect node and inode, resulting in serious write amplification. F2FS uses B-tree based file indexing with node blocks and node address table to eliminate update propagation (i.e. wandering tree problem [25]), only one node block will be updated if a data block has been modified, while traditional log-structured file systems updates two or three pointer blocks recursively [15]. If one needs to update multiple data blocks within the range of the same direct node in SFST (Fig. 5), only one node block needs to be updated. Multiple nodes will be updated only when the modified data blocks are managed by different direct nodes.

3. Hybrid memory versioning file system

HMVFS is a versioning file system based on HMFS. We implement a stratified file system tree (SFST) to convert the original HMFS into HMVFS. HMVFS allows the existence of multiple snapshot roots of SFST, as Fig. 5 shows, each root points to a version branch that represents a valid snapshot of the file system.

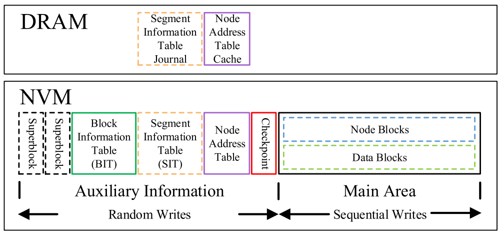
3.1. *NVM friendly CoW B-tree*

B-tree is widely used as the core structure of file systems, but trivial reference count updates hinder disk-based file systems to achieve efficient snapshotting by causing the entire block to be written again [7]. However, in memory-based file systems, reference counts can be updated precisely at variable bytes granularity, which motivates us to build an NVM-aware CoW friendly versioning B-Tree as a snapshotting solution to a B-tree based in-memory file system.

In SFST, we decouple these reference counts and other block-based information from B-tree blocks. As shown in Fig. 3(b), the auxiliary information is stored at a fixed location and updated with random writes to achieve efficiency, whereas the actual blocks of B-tree are stored in the main area and updated with sequential writes to support versioning. We adopt the idea of CoW friendly B-tree [24] to the whole file system on version basis, in which continuous file system snapshots are organized as SFST with shared branches, as shown in Fig. 5. With the idea of lazy reference counting [22], we achieve efficient snapshotting and also solve the block sharing problem which involves frequent updates to the reference counts of shared blocks.

3.2. *Stratified File System Tree (SFST)*

We implemented SFST (Fig. 5) in HMFS to convert it into a Hybrid Memory Versioning File System (HMVFS), the layout of HMVFS is shown in Fig. 3(b). SFST is a 7-level B-tree where the levels can be divided into four different categories: one-level checkpoint layer, four-level node address tree (NAT) layer, one-level node layer and one-level data layer. We convert the original checkpoint into a list of checkpoint blocks (CP) to keep status information of every snapshot the file system creates. We convert node address table into node address tree (NAT) which contains different versions of the original node address table. We also move NAT and CP to sequential write zone to better support block sharing in SFST. Each *node* in SFST is a 4 kB block located in the main area of HMVFS, and yields strictly sequential writes as Fig. 3(b) shows.



(a) Layout of HMFS.

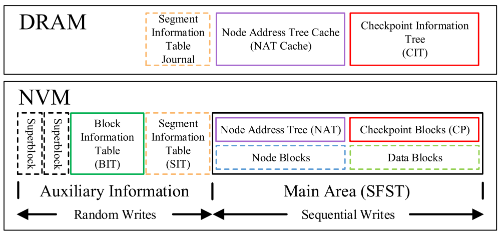

DRAM
Segment Information Table Journal
Node Address Tree Cache (NAT Cache)
Checkpoint Information Tree (CIT)
NVM
Superblock
Superblock
Block Information Table (BIT)
Segment Information Table (SIT)
Node Address Tree (NAT)
Checkpoint Blocks (CP)
Node Blocks
Data Blocks
Auxiliary Information
Main Area (SFST)
Random Writes
Sequential Writes
(b) Layout of HMVFS.

Fig. 3. Layout of HMFS and HMVFS, sections in the dash rectangles are shared between HMFS and HMVFS and other sections are specific to HMFS or HMVFS.

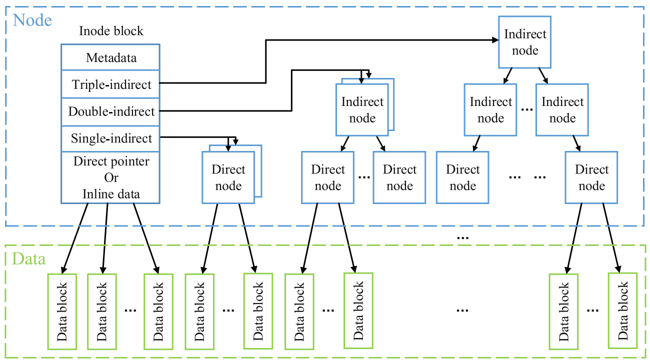


Fig. 4. File structure. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

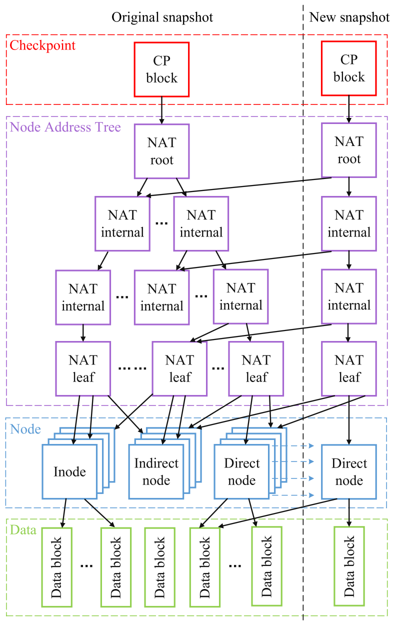


Fig. 5. Logical layout of SFST.

The reference counts of tree *nodes* are stored in BIT. The decoupled design of auxiliary information and blocks can be applied to other in-memory B-tree based file systems to achieve efficient snapshotting as well. We discuss the auxiliary information of SFST in Section 3.5.

In SFST, **Data Layer** contains raw data blocks. **Node Layer** is a collection of inodes, direct nodes and indirect nodes. Each node is attached with a 32-bit unique NID, which remains constant once the node is allocated. In order to support larger number of files, we can increase the NAT height to support more nodes in the file system. For each update to a node, a new node is written with the same NID as the old one, but in a different version branch of SFST.

Node Address Tree (NAT) is a CoW friendly B-tree which contains the addresses of all valid node blocks within every version, it is an expansion of node address table with added dimension of version. The logical structure of NAT in NVM is shown in Fig. 5. Node blocks are intermediate index structures in files (relative to data blocks), and node address tree keeps the right addresses of nodes with respect to different versions. NAT is further discussed in Section 3.3.

Checkpoint block (CP) is the root of every snapshot. It contains snapshot status, such as valid node count, timestamp of the snapshot. The most important part of CP is the pointer to the NAT root, which is exclusive to every snapshot. Snapshot information in CP is crucial to recovering the file system. The structure and management of checkpoint blocks are further discussed in Section 3.4.

In DRAM, HMVFS keeps metadata caches as residents, these caches provide faster access to metadata and are written back to NVM when taking snapshots. We use journaling for SIT entry updates and radix tree for NAT cache. The structure of Checkpoint Information Tree (CIT) is a red–black tree. The nodes of CIT are recently accessed checkpoint blocks of different snapshots, which are frequently used during garbage collection.

3.3. Node address tree

Node address tree (NAT) is the core of SFST. We design NAT as a multi-version node address table. To access the data of a file, one must follow the link of blocks like: inode⇒indirect node⇒direct node→data. However, in the file structure of HMVFS, the connections between nodes (⇒) are not maintained by pointers but NIDs, i.e. an indirect node contains only NIDs of direct nodes. In order to acquire the block address of the next node on the link, file system has to lookup the node address table of the current version. For file system versioning, to access the node blocks with the same NID from different versions, NAT must maintain different views of node address table for different versions. It is a challenging task to implement NAT such that NAT can be updated with low latency and be constructed space-efficiently. Fortunately, the byte-addressability of NVM allows us to update the variables of auxiliary information with fine-grained access, which inspired us to build a stratified versioning B-tree with block-based reference-counting mechanism without any I/O amplification.

