HMVFS: A Hybrid Memory Versioning File System

Shengan Zheng, Linpeng Huang, Hao Liu, Linzhu Wu, Jin Zha
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Email: {venero1209, lphuang, liuhaosjtu, wulinzhu, qweeah}@sjtu.edu.cn

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 a Hybrid Memory Versioning File System (HMVFS) that achieves fault tolerance efficiently and has low impact on I/O performance. Our results show that HMVFS provides better performance on snapshotting 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 [1], SCMFS [2], and BPFS [3]. These file systems leverage byte-addressability and non-volatility of NVM to gain maximum performance benefit [4].

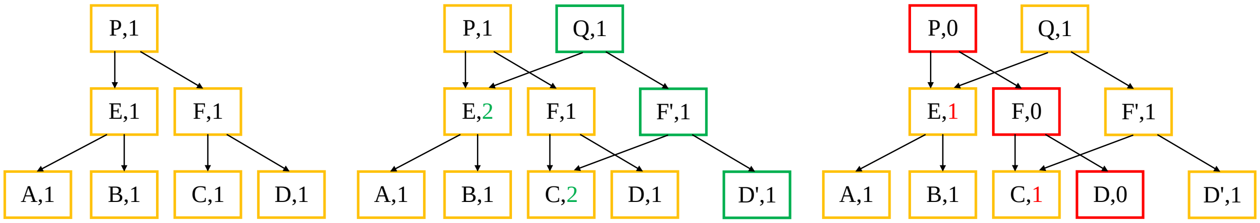
While NVM-based in-memory file systems allow a large amount of data to be stored and processed in memory, the benefits of NVM are compromised by hardware and software errors. Bit flipping and pointer corruption are possible due to the byte-addressability of NVM media. Moreover, The applications that 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.

Versioning, snapshotting and journaling 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 [5]. 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 [6] 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.

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 [8], 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 [9]. However, such design introduces unnecessary overhead to in-memory file systems due to the following two reasons.

Firstly, the key to taking fast and reliable snapshots is to reduce the amount of metadata to be flushed to persistent



(a) Original tree P (b) Inserting node D' and forming a new tree Q (c) Deleting tree P

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

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 writeback. 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 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 preserve 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, improving by a factor of 9.7 and 6.6, respectively. The contributions of this work are summarized as follows.

• We are the first to solve 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 one of the most widely used approaches to build efficient file systems that support snapshotting. It allows CoW-based file systems to better scale with the size of their storage and create snapshots in a space-efficient mode.

Figure 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.

Figure 1(a) illustrates an original state of a simplified CoW friendly B-tree with reference counts. In Figure 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 haven't changed, they are accessible by the newly created tree Q.

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

As is shown in Figure 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 4KB. 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.

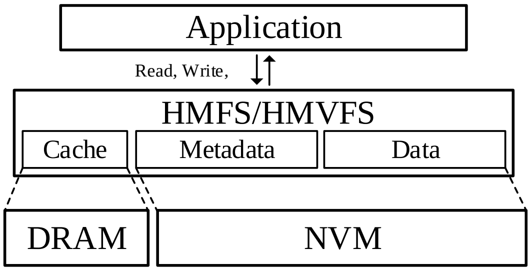


Figure 2. Architecture of HMFS/HMVFS

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. 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 2TB and is still efficient for small files (less than 4KB) with inline data. To support even greater need of the maximum file size, a quadruple-indirect pointer or a second triple-indirect pointer can be added to the inode, which expands the maximum file size to 2PB or 4TB.