We implement NAT with a four-level B-tree, as Fig. 5 shows. The leaf blocks of NAT consist of NID-address pairs (NAT entries) and other tree blocks contain pointers to lower level blocks. The B-tree design ensures low access overhead as well as space-efficiency for snapshotting.

In the tree of file structure, the parent *node* contains NIDs of the child *nodes* (unless the child nodes are data blocks, then it contains data block addresses), which remain unchanged in any later version during the existence of the file. For example, consider an inode that contains two direct nodes (NID = 2,3). NAT of the current version has two entries for NID and block address: {2,0xB}, {3,0xC}. In the next version, if a modification to this file requires to write a new node #3 (due to Copy-On-Write), NAT in DRAM should write a dirty entry {3,0xD} and keep other clean entries traceable. During snapshot creation, these dirty entries will be written back to NVM and form a new branch of B-tree (like the right part of Fig. 5). An important thing to notice is that throughout the entire data update procedure, the inode that contains NID 2 and 3 is never modified, it is still traceable by the original version of file system. The overhead of snapshotting is only caused by NAT updates which is much less than that of traditional schemes. CoW friendly B-tree [22] helps us to keep minimum modification size as well as the integrity of NAT among multiple versions.

The design of NAT not only reduces the overhead of block updates within files, but also excludes directory updates from file updates. In HMVFS, the directories contain inode numbers, which remains unchanged during file updates. In some file-grained versioning approaches (like NILFS [14], ZFS [2]), a directory containing different versions of a file has different pointers to each version of the file. Not only does it amplify the overhead of file updates, but it also increases unnecessary entries in the directory which leads to a longer file lookup time.

3.4. Checkpoint

As is shown in Fig. 5, checkpoint block is the head of a snapshot. In HMVFS, checkpoint block stores the status information of the file system and a pointer to the NAT root block. All the blocks of snapshots are located in the main area, and follow the rule of log-structured write in order to simplify the design and improve the wear-leveling of NVM. HMVFS separates data and node logging in the main area with data and node segments, where data segments contain all the data blocks and node segments contain all the other types of blocks (inodes, direct nodes, indirect nodes, checkpoint blocks and NAT blocks).

Checkpoint blocks are organized with doubly linked list in NVM. There is a pointer in superblock that always points to the newest checkpoint block for the purpose of system rebooting and crash recovery. In HMVFS, we have also implemented a Checkpoint Information Table (CIT) in DRAM that keeps checkpoint block cache for faster lookup.

3.5. Auxiliary information

Compared with HMFS, the random write zone in HMVFS contains the following three sections as the auxiliary information of SFST.

Superblock (SB) in HMVFS not only contains static information of the file system, but also maintains the list head of all snapshots and a pointer to the newest one as the default version for mounting. SB also contains a state indicator of the file system, it can be set to *checkpointing*, *garbage collecting*, etc.

Segment Information Table (SIT) in HMVFS is the same as the SIT in HMFS, but we revise the process of SIT update to ensure stronger consistency for versioning.

Block Information Table (BIT) is expanded from the BIT in HMFS to keep more information for snapshotting. It contains not only the node information of the block, but also maintains the start and end version number which indicates the valid duration of a block. BIT also contains reference counts of each block in SFST, and manages these blocks to form a CoW friendly B-tree with lazy reference counting.

The auxiliary information of these sections occupies only a small fraction of total space, which is about 0.4% since we use a 16B BIT entry to keep the information of each 4 kB block.

4. Implementation

In this section, we provide implementation details of HMVFS. We first illustrate how to manage snapshots with snapshot policy and operations. We then show the metadata operations of SFST and how to ensure overall consistency. Finally, we demonstrate how garbage collection handles block movements in multiple snapshots.

4.1. Snapshot management

We implement our snapshots with space and time efficiency in mind. Our snapshot is a height-restricted CoW friendly B-tree, with total height of 7. In the current prototype, we choose NAT height to be 4 to support up to 64PB of data in NVM. NAT height can be increased to adapt future explosive needs of big data (each additional level of NAT expands the support of the total size of NVM by 512x).

`Checkpoint` is a function that creates a valid snapshot of the current state. If a data block update is the only difference between two consecutive versions, SFST applies a bottom-up CoW update with 7 block writes, only 5 blocks of NAT and CP are actually written on `Checkpoint` because node and data blocks are updated in run time by XIP scheme. Moreover, these NAT nodes are updated only when new nodes are allocated, for the best case of sequential writes to a file, approximately 512 MB (An NAT leaf contains 256 NAT entries and a direct node block contains 512 pointers to data blocks, thus size $\approx 256 \times 512 \times 4$ kB $= 512$ MB) of newly written data requires one new NAT leaf update. Even for an average case, node updates are fewer compared with other B-tree based versioning file systems which suffer from write amplification problem. Therefore, `Checkpoint` in SFST is fast and space-efficient because it only contains few updates of NAT blocks.

4.1.1. Snapshot policy

We adopt a flexible snapshot policy for snapshot creation and deletion. Users can either explicitly invoke snapshot creation/deletion function in the foreground, or set **time interval** for periodical snapshot creation and **period of validity** for snapshot deletion in the background according to their usage patterns.

Specifically, snapshot creation is triggered on one of these occasions: (1) after an `fsync` system call from the applications; (2) after garbage collection; (3) after a certain **time interval** since last snapshot creation; (4) when explicitly invoked by users. In our design, background garbage collection is carried out every five minutes automatically, hence `Checkpoint` is conducted at most every five minutes. Shorter **time interval** can be specified by user-defined parameters if the users are eager to save their rapid changes to files. Moreover, `Checkpoint` only needs to build NAT and the checkpoint block of SFST in the main area, since node blocks and data blocks are updated directly using XIP. Thanks to our efficient design of snapshots, we manage to do frequent checkpoints with low impact on foreground performance.

Snapshot deletion is triggered by the user manually or by background cleaning in order to save NVM storage space. We provide interfaces for users to delete snapshots manually, or set up **preserve** flags and **period of validity** to delete old and unwanted snapshots while maintaining the important ones. In the foreground, users can explicitly delete a snapshot given its version number, or its timestamp. In the background, there are two ways to deal with the unwanted snapshots which are out of **period of validity**. One way is to directly delete them all, thus the file system will only contain recent snapshots. This can be used to save NVM storage space for intensive use of the file system. Another way is to select a some of the snapshots and skip them during background cleaning, these snapshots will survive the background deletion process and serve as a old reference for future recovery. This is designed for ordinary use of the file system where old snapshots are useful to retrieve old but important files. A gap number n can be set by the user to keep only one valid snapshot out of n continuous old snapshots.

We can keep up to 2^{32} different versions of the file system (due to the 32-bit version number). For most file systems, taking snapshots at a high frequency automatically reduces the I/O performance and occupies a large amount of memory space, while SFST creates space-efficient snapshots that imposes little overhead in snapshotting and I/O performance.

4.1.2. Snapshot creation

When `Checkpoint` is called, the state of HMVFS will be set to *checkpointing* and file I/O will be blocked until `Checkpoint` is done. The procedure of creating a snapshot is shown in List 1, the snapshot becomes valid after step 4.

List 1 Steps to Create a Snapshot

1. Flush dirty NAT entries from DRAM and form a new NAT in NVM in a bottom-up manner;
2. Copy SIT journal from DRAM to free segments in the main area in NVM;
3. Write a checkpoint block;
4. Modify the pointer in superblock to this checkpoint;
5. Update SIT entries;
6. Add this checkpoint block to doubly linked list in NVM and CIT in DRAM;

To create a snapshot, we build a new branch of SFST from the bottom up, like the right part of Fig. 5. With XIP in mind, updates in node layer and data layer have already been flushed to NVM at runtime. Therefore, we start from the NAT layer.

In step 1, since NAT caches are organized in radix tree, we can retrieve dirty blocks effectively. We write back the dirty blocks and follows the bottom-up procedure to construct NAT of this version. New child blocks create parent blocks that contain pointers to both these new blocks and adjacent blocks from old snapshots. The old blocks are shared with this new NAT and their reference counts are increased by one. We then recursively allocate new blocks till NAT root to construct the NAT of this version.