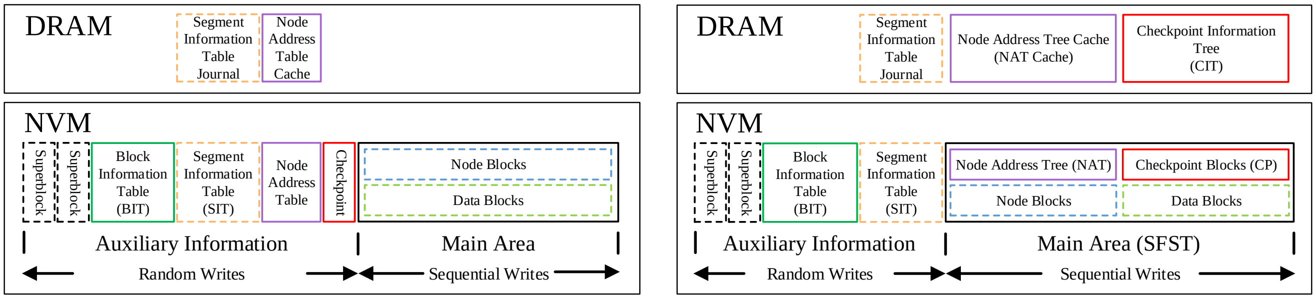
The idea of node and data blocks is inspired by F2FS [10]. In the design of traditional log-structured file system (LFS), any update to low-level data blocks will lead to a series of updates in direct node, indirect node and inode, resulting in serious write amplification. F2FS uses B-tree based file indexing with node blocks and node address table to eliminate update propagation (i.e. wandering tree problem [11]), only one node block will be updated if a data block has been modified.

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 Figure 4 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



(a) Layout of HMFS

(b) Layout of HMVFS

Figure 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.

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 Figure 3(b) shows, the auxiliary information is stored at a fixed location and updated with random writes, whereas the actual blocks of B-tree are stored in the main area and updated with sequential writes. We adopt the idea of CoW friendly B-tree [9] to the whole file system on version basis, in which continuous file system snapshots are organized as SFST with shared branches, as shown in Figure 4. With the idea of lazy reference counting [8], 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 (Figure 4) in HMFS to convert it into a Hybrid Memory Versioning File System (HMVFS), the layout of HMVFS is shown in Figure 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 4KB block located in the main area of HMVFS, and yields strictly sequential writes as Figure 3(b) shows.

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 Figure 4. 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.

The blocks of files from different versions are located in the main area. They are updated through log-structured writes and managed by our in-memory stratified file system tree (SFST). To look up a file F in the directory D, we search the data of D and traverse its directory entries with hash lookup to find the entry with file name F. Once the directory entry is located, we can extract the inode number i from it. After that, we perform a top-down search in the NAT of current version in DRAM to locate the address of the node block with the inode number i. The address in that NAT entry will point to the inode block of file F.

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

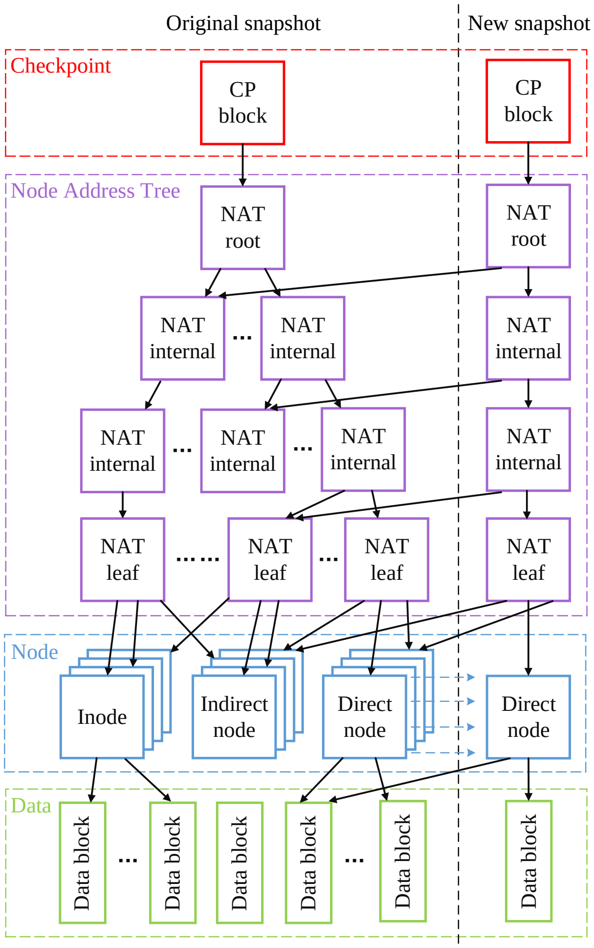


Figure 4. Logical layout of SFST

file structure of HMVFS, the connection between nodes (⇒) is 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 Figure 4 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 Figure 4). 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 [8] 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 SFST, the directories contain inode numbers, which remains unchanged during file updates. In some file-grained versioning approaches (like NILFS [12], ZFS [13]), 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 Figure 4, 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 to 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 collection*, etc.

Segment Information Table (SIT) in HMVFS is the same as the SIT in HMFS, but we revise the process of SIT updates 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 4KB block.

4. Implementation

In this section, we provide implementation details of SFST. We first illustrate how to manage snapshots with snapshot operations. We then show the metadata operations on which SFST relies to ensure consistency and how garbage collection handles block movements in multiple snapshots. Finally, we demonstrate how to recover using 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 512MB (An NAT leaf contains 256 NAT entries and a direct node block contains 512 pointers to data blocks, thus size $\approx 256 \times 512 \times 4\text{KB} = 512\text{MB}$) 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.

`Checkpoint` is triggered on one of these occasions: (1) after an `fsync` system call from the applications; (2) after garbage collection; (3) when explicitly specified by the user. Background garbage collection is carried out every five minutes automatically, hence `Checkpoint` is conducted at most 5 minutes since last execution. Also, `Checkpoint` only builds NAT and the checkpoint block of SFST in the main area, as node blocks and data blocks are updated directly using XIP. With the space and time efficient design of snapshots, we manage to do frequent checkpoints with low impact on performance.

Snapshot deletion is triggered by the user manually or by background cleaning. A snapshot is considered old enough for deletion if the version number is much earlier than current version number and is not specified to be preserved by the user.

We can keep up to 2^{32} different versions of the file system (due to the 32-bit version number). We also provide interfaces for users to delete snapshots manually, or set up *preserve* flags and *time of effective* to delete old and unwanted snapshots while maintaining the important ones. For most file systems, taking snapshots at such high frequency automatically and preserving a lot of snapshots hurt the I/O performance and waste a large amount of space, but SFST creates space-efficient snapshots that imposes little overhead in snapshotting and I/O performance.

4.1.1. 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 Figure 4. 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. If the system crashes before step 4, the file system will start from the last completed snapshot and discard any changes in this version. After the snapshot becomes valid in step 4, the file system updates SIT according to the SIT journal on NVM. If a system crash happens during or after step 5, we redo `Checkpoint` from this step. 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 super block list, the snapshot becomes valid. Also, HMVFS will always start from the last completed snapshot after a crash or reboot.

4.1.2. 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 Figure 1(c). We start the decrement of `count` from the checkpoint block through the whole branch of SFST recursively. To recover from a sudden crash during snapshot deletion, we keep a record of the snapshot version number and the progress of deletion. 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, it will redo the deletion before available to users. Algorithm 1 describes the snapshot deletion operation in pseudocode.

Algorithm 1 Delete snapshot

```
function SNAPSHOTDELETION($CheckpointBlock$)
    Record $CheckpointBlock$ in $superblock$;
    $BlockDeletion(CheckpointBlock)$;
    Remove $CheckpointBlock$ from CIT;
    Call $GarbageCollection$;
end function
function BLOCKDELETION($block$)
    Record $block$ in $superblock$;
    $count[block] - -$;
    if $count[block] > 0$ or $type[block] = data$ then
        return
    end if
    for each $ptr$ to child node of $block$ do
        $BlockDeletion(ptr)$;
    end for
end function
```

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 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.