Segment Information Table (SIT) contains information of segments which are updated frequently. We use SIT journal in DRAM to absorb updates to SIT in NVM, and update SIT only during snapshot creation. In step 2, SIT journal in DRAM is copied to the free segment of main area without any BIT update. Step 4 is an atomic pointer update in superblock, which validates of the checkpoint. After that, the file system updates SIT according to the SIT journal on NVM. The SIT journal is useless after snapshot creation, but since it is never recorded on BIT or SIT, the file system will overwrite them as regular blocks. This ensures consistency in SIT, and in the meantime, avoid frequent updates to NVM.

When a checkpoint block is added to the superblock list in step 6, the snapshot becomes valid. HMVFS will always start from the last completed snapshot after a crash or reboot.

4.1.3. Snapshot deletion

On snapshot deletion, a recursive count decrement is issued to the target version. The address of checkpoint block of this version (the root of this branch of SFST) is found on the red–black tree on CIT in DRAM. The basic idea of snapshot deletion is illustrated in Fig. 1(c). We start the decrement of `count` from the checkpoint block through the whole branch of SFST recursively. Algorithm 1 describes the snapshot deletion operation in pseudo code.

When decrement is done, we remove the checkpoint block from the doubly linked list in NVM and the red–black tree in DRAM. After that, all the blocks of this version are deleted and the occupied

Algorithm 1 Delete snapshot

```
1:  function SnapshotDeletion(CheckpointBlock)
2:      Record CheckpointBlock in superblock;
3:      BlockDeletion(CheckpointBlock);
4:      Remove CheckpointBlock from CIT;
5:      Call GarbageCollection;
6:  end function
7:  function BlockDeletion(block)
8:      Record block in superblock;
9:      $count[block] - -$;
10:     if $count[block] > 0$ or $type[block] = data$ then
11:         return
12:     end if
13:     for each ptr to child node of block do
14:         BlockDeletion(ptr);
15:     end for
16: end function
```

space will be freed during garbage collection. Since snapshot deletion produces considerable free blocks, we call garbage collection function right after it. For the blocks which are referred in other valid versions, their reference counts are still greater than zero, and HMVFS considers these blocks as valid ones. The massive reads and deletions to reference counts are the bottleneck of snapshot deletion, but since NVM provides variable updates at the granularity of byte, the overhead of snapshot deletion is small and acceptable.

4.1.4. Recovery

HMVFS provides one writable snapshot for the newest version and a large number of read-only snapshots for old versions. To recover from an incorrect shutdown, HMVFS mounts the last completed snapshot. Since HMVFS keeps all dynamic system information in checkpoint block and SIT is not tainted (SIT is updated only during the last snapshot creation with consistency guarantee), the blocks which are written after the last completed snapshot are invalid and cannot be found in any SFST. After `fsck` cleans the invalid BIT entries, the file system is recovered and can overwrite the blocks from the incomplete version as regular blocks. We discuss how to handle file system crash during snapshot creation in Section 4.3.

To access the files in snapshots, HMVFS follows the link from superblock to checkpoint block to NAT and then to the node and data blocks of files, which introduces no additional latency in recovery. For mounting read-only old snapshots, we only need to locate the checkpoint block of the snapshot in the checkpoint list, which only introduces little extra time.

4.2. Metadata operations

In log-structured file systems (LFS), all kinds of random modifications to file and data are written in new blocks instead of directly modifying the original ones. In our implementation, the routines of file operations remain almost the same as that of a typical LFS. Only a small amount of additional reference data is written to block information table (BIT) on each block write. A BIT entry contains six attributes, and is 16 bytes in total, which is negligible in block operations where the block size is 4096 bytes.

BIT entries are used to solve block sharing problem incurred by multi-version data sharing that is caused by garbage collection and crash recovery. As Fig. 5 shows, there are *nodes* in SFST that have multiple *parent nodes*. When a NAT/node / data block is moved, we have to alert all its NAT parent blocks/NAT entries/parent node blocks to update the corresponding pointers or NAT entries to the new address in order to preserve consistency. We store the parent

Table 1
Types of blocks in the main area.

type of the block	type of parent	node_id
Checkpoint	N/A	N/A
NAT Internal NAT leaf	NAT internal	Index code in NAT
Inode Indirect Direct	NAT leaf	NID
Data	Inode or direct	NID of parent node

node information of all the blocks in BIT, and utilize the byte-addressability of NVM to update the information at the granularity of byte.

```
struct hmvfs_bit_entry {
__le32 start_version;
__le32 end_version;
__le32 node_id;
__le16 offset_and_type;
__le16 count;
}
```

For any block in SFST, BIT entries are updated along with the block operations without journaling. Meanwhile, the atomicity and consistency guarantees are still provided.

• start_version and end_version are the first and last versions in which the block is valid, i.e. the block is a valid node on each branch of SFST from start_version to end_version. write and delete operations to the block set these two variables into the current version number. Since SFST follows strict log-structured writes, these two variables are unchangeable.

• type is the type of the block. There are seven kinds of blocks in the main area, which is shown in Table 1. We store the types of these blocks in their BIT entries, because different parent nodes in SFST lead to different structures of storing the links to the child nodes. type is set at the same time when the block is written.

• node_id is the key to finding the parent node, it is set once the block is written. node_id has different meanings regarding different types, as Table 1 shows. For NAT nodes, each node contains 512 addresses, and the tree height is 4. We use $\log_2 512 \times 3 = 27$ bits in node_id to keep the index of all NAT nodes. For a node block of a file, node_id is the exact NID that NAT stores and allocates. For a data block of a file, node_id is the NID of its parent node. Given the above relation, the parent node of any block in SFST can be easily found.

• offset is the offset of the corresponding pointer to the block in its parent node. Combined with node_id, we can locate the address of the pointer to the block. With the byte-addressability of NVM, the pointer can be accessed with little cost. For a typical CoW update, a new parent block is written by copying the old one and modifying the pointer at offset to the latest address. After building all parent nodes recursively from the bottom up, and we will get a new version tree as the right part of Fig. 5.

• count is the reference count of the block, but differs from that of normal files, count basically records all the references by different versions. When a new block is allocated, its count is set to 1. If a new link is added from a parent block to it, its count increases by 1. If one of its parent block is no longer valid (only occur after snapshot deletion), its count decreases by 1. Once the count reaches zero, the block is freed and can be reused. We implement the idea of reference counting from Rodeh on HMVFS to maintain file system consistency [22,23]. write and delete are not the only two operations that modify count. Version level operations such as snapshot creation and deletion modify it as well. The rule of updating reference counts has been revealed in Section 2.1.

BIT contains not only the reference counts of all the blocks in the main area, but also the important metadata that reveals the relation of discrete blocks in SFST. The metadata operations above maintain the correctness of BIT and the consistency of SFST with minor cost in I/O performance.

4.3. Consistency guarantee

SFST guarantees the file system consistency after a power crash at any time. The metadata of SFST is consistent as long as NVM is attached to the memory bus and support at least 8-byte atomic write. In that case, we can access data in NVM directly via LOAD/STORE instructions of CPU(x86_64). Like other in-memory storage systems, SFST uses CPU primitives mfence and clflush to guarantee the durability of metadata.

When the above hardware conditions are met, the consistency of BIT and SIT is ensured all the time. Some parameters in BIT (start_version, node_id, offset and type) are set only once during block allocation by one thread, and the file system can undo these changes according to the version number. We can also call fsck to scan BIT and invalidate end_version for an uncommitted version of the snapshot. For count, the only operation which will decrease count and may cause inconsistency is snapshot deletion, which we will discuss later.