4.2. 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 Figure 4 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 *node* information of all the blocks in BIT, and utilize the byte-addressability of NVM to update the information at the granularity of byte.

To ensure the consistency of the whole file system, we expect NVM to be attached to memory bus rather than PCI bus, also NVM should support at least 8 bytes atomic write. Then 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.

```
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.

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 internal	index code in NAT
NAT leaf		
inode	NAT leaf	NID
indirect		
direct		
data	inode or direct	NID of parent node

- 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 Figure 4.
- 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 [8], [14]. 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.

For start_version, node_id, offset and type, they are set only once during block allocation by one thread, the file system can undo these changes according to version number. We can also call fsck to scan BIT and invalidate end_version for an uncommitted version of snapshot. For count, the only operation which will decrease count and may cause inconsistency is snapshot deletion. We provide Algorithm 1 in Section 4.1.2 to ensure consistency.

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. 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 end_version of the target block. The byte-addressability and low access latency of NVM accelerates the update process substantially.

4.4. 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 4.1.1.

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.

5. Evaluation

We evaluate the performance of HMVFS using various real workloads. For each workload, we first 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. Then, 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 2GHz processors and 512GB DRAM, running the Linux 3.11 kernel. For EXT4, BTRFS, NILFS2, we use ramdisk carved from DRAM and configure 128GB as ramdisk to simulate NVM. In the case of HMVFS and PMFS, we reserve 128GB 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, 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 Filebench suite [15] to emulate five common workloads. The random-read, random-write and create-files workloads generate basic I/O performance results of typical file operations. 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 also use postmark to emulate different biases of read/write operations in real scenarios. Postmark [16] is a single-threaded synthetic benchmark that simulates a combined workload of email, Usenet, and web-based commerce transactions. This benchmark creates a lot of small files and performs read/write operations on them.

We compare HMVFS with two popular versioning file systems, BTRFS [9] and NILFS2 [12], to show the efficiency of snapshotting operations of HMVFS. We also compare HMVFS with two non-versioning but state-of-the-art in-memory file systems with the best I/O performance: PMFS [1] and EXT4 [17]. PMFS is a typical in-memory file system and EXT4 is one of the most widely-used traditional file system. Besides the IOPS results of every benchmark, we also show *efficiency*, the reciprocal of time overhead on creating a snapshot, to illustrate how HMVFS achieves snapshotting with neglectable performance overhead.

In the following sections, we present our experimental results to show how HMVFS reduces the performance overhead imposed by snapshotting operations. Section 5.2 gives a basic understanding on how files and directories affect the performance of file system Section 5.3 demonstrates the overhead of snapshot creation and deletion; Section 5.4, 5.5 and 5.6 show how different total numbers of files, directory structures and percentages of reads of workloads affect the I/O performance and the overhead of snapshotting in detail.

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 result shows that HMVFS imposes acceptable I/O overhead to achieve snapshotting.

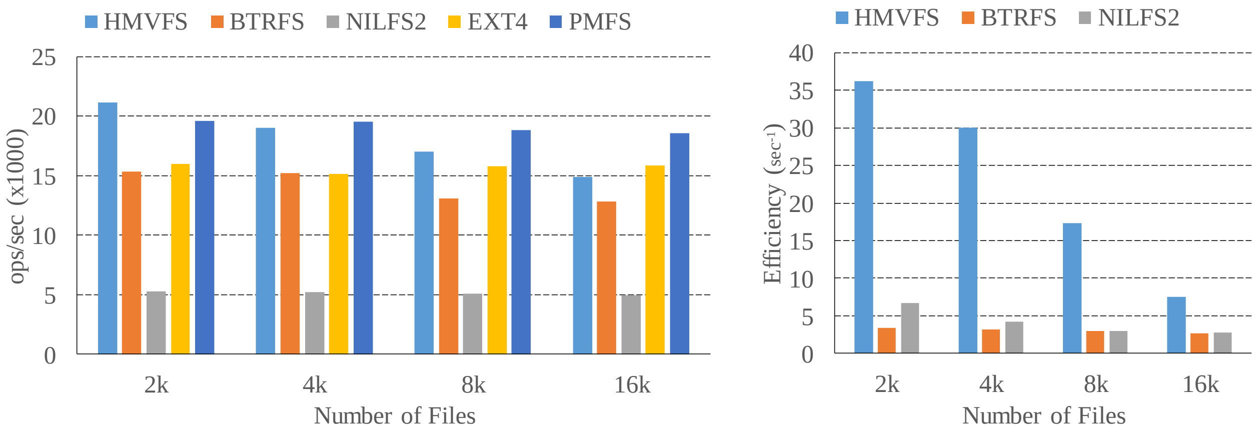
To demonstrate the overall I/O performance of HMVFS, we first run randomwrite and randomread workloads on single file. These two workloads perform random writes and reads on a pre-allocated file. We take average I/O results of five runs on a 5GB file. The results in Table 2 show that on randomwrite, EXT4 performs the best, HMVFS and PMFS perform close to EXT4. BTRFS and NILFS2 perform 46% and 29% the write speed of HMVFS. On randomread, all five file systems perform close to each other. This indicates that HMVFS has similar efficiency compared with state-of-the-art single version in-memory file system, and has additional advantages to traditional approaches.

TABLE 2. Performance of random read and write

	Random read (ops/sec)	Random write (ops/sec)
HMVFS	25493	21726
BTRFS	26194	10026
NILFS2	25460	6379
EXT4	25447	22721
PMFS	25728	21628

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 1GB 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 (1GB each), and measure their overhead of creating snapshots. For BTRFS, the overhead is strongly related to the number of



(a) Operations per second

(b) Efficiency of snapshot creation

Figure 5. Performance of file systems under fileserver workload

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 isn't 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 minutes, take snapshots every 5 minutes, 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

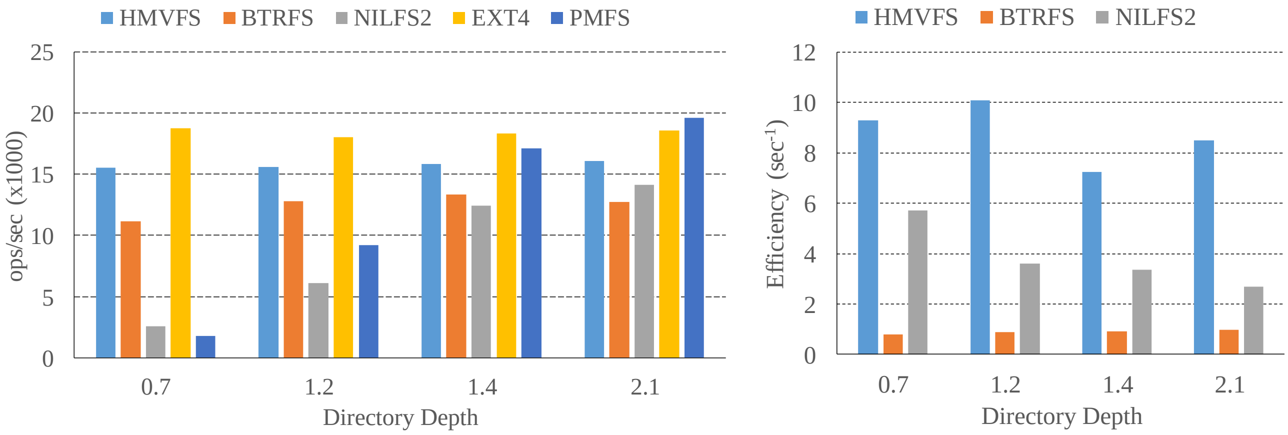
Different numbers of files in file systems lead to different overhead of creating snapshots. We use fileserver workload from Filebench to emulate I/O activities of a simple file server containing different amounts of files. The file operations include creates, deletes, appends, reads, and writes. In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

Figure 5(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 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, but 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. 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



(a) Operations per second

(b) Efficiency of snapshot creation

Figure 6. Performance of file systems under varmail workload

in memory, lots of redundant blocks of data and metadata are written for every append operation which takes extra time and space.

Figure 5(b) shows the efficiency 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) metadata file, and appends the changes on every snapshot [18]. However, this metadata 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.

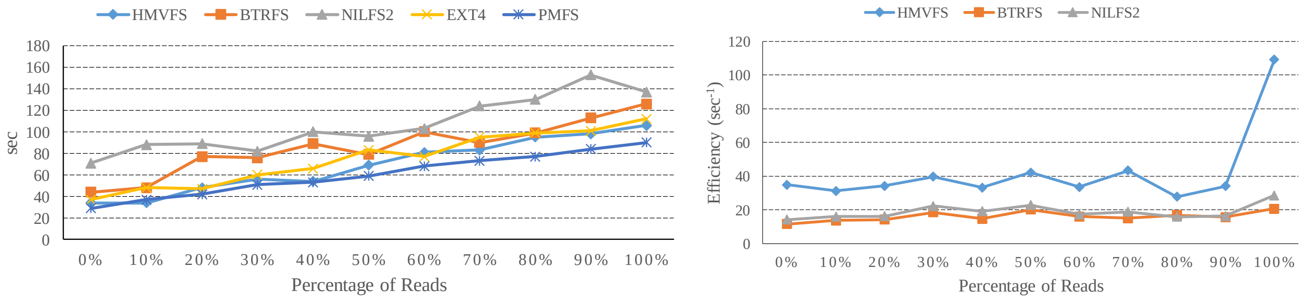
5.5. Impact of Directory Structure

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 16KB, the number of total files to 100,000, mean append size to 8KB and run time to 5 minutes. We run varmail on each different structures of directories three times and report the average readings.

Figure 6(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], [17]. 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 doesn't 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



(a) Duration of performing transactions

(b) Efficiency of snapshot creation

Figure 7. Performance of file systems under Postmark workload

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).

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.