The updates to start_version, end_version, node_id, offset and type occur before the block allocation, their value will not be modified in run time. Atomic update to the whole entry is not necessary. We use atomic writes (provided by CPU) to update every single variable of BIT. Though the updates to these four members of BIT are not atomic, the system remains consistent. For example, when the system crashes in the process of updating node_id and the block has not been allocated to a specific file, we cannot index it from the NAT. The corrupted BIT data of this block has no effect on the consistency of the whole file system. We update count and end_version by atomic write at runtime.

```
struct hmvfs_sit_entry {
__le32 valid_blocks;
__le32 mtime;
}
```

For SIT, valid_blocks is the number of valid blocks that a segment contains. The block is valid if it can be indexed from any valid checkpoint. That is, there is at least one valid version number in [start_version, end_version) of its BIT entry. Mtime is the modified time of this segment. For example, if one block of the segment is invalidated or allocated, the mtime of the segment will be set to the system time in milliseconds. Therefore, SIT entry is used to describe usage information of a segment, which is the reason that we select victim segments in garbage collection function according to SIT. Because of the frequent modifications to SIT, we keep a cache of SIT in DRAM to absorb writes. We initialize these entries in DRAM during mounting, and flush the update log to free space on the main area in NVM when creating a checkpoint. After the checkpoint becomes valid (i.e., the pointer of superblock is modified to this checkpoint block), the log will be written to SIT, which can be redone if the system crashes during SIT updating. Such design ensures consistency in SIT, and in the meantime, avoids frequent updates to NVM.

File system consistency is also guaranteed during snapshot creation and deletion. For snapshot creation, when a crash happens before step 4 of List 1 in Section 4.1.2 , the file system will start from the last completed snapshot and discard any changes in this version. If a system crash happens during or after step 5, we redo `Checkpoint` from this step. During snapshot deletion, we keep a record of the snapshot version number and the progress of deletion, as is shown in Algorithm 1 . Since we follow a strict DFS procedure, only the information of currently being deleted block is needed to show where the deletion stops. If the file system is remounted from a crash occurred during snapshot deletion, we will redo the deletion before the file system becomes available to users.

4.4. Garbage collection

Garbage collection function is implemented in every log-structured file system to reallocate unused blocks and reclaim the storage space. The garbage collection process leverages the byte-addressability of NVM to perform efficient lookup of the most invalid block count in every SIT entry, and uses it to select the victim segment. After that, the garbage collection process moves all the valid blocks of the victim segment to the current writing segment, and sets the original segment free for further use. When the target segment is truly freed, garbage collection thread will perform an update to SIT journal accordingly. Since garbage collection does not require any updates in SFST, it has little impact on foreground operations like snapshot creation, block I/Os, etc.

The challenge of garbage collection for SFST is that when a block is moved, we must update the pointers in its parent node of every version to the new address. Otherwise, the consistency among multiple snapshots will be broken. Since we have inserted BIT modifications into normal file operations, we can use `node_id` and `offset` in BIT entry of the block to pinpoint the pointers from parent blocks to target block and modify them to the new address. To determine the related versions, we traverse from `start_version` to `end_version` of the existence of target block. The trick to performing quick inquiry of the parent nodes is that for each parent node, we skip all but one of the versions in the existence of parent node due to the fact that the parent node remains the same and valid during its own existence. All we have to do is find the successor parent node version by $\texttt{version}_{successor} \leftarrow \texttt{end\_version}_{this} + 1$ and continue until we reach the `end_version` of the target block. The byte-addressability and low access latency of NVM accelerates the update process substantially.

5. Evaluation

We evaluate the performance of HMVFS using various real workloads. We examine HMVFS against traditional file systems in memory, demonstrating that HMVFS achieves versioning at an acceptable performance cost compared with other in-memory file systems. Moreover, we compare the overhead of snapshot operations of HMVFS with popular versioning file systems to show the snapshotting efficiency of HMVFS.

5.1. Experimental setup

We conduct our experiments on a commodity server with 64 Intel Xeon 2 GHz processors and 512 GB DRAM, running the Linux 3.11 kernel. For EXT4, BTRFS, NILFS2, we use ramdisk carved from DRAM and configure 128 GB as ramdisk to simulate NVM. In the case of HMVFS, PMFS and NOVA, we reserve 128 GB of memory using the grub option memmap. User processes and buffer cache use the rest of the free DRAM space. The swap daemon is switched off in every experiment to avoid swapping off pages to disks.

It is important to note that our experiments focus on file system performance and the overhead of snapshotting operations, rather than evaluate different types of NVM. Since most of the performance results are relative, the same observations in DRAM should be valid in NVM. To compare the efficiency of snapshot creation and recovery, since the performance of time overhead is important and strongly related to space consumption, we use the comparison of time overhead as an effective and straightforward way to evaluate the efficiency of snapshotting among versioning file systems.

We use IOzone [13] to run sequential/random read/write operations over different transfer size to emulate different file I/O workloads. We also use Filebench suite [10] to emulate two common workloads in real life. The fileserver workload emulates file system activities of an NFS server, it contains an even mixture of metadata operations, appends, whole-file reads and writes. The varmail workload is characterized by a read–write ratio of 1:1, it consists of sequential reads as well as append operations to files.

We compare HMVFS with two popular versioning file systems, BTRFS [24] and NILFS2 [14], to show the efficiency of snapshot operations of HMVFS. We also compare HMVFS with three non-versioning but state-of-the-art in-memory file systems with the best I/O performance: PMFS [8], NOVA [35] and EXT4 [9]. PMFS is a typical in-memory file system, NOVA is a log-structured in-memory file system, and EXT4 is one of the most widely-used traditional file system with direct access patch *DAX* [6,28].

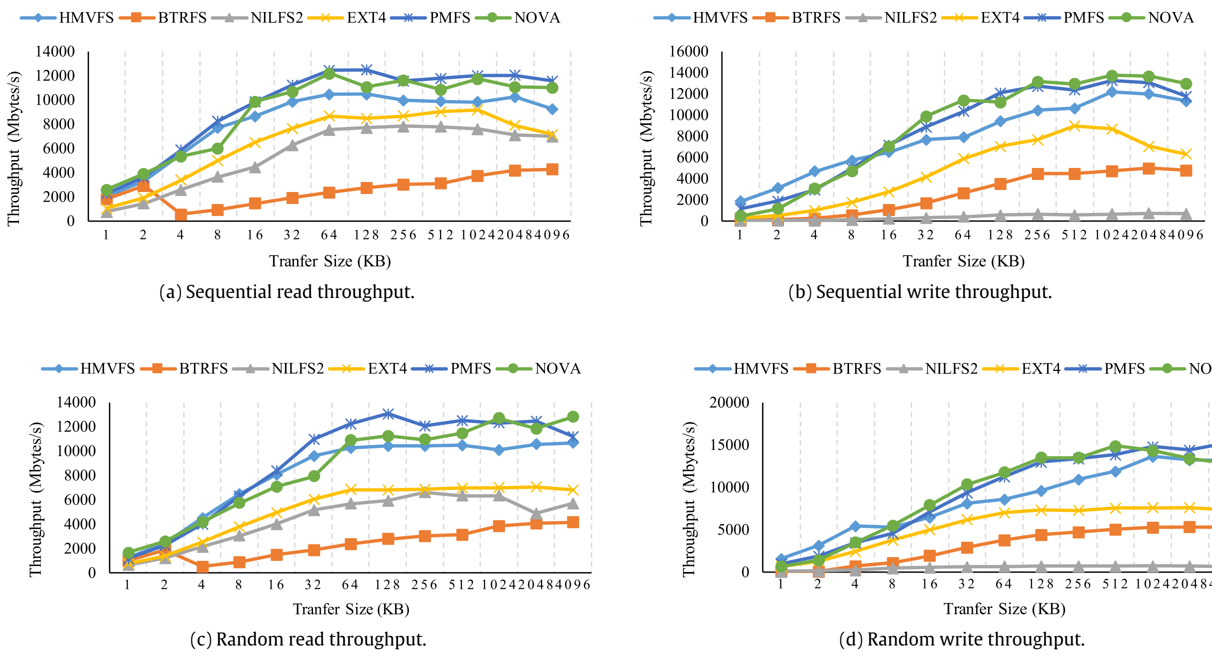
5.2. Overall performance

Before analyzing the efficiency of snapshot operations, we must show the file operations of HMVFS is efficient and suitable for in-memory computing. We run a series of benchmarks against HMVFS and evaluate its performance. The results show that HMVFS imposes acceptable read/write overhead to achieve snapshot functionality.