Figure 6(b) shows the efficiency 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 can not 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. Impact of Read vs. Write

Postmark emulates an email server that concurrently performs read and write operations. We set the number of postmark transactions to 200,000 and show the time taken by running these transactions with 10,000 512KB files. Buffering is switched off all the time in case of data buffering from DRAM. We run postmark three times on different read bias number from 0 to 10, i.e. 0% to 100% of the total transactions are reads and the rest of the transactions are writes (appends), and report the average readings.

Figure 7(a) shows the results of running these transactions by different file systems. We see that PMFS performs the best among all percentages of reads while NILFS2 performs the worst. On average, HMVFS, BTRFS, NILFS2, EXT4 take 13%, 44%, 85%, 24% more time to complete the transactions than PMFS takes. Among these file systems, HMVFS performs the closest to PMFS which is only 13% worse than PMFS. The result is acceptable since HMVFS takes multiple snapshots during performing transactions.

Different snapshotting overheads in Figure 7(b) also show that HMVFS is the right choice for in-memory versioning file systems. To get the accurate overhead of snapshotting, we take snapshot A after preallocation, snapshot B after 200,000 transactions are committed, snapshot C after cleanup of postmark. Figure 7(b) shows the efficiency of taking snapshot B.

As is shown in the figure, HMVFS consumes the least time for taking snapshots. On average, BTRFS and NILFS2 takes 2.67 and 2.25 times of HMVFS does. Although the bias of reads is rising, the efficiency of taking snapshots doesn't grow much until the percentage reaches 100%. For any valid file in a snapshot, a little change in data leads to a write amplification of new blocks in the whole file, such as new inode, new direct pointer block, new data block, etc. Also, the metadata and directory entry of the file have to be updated to ensure consistency. HMVFS utilizes SFST to eliminate the write amplification problem by a limited height, but there is no such mechanism to prevent write amplification in BTRFS and NILFS2.

We set a balanced ratio for creation and deletion transactions, therefore, the snapshot contains data updates not only from write transactions, but also from file creation and deletion transactions. HMVFS also handles these transactions well, the directory structure and inline file data ensures the efficiency of snapshotting, which is the reason why HMVFS only takes approximately 40% of the snapshotting overhead of traditional versioning file systems.

6. Related Work

Journaling and snapshotting are two main mechanisms which ensure file system consistency from rebooting after sudden power failure. Usually, 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 [19], LinLogFS [20], UBIFS [21], XFS [22], NTFS [23] and many other traditional on-disk file systems use this simple implementation to support metadata consistency.

Among log-structured file systems, single snapshot is often an efficient way to store a consistent state of whole file system. YAFFS2 [24], F2FS [10] and Sprite LFS [11] store a valid snapshot back to persistent storage once in a while. Log-structured file system also suits 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.

CORFU [25] utilizes log-structured writes to expose a cluster of network-attached flash devices as a single, shared log which can be accessed concurrently by multiple clients over the network. It focuses on mapping, tail-finding and replication protocol, and it achieves wear-leveling for clustered flash devices. Although it shares some resemblance to the log-structured design in HMVFS and obtains cluster-wide consistency guarantees, it does not provide a versioning or snapshotting mechanism to the system which is the main focus of this work.