Iozone [13] is a file system benchmark tool which generates a variety of file operations. In this experiment, we configure the file size to 16 GB, and measure the average read/write throughput according to different transfer size (the granularity of file I/O). By default, we use direct I/O for all file operations to make sure that there is no page cache or buffer benefit in any file system. The transfer size is ranged from 1 kB to 4096 kB.

Fig. 6 shows the results of sequential/random read/write throughput respectively. As we can see, the read/write throughput of each file system mainly rises along with the increases of transfer size. Meanwhile, there is no large difference between the sequential and random result, since all the data is located in memory. Among all file systems, PMFS and NOVA perform the best in all four figures, while the read throughput of BTRFS and the write throughput of NILFS2 are the poorest. PMFS utilizes its simple block allocation and deallocation policies to achieve high performance, while NOVA uses per-inode logging to record all the data updates with little overhead. The throughput of HMVFS is the closest one to PMFS and NOVA among other file systems. HMVFS even surpasses PMFS's write throughput when transfer size is less than 16 kB. It is clear that HMVFS utilizes its multi-level node structure in each file to achieve particularly well performance in small-size I/O, since node cache in DRAM allows lower level data blocks to be found efficiently. PMFS also uses B-tree as its file structure, but it mainly focuses on uniform access control in its index structure to achieve metadata level consistency, while HMVFS utilizes B-tree to create snapshots so as to achieve version level consistency. NOVA achieves the best performance in sequential and random writes with large IO size, this is because it keeps per-inode radix tree in DRAM to index all its data blocks and perform log-structured writes to NVM data blocks.

Among three traditional file systems, EXT4 performs the best. There are some optimizations in EXT4 designed for direct access,



(a) Sequential read throughput.
(b) Sequential write throughput.
(c) Random read throughput.
(d) Random write throughput.

Fig. 6. Performance of file systems under iozone workloads.

like bypassing page cache layer, which makes it a popular choice for in-memory file system on RAMdisk. In Figs. 6(a) and 6(c), the read and write performance of EXT4 and NILFS2 are similar. Meanwhile, BTRFS suffers from its complex B-tree-based design. As is shown in Fig. 6(a), BTRFS still performs quite well before the IO size is bigger than 4 kB. This is because such a small size IO can fit into one extent, without further read in other blocks. However, when the IO size is greater than 4 kB, the file system has to locate other extents to complete the read request. This means BTRFS has to go through multiple layers of indirections to find the corresponding data block, which increases the read overhead significantly.

Since NILFS2 is a completely log-structured file system, each write is amplified to multiple updates to data and metadata blocks, which explains its low write performance in Figs. 6(b) and 6(d). Although BTRFS is also a log-structured file system, the writes in BTRFS are organized with extents. Extent-based file systems can handle write requests much better than a simple file block layout like NILFS2, hence BTRFS achieves a relatively higher write performance compared with NILFS2.

We notice that almost all file system throughputs reach the top values at around 1024 kB transfer size. This is because the L2 cache size in our experiment is 1024 kB. For transfer size larger than L2 cache size, L2 cache miss rate and data TLB misses increase significantly, which will affect the file I/O throughput.

5.3. Snapshot creation and deletion

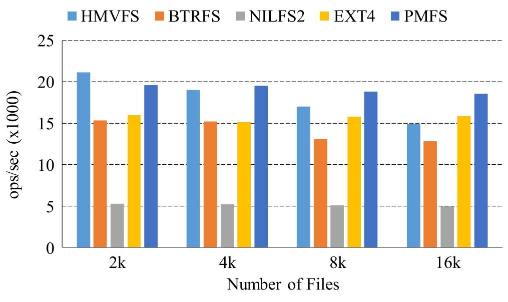
Despite I/O performance, HMVFS also ensures less snapshotting overhead compares with other approaches. For a snapshot contains only one 1 GB file, BTRFS and NILFS2 consumes 1.38 and 1.84 times the overhead of HMVFS. To measure the benefits of directory structure on snapshots, we take snapshots of a directory containing 1 to 64 files (1 GB each), and measure their overhead of creating snapshots. For BTRFS, the overhead is strongly related to the number of files, the structural design of BTRFS is similar to EXT4, which is based on B-tree. Such design makes it vulnerable when the number of files in one directory is not large enough to reflect the advantage of B-tree structure. In HMVFS, the directory is also B-tree based, but since HMVFS creates snapshot on `fsync` to deal with consistency problem instead of logging and performing transactions, less work is required and the overhead of HMVFS remains low under various number of files. NILFS2 organizes directory entries with array, which ensures NILFS2 to receive relatively well performance in searching and adding entries when there are not too many files in one directory. However, when the amount of entries grows, array based directories introduce much more overhead than B-tree based ones.

HMVFS deletes obsolete snapshots automatically, just as NILFS2 does. To evaluate the efficiency of snapshot deletion, we run fileserver, webserver and webproxy workloads for more than 15 min, take snapshots every 5 min, and delete the second snapshot. We record the time of creating and deleting the second snapshot on HMVFS, BTRFS, NILFS2, and compare the average overheads under the same workloads. We find out that although the overhead of snapshot deletion of BTRFS and NILFS2 are several times less than the overhead of their snapshot creation (BTRFS:2.36; NILFS2:1.57; HMVFS:1.03), HMVFS still achieves fastest snapshot deletion among all.

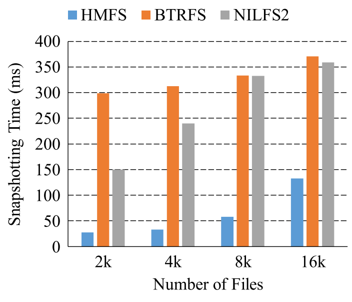
5.4. Impact of file count

Different numbers of files in file systems lead to different overhead of creating snapshots. We use fileserver workload from Filebench to emulate I/O activities of a simple file server containing different amounts of files. The file operations include creates, deletes, appends, reads, and writes. In this experiment, we configure mean file size to 128 kB, mean append size to 8 kB, directory width to 20 and run time to 5 min. We run fileserver three times and report the average readings.

Fig. 7(a) shows the results of varying the number of files from 2k to 16k. We see that on average HMVFS and PMFS perform the best amongst all file systems, while NILFS2 performs the worst. Although HMVFS has to keep records of reference count updates in DRAM and rebuild the file system tree on every `fsync` to ensure consistency, such overhead causes HMVFS drops the performance by only 8% compared with PMFS. Since fileserver mainly focuses on random writes on large files and updates on metadata, PMFS



(a) Operations per second.



(b) Efficiency of snapshot creation.

Fig. 7. Performance of file systems under fileserver workload.

utilizes the simple block allocation and deallocation policies, while HMVFS takes the advantage of NAT and performs quick block allocation.

The directories of HMVFS are also log-structured like normal files, we use hashed and multi-level index entry tree to store and lookup the contents, the complexity of which is $O(log(\#entries))$. We achieve quick access from such design when each directory contains moderate amount of entries. As the number of entries grows, the overhead of entry update increases, due to the log-structure nature of all inodes. Thus, we observe that the performance of HMVFS slightly degrades as the number of file increases.

EXT4 and BTRFS perform 13% and 22% worse than HMVFS. Despite the ramdisk environment, the short code path and fast random access design of EXT4 still ensure good I/O performance, which is also stable under different numbers of files. More importantly, in BTRFS and EXT4, data is organized with extents and the file system structure is also B-tree. It fits the nature of data extent that fileserver performs only write-whole-file and append. On file creation, a large data extent can be built. When fileserver appends data to existing files, BTRFS and EXT4 only need to create a new extent and merge it with the old one, which is very efficient. These extent-based file systems perform well under fileserver workload.

NILFS2 performs 73% worse than HMVFS. Since NILFS2 is a completely log-structured file system, any write to a file or directory will lead to recursive updates to multiple data and metadata blocks, resulting in a wandering tree problem. This problem gets worse when NILFS2 is mounted in memory, lots of redundant blocks of data and metadata are written for every append operation which takes extra time and space.

Fig. 7(b) shows the time overhead of creating snapshots after processing the workload above. On average, the overhead of snapshotting by BTRFS and NILFS2 is 9.7x and 6.6x worse compared with HMVFS. Without snapshotting, `fsync` operation of EXT4 and PMFS is to flush data and metadata back to storage in order to maintain consistency. We compare our snapshotting overhead with that of `fsync` operation after the same workload on these two file systems to further show the efficiency of HMVFS. EXT4 takes 14.3x time of HMVFS does and PMFS takes only 36%. On `fsync`, EXT4 has to commit all the changes as well as to keep checksum and journal up-to-date, which leads to a significant overhead. On the contrary, since PMFS supports XIP, most of the data changes have already been committed to storage, only a small part of work is left for `fsync` to do except waiting for current flush to complete.

BTRFS is capable of taking snapshots of its subvolume, based on the idea of CoW friendly B-tree. However, the structure of B-tree is applied to the file system directory layout. In order to take a snapshot that records a single change of data, BTRFS must rebuild all the directories along the path from the inode to the root, which introduces a significant overhead for snapshotting. As a result, BTRFS performs the worst when there is negligible change in data.

NILFS2 keeps all valid versions of files, it uses B-tree for scalable block mapping between the file offset and the virtual sector address. It also translates the virtual sector address to the physical sector address by using the data address translation (DAT) file, and appends the changes on every snapshot [18]. However, the DAT file is array-based, and NILFS2 suffers from inner wandering tree problem with its inode design.

To conclude, HMVFS and PMFS perform well under fileserver workload, meanwhile BTRFS and EXT4 exploit the advantage of extent-based file systems and achieve good performance. As for snapshotting efficiency, HMVFS outperforms BTRFS and NILFS2 by a large scale.

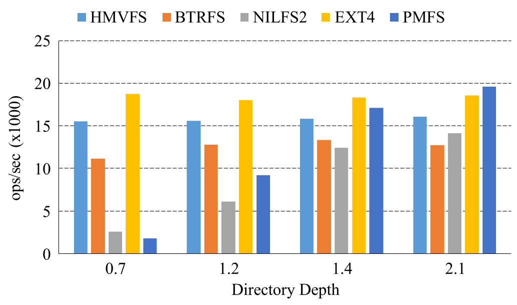
5.5. Effects of directory structure

Varmail emulates a mail server, which performs a set of create–append–sync, read–append–sync, read and delete operations. The read–write ratio is 1:1. In this experiment, we configure the mean file size to 16 kB, the number of total files to 100,000, mean append size to 8 kB and run time to 5 min. We run varmail on each different structures of directories three times and report the average readings.

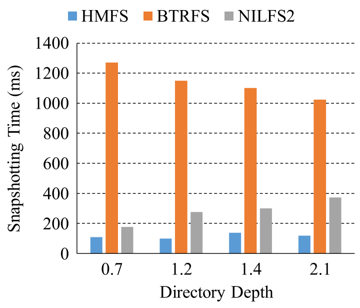
Fig. 8(a) shows the IOPS results of different mean-directory-depths of varmail ($depth = \log_{width}\#files$). We see that EXT4 performs the best among all file systems, while HMVFS and BTRFS achieves lower but also stable throughput. This is because all these three file systems use hashed B-tree to organize and locate directory entries [9,24]. However, NILFS2 and PMFS use only flat structure to store directory entries, and their performance degrades as the directory depth decreases.

The I/O performance of EXT4 outperforms HMVFS by a factor of 1.21 in varmail, and that of BTRFS is close to HMVFS. Extent based file systems perform reasonably well on append based workloads and varmail is a typical one that emulates new emails with appends. The difference between BTRFS and EXT4 is caused by the write amplification problem of appends and deletes on BTRFS write. Although HMVFS does not adopt the idea of extent, write amplification problem is eliminated by the NAT updates in SFST.

On the other hand, the IOPS results of NILFS2 and PMFS grow as the mean-directory-depth increases, i.e. the number of entries in single directory decreases. These two file systems use flat and array-based structure to organize directory entries, and that reduces the performance when each directory holds more than 100,000 entries. However, if the number of directory entries is decreased to around 4000 (depth=1.4), the performance of the two file systems increases sharply and nearly reaches their limits. PMFS even performs the most operations per second when each directory contains only 240 entries (depth = 2.1).



(a) Operations per second.



(b) Efficiency of snapshot creation.

Fig. 8. Performance of file systems under varmail workload.

During this experiment, we also notice that the preallocation time of each workload is inversely proportional to the final performance. To prove this point, we do file creations under the same workloads. Among five file creation results for each of dir-depth, HMVFS, BTRFS, EXT4 create files with steady overhead regardless of directory structure (standard deviation: 1.19, 0.78, 0.76). On the other hand, the overhead of file creation on NILFS and PMFS increases 9.8x and 16.6x when dir-depth is decreased from 2.1 to 0.7, which clearly shows that directory structure of many workloads affects the performance and access time of array-based directory file systems. Meanwhile, The B-tree based directory structure of HMVFS handles massive files well and provides stable throughput.

Fig. 8(b) shows the time overhead of creating a snapshot after processing the varmail workload above. On average, the overhead of snapshotting by BTRFS and NILFS2 is 8.7x and 2.5x worse compared with HMVFS. For BTRFS, taking snapshot leads to a CoW update in B-tree, the difference among the results of different dir-depths is that updates are distributed and logged in different directories, which can be improved by the concurrency of CPU.

In NILFS2, all the updates of files in snapshots are stored in their parent directories, with new entries pointing to the new addresses of corresponding inodes. Since the overhead of accessing each directory cannot be ignored, the cost of snapshotting on NILFS2 is proportional to the number of directories in this snapshot, which leads to a slight increase in snapshotting overhead along with the growth in dir-depth.

5.6. *Recovery performance*

In this section, we evaluate the recovery performance of three versioning file systems: HMVFS, BTRFS and NILFS2. For all the experiments, we perform five write sequences, each of which writes sixteen 1 GB files. At the end of each write sequence, a snapshot is created. We measure the time of recovering from the second, third and fourth snapshot to show the average recovery time from an arbitrary snapshot. For each experiment, firstly, we unmount the file system to emulate a sudden crash. Secondly, we mount the file system, and load the snapshot after that. The recovery time is shown in Fig. 9. HMVFS outperforms BTRFS and NILFS2 in crash recovery by a factor of 1.8 and 3.3.

For HMVFS, we mount the file system based on the given snapshot number. We measure the time of the mount operation, and the time for loading snapshot separately. In Fig. 9, the loading time of HMVFS is almost zero. During that time, the file system searches the given version number in checkpoint list, and then changes the checkpoint pointer in the superblock. The overhead of loading snapshot in HMVFS is significantly lower than that in BTRFS or NILFS2. Once the snapshot has been loaded, HMVFS resumes normal mounting procedure, where it utilizes the byte-addressability of NVM to mount more efficiently than BTRFS and NILFS2.

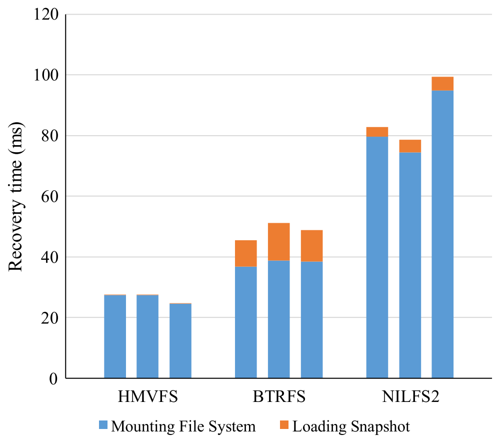


Fig. 9. Recovery performance.

We find that the mounting time of BTRFS is longer than HMVFS. Moreover, the loading procedure takes more time than the other two file systems because BTRFS loads its snapshot by user requests, and there is context switching overhead. Although BTRFS uses CoW-friendly B-tree like HMVFS, the granularity of its snapshot is not the whole file system but a subvolume. The subvolume tree can be snapshotted to another directory which serves as the same directory as the original one, except that it is only a snapshot. Therefore, when users call the recovery function to load the old snapshot, BTRFS simply removes the current subvolume and changes the name of the snapshot subvolume. When the renaming is complete, the snapshot can be accessed as the original subvolume.

NILFS2 uses one stratified tree to represent the whole file system. Therefore, like HMVFS, the principle of versioning is simply achieved by switching the head pointer of the *checkpoint* block to the *snapshot*. NILFS2 creates *checkpoints* automatically. Users can select significant versions among continuously created *checkpoints*, and can change them into *snapshots* which will be preserved until they are changed back to *checkpoints*. The overhead of loading a snapshot is low because searching for the *snapshot* is efficient in memory. However, NILFS2 is not optimized for byte-addressable access. Hence, the mounting time is longer than the other two file systems, which makes it the slowest file system for snapshot recovery.

6. Related work

6.1. In-memory file systems

To better cope with emerging non-volatile memory, in-memory file systems such as PMFS [8], SCMFS [34], BPFS [4], Aerie [32], Shortcut-JFS [16] and NOVA [35] leverage the byte-addressability and random access features of NVM to gain maximum performance benefit [26]. Among all these file systems, PMFS is the only open source NVM-aware file system available, hence we only include PMFS in our evaluation. Researchers have also proposed many hybrid volatile/non-volatile main memory systems in [1,21,38]. The consistency issues for in-memory systems are addressed in [11,31,3]. However, existing in-memory file systems focus on providing direct access functionality and high performance, which ensures only data and metadata consistency. Meanwhile, HMVFS focuses on implementing fast, space-efficient and reliable snapshot function. A preliminary version of HMVFS was published in [37], and the consistency mechanism of HMVFS is described in [36].

PMFS [8] is a state-of-the-art in-memory file system optimized for byte-addressable non-volatile memory. It provides direct access to NVM, which bypasses the block layer and file system page cache to improve performance. Like EXT4, it uses B-tree to organize the data pages of a file. For consistency, PMFS uses journaling for metadata updates, along with `clflush` and `mfence`, to ensure metadata consistency.

BPFS [4] uses a new technique called short-circuit shadow paging to provide atomic, fine-grained updates to persistent storage to ensure consistency. It also requires architectural enhancements to enforce write ordering. A limitation of BPFS is that atomic operations which span a large portion of the tree can require a significant amount of extra copies which incurs large overheads.

NOVA [35] is a log-structured file system for DRAM/NVM hybrid memory. Compared with HMVFS, NOVA maintains separate logs for each inode to improve concurrency, but do not keep logs for file data. NOVA keeps logs in NVM and indices in DRAM. It uses logging and lightweight journaling for complex atomic updates. NOVA provides metadata, data consistency, rather than version consistency.

Consistent and Durable Data Structure (CDDS) is proposed in [31] to store data efficiently. Similar to PMFS, CDDS expects NVM to be exposed across a memory bus. To reduce the overhead of system call, CDDS maps data into the address space of process and use userspace libraries to handle data access. Furthermore, all updates in CDDS are executed in place to reduce the cost of moving data. In physical layer, CDDS uses the primitives sequence {`mfence`, `clflush`, `mfence`} to provide atomicity and durability for object store. In data structure, CDDS guarantees consistency by versioning, Copy-On-Write, and B-Tree. However, CDDS is in a single-level storage hierarchy, which ignores the difference between NVM and DRAM, such as the speed of writing and device's lifetime. Frequent updates to the same location on NVM will decrease system performance. Moreover, objects are flushed from cache of CPU to NVM once they have been updated, which will reduce memory locality and pollute the CPU cache.

6.2. Versioning file systems

Journaling and snapshotting are two main mechanisms which ensure file system consistency from rebooting after sudden power failure. Journal is a running transaction log that keeps track of all modifications to the file system since the last consistent state, and snapshots contain consistent metadata and data backups of the file system.

Many file systems only use journaling to record all the updates and recover with journal, which leaves the file systems in only one consistent state. Ext3 [30], LinLogFS [5], UBIFS [12], XFS [29], NTFS [19] and many other traditional on-disk file systems use this simple implementation to support metadata consistency. Compared with versioning consistency, journaling can only recover the data over a short period of time. Most journaling file systems delete the journals when they are committed. This is because the cost of replaying and storing journals is proportional to the number of journals, which increases rapidly along with the amount of operations. Versioning consistency can achieve versioning in a long period of time with little recovery and space overhead. Moreover, journaling introduces additional double copy overhead to the metadata and data of the file system, which is unnecessary in versioning file systems like HMVFS.

Among log-structured file systems, single snapshot is often an efficient way to store a consistent state of whole file system. YAFFS2 [33], F2FS [15] and Sprite LFS [25] store a valid snapshot back to persistent storage once in a while. Log-structured file systems also suit data salvage and snapshots because past data is kept in the storage, some modern implementations of LFS offer multiple versions of file system states by taking advantage of this feature.

In ZFS [2], writeable snapshots (clones) can be created, resulting in two independent file systems that share a set of blocks. For any file updates to either of the cloned file systems, new data blocks are created to reflect those changes, but the unchanged block continues to be shared. ZFS can take single snapshot of a file or recursive snapshots of a directory, providing a consistent moment-in-time snapshot of the file system. However, during snapshotting, a new inode is allocated for each inode already in the file system, which leads to a significant overhead of creating a snapshot of a large amount of files.

BTRFS [24] is constructed from a forest of CoW friendly B-trees, and snapshots are taken to the subvolume with a new tree sharing everything but the different parts. In BTRFS, file data, metadata and snapshots are organized in extents instead of blocks. Extent-based file systems perform better sequential I/O than block-based ones, but in byte-addressable storage like NVM, extent-based file structure gains little benefit at the cost of complexity. Moreover, each extent contains a back-reference to the tree node or the file that contains the extent, which causes more overhead of snapshot operations.

Gcext4 [7] is a modified version of EXT4 based on GCTree, a novel method of space management that uses the concept of block lineage across snapshots, as the basis of snapshots and Copy-on-Write. GCTree inserts several pointers to the inodes and index blocks of EXT4 to form a list of descendants of an inode or block. GCTree also implements a special (*ifile*) to add a layer of indirection between directory entries and inodes, which shares similar purpose to NAT in our design. However, *ifile* itself in Gcext4 is array-based rather than tree-based, which makes it difficult to deal with substantial inode updates efficiently.

NILFS [14] creates a number of snapshots every few seconds or per synchronous write basis. Users can select significant versions among continuously created snapshots, and change them into specific snapshots which are preserved all the time. There is no limit on the number of snapshots and each snapshot is mountable as a read-only file system. However, different versions of a file are stored in different snapshots, which increases the overhead of data seeking and segment cleaning. Also, NILFS suffers from wandering tree problem with its file structure, which results in a large overhead of file access.

Although the above file systems provide snapshotting functionality, they are built around the traditional storage layout with block I/O interface. They do not support direct access (DAX) to the file data on NVM. Without DAX, not only the performance will drop since they only perform block-based I/O, but the consistency problem will also arise. Traditional block-based file systems rely on block driver to ensure consistency, while NVM has no such driver. Hence, the file systems have to ensure the consistency of data and metadata writes to NVM. . Traditional file systems typically neglect this issue. Therefore, they can hardly be extended to support versioning in NVMs.

7. Conclusion

As the need of NVM-based file system increases, snapshotting has become a crucial component of fault tolerance. To utilize the byte-addressability of memory and lower the overhead of snapshotting, we present HMVFS, a new versioning file system on DRAM/NVM hybrid memory. We use the stratified file system tree (SFST) as the core structure of the file system such that different versions of files can be easily updated and snapshotted with minimal updates to metadata. HMVFS exploits the structural benefit of CoW friendly B-tree and the byte-addressability of NVM to automatically take frequent snapshots with little cost in time and space. While other studies focus on fast and reliable access to the file systems in NVM, we develop a file system with snapshot functionality and only takes little performance cost compared with other in-memory file systems. The snapshotting overhead of HMVFS outperforms BTRFS up to 9.7x and NILFS2 up to 6.6x, and the I/O performance of HMVFS is close to PMFS. To the best of our knowledge, this is the first work that solves the consistency problem for NVM-based in-memory file systems using snapshotting, and we expect it to become a powerful choice of in-memory versioning solutions.

Acknowledgments

We thank the anonymous reviewers for their insightful and helpful comments, which improve the paper. We also thank Linzhu Wu for his assistance. The work described in this paper is supported by the National High-tech R&D Program of China (863 Program) under Grant No. 2015AA015303 and the National Natural Science Foundation of China under Grant No. 61472241. Yanyan Shen is partially supported by NSFC (No. 61602297).

References

[1] J. Arulraj, A. Pavlo, S.R. Dulloor, Let's talk about storage & recovery methods for non-volatile memory database systems, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, 2015, pp. 707–722.
[2] J. Bonwick, B. Moore, ZFS: The last word in file systems, 2007.
[3] V. Chidambaram, T. Sharma, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Consistency without ordering, in: FAST, 2012, p. 9.
[4] J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, D. Coetzee, Better I/O through byte-addressable, persistent memory, in: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, ACM, 2009, pp. 133–146.
[5] C. Czezatke, M.A. Ertl, LinLogFS-A log-structured file system for linux, in: USENIX Annual Technical Conference, FREENIX Track, 2000, pp. 77–88.
[6] DAX: Page cache bypass for filesystems on memory storage, 2014. URL https://lwn.net/Articles/618064/.
[7] C. Dragga, D.J. Santry, Gctrees: Garbage collecting snapshots, ACM Trans. Storage (TOS) 12 (1) (2016) 4. http://dx.doi.org/10.1145/2857056.
[8] S.R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, J. Jackson, System software for persistent memory, in: Proceedings of the Ninth European Conference on Computer Systems, ACM, 2014, p. 15.
[9] Ext4, https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.
[10] Filebench, http://filebench.sourceforge.net.
[11] J. Huang, K. Schwan, M.K. Qureshi, NVRAM-aware logging in transaction systems, Proc. VLDB Endow. 8 (4) (2014) 389–400.
[12] A. Hunter, A Brief Introduction to the Design of UBIFS, Rapport Technique, 2008.
[13] Iozone Filesystem Benchmark, http://www.iozone.org/.
[14] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, S. Moriai, The Linux implementation of a log-structured file system, Oper. Syst. Rev. 40 (3) (2006) 102–107.
[15] C. Lee, D. Sim, J. Hwang, S. Cho, F2FS: A new file system for flash storage, in: 13th USENIX Conference on File and Storage Technologies, FAST 15, 2015, pp. 273–286.
[16] E. Lee, S. Yoo, J.-E. Jang, H. Bahn, Shortcut-JFS: A write efficient journaling file system for phase change memory, in: 012 IEEE 28th Symposium on Mass Storage Systems and Technologies, (MSST), IEEE, 2012, pp. 1–6.
[17] G. Lu, Z. Zheng, A.A. Chien, When is multi-version checkpointing needed?in: Proceedings of the 3rd Workshop on Fault-Tolerance for HPC At Extreme Scale, ACM, 2013, pp. 49–56.
[18] C. Min, S.-W. Lee, Y.I. Eom, Design and implementation of a log-structured file system for flash-based solid state drives, IEEE Trans. Comput. 63 (9) (2014) 2215–2227. http://dx.doi.org/10.1109/TC.2013.97.
[19] R. Nagar, Windows NT File System Internals: A Developer's Guide, O'Reilly & Associates, Inc., 1997.
[20] Z. Peterson, R. Burns, Ext3cow: a time-shifting file system for regulatory compliance, ACM Trans. Storage (TOS) 1 (2) (2005) 190–212. http://dx.doi.org/10.1145/1063786.1063789.
[21] M.K. Qureshi, V. Srinivasan, J.A. Rivers, Scalable high performance main memory system using phase-change memory technology, ACM SIGARCH Comput. Archit. News 37 (3) (2009) 24–33.
[22] O. Rodeh, B-trees, shadowing, and clones, ACM Trans. Storage (TOS) 3 (4) (2008) 2. http://dx.doi.org/10.1145/1326542.1326544.
[23] O. Rodeh, Deferred reference counters for Copy-On-Write B-trees, Tech. rep., Technical Report rj10464, IBM, 2010.
[24] O. Rodeh, J. Bacik, C. Mason, BTRFS: The Linux B-tree filesystem, ACM Trans. Storage (TOS) 9 (3) (2013) 9. http://dx.doi.org/10.1145/2501620.2501623.
[25] M. Rosenblum, J.K. Ousterhout, The design and implementation of a log-structured file system, ACM Trans. Comput. Syst. (TOCS) 10 (1) (1992) 26–52.
[26] P. Sehgal, S. Basu, K. Srinivasan, K. Voruganti, An empirical study of file systems on nvm, in: 2015 31st Symposium on Mass Storage Systems and Technologies, (MSST), IEEE, 2015, pp. 1–14.
[27] D.-I.J. Stender, Snapshots in Large-Scale Distributed File Systems (Ph.D. thesis), Humboldt-Universität zu Berlin, 2013.
[28] Supporting filesystems in persistent memory, 2014. https://lwn.net/Articles/610174/.
[29] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, G. Peck, (1996) Scalability in the XFS file system, in: USENIX Annual Technical Conference, Vol. 15, 1996.
[30] S.C. Tweedie, Journaling the Linux ext2fs filesystem, in: The Fourth Annual Linux Expo, 1998.
[31] S. Venkataraman, N. Tolia, P. Ranganathan, R.H. Campbell, Consistent and durable data structures for non-volatile byte-addressable memory, in: FAST, Vol. 11, 2011, pp. 61–75.
[32] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, M.M. Swift, Aerie: Flexible file-system interfaces to storage-class memory, in: Proceedings of the Ninth European Conference on Computer Systems, ACM, 2014, p. 14.
[33] C.-H. Wu, T.-W. Kuo, L.-P. Chang, Efficient initialization and crash recovery for log-based file systems over flash memory, in: Proceedings of the 2006 ACM Symposium on Applied Computing, ACM, 2006, pp. 896–900.
[34] X. Wu, A. Reddy, SCMFS: a file system for storage class memory, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, p. 39.
[35] J. Xu, S. Swanson, NOVA: a log-structured file system for hybrid volatile/non-volatile main memories, in: 14th USENIX Conference on File and Storage Technologies, FAST 16, 2016, pp. 323–338.
[36] J. Zha, L. Huang, L. Wu, S.-a. Zheng, H. Liu, A consistency mechanism for NVM-Based in-memory file systems, in: Proceedings of the ACM International Conference on Computing Frontiers, ACM, 2016, pp. 197–204.
[37] S. Zheng, L. Huang, H. Liu, L. Wu, J. Zha, Hmvfs: A hybrid memory versioning file system, in: Mass Storage Systems and Technologies, (MSST), IEEE, 2016, pp. 1–14.
[38] P. Zhou, B. Zhao, J. Yang, Y. Zhang, A durable and energy efficient main memory using phase change memory technology, in: ACM SIGARCH Computer Architecture News, Vol. 37, ACM, 2009, pp. 14–23.

Shengan Zheng is currently a Ph.D. student at Shanghai Jiao Tong University, China. He received his bachelor degree from Shanghai Jiao Tong University, China, in 2014. His research interests include in-memory computing and file systems.

Hao Liu is currently a Ph.D. student at Shanghai Jiao Tong University, China. He received his master degree from University of Science and Technology of China in 2009. His research interests include in-memory computing storage systems, mainly focus on file system and key value store system.

Linpeng Huang received his M.S. and Ph.D. degrees in computer science from Shanghai Jiao Tong University in 1989 and 1992, respectively. He is a professor of computer science in the department of computer science and engineering, Shanghai Jiao Tong University. His research interests lie in the area of distributed systems and service oriented computing.

Yanyan Shen is currently an assistant professor at Shanghai Jiao Tong University, China. She received her B.Sc. degree from Peking University, China, in 2010 and her Ph.D. degree in computer science from National University of Singapore in 2015. Her research interests include distributed systems, efficient data processing techniques and data integration.

Yanmin Zhu is a professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University, Shanghai, China. He obtained his PhD in 2007 from Hong Kong University of Science and Technology, Hong Kong. His research interests include crowd sensing, and big data analytics and systems. He is a member of the IEEE and IEEE Communication Society. He has published more than 100 technical papers in major journals and conferences.