To better cope with emerging non-volatile memory, new in-memory file systems such as PMFS [1], SCMFS [2], and BPFS [3] leverage byte-addressability and random access features of NVM to gain maximum performance benefit [4]. But these file systems ensure only data consistency and can not be integrated with snapshot mechanisms. Among all these file systems, PMFS is the only open source NVM-aware file system available, hence we only include PMFS in our evaluation.

Consistent and Durable Data Structure (CDDS) is proposed in [26] 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.

In ZFS [13], 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 [9] 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.

Ext3cow [27], which is built on the ext3 file system, provides a time-shifting interface that permits real-time and continuous views of data in the past. Users can keep multiple versions of files in Ext3cow, and access them by appending timestamps. Ext3cow implements the idea of Copy-on-Write by maintaining an epoch number on a per-file basis, which makes the overhead of creating a file system snapshot proportional to the total number of files. Ext3cow employs a bitmap stored in each inode and indirect block to track the blocks for Copy-on-Write, which lacks facilities to handle B-tree structures and therefore hard to scale.

NILFS [12] 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.

7. Conclusion

As the need of NVM-based file system increases, snapshotting has become a crucial component to fault tolerance. To utilize the byte-addressability of memory and lower the overhead of snapshotting, we present HMVFS, a new hybrid memory versioning file system. 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 function and only takes little performance cost compared with other in-memory file systems. The snapshot 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.

8. Acknowledgment

We thank the shepherd Swaminathan Sundararaman and anonymous reviewers for their insightful and helpful comments, which improve the paper. We also thank Hong Mei and Yanyan Shen for their constructive suggestions and comments on this paper. 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.

References

[1] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 15.

[2] X. Wu and 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.

[3] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and 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.

[4] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on nvm," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 2015, pp. 1–14.

[5] D.-I. J. Stender, "Snapshots in large-scale distributed file systems," Ph.D. dissertation, Humboldt-Universität zu Berlin, 2013.

[6] G. Lu, Z. Zheng, and 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.

[7] C. Dragga and D. J. Santry, "Gctrees: Garbage collecting snapshots," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 2015, pp. 1–12.

[8] O. Rodeh, "B-trees, shadowing, and clones," *ACM Transactions on Storage (TOS)*, vol. 3, no. 4, p. 2, 2008.

[9] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, p. 9, 2013.

[10] C. Lee, D. Sim, J. Hwang, and 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.

[11] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[12] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, "The linux implementation of a log-structured file system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 102–107, 2006.

[13] J. Bonwick and B. Moore, "Zfs: The last word in file systems," 2007.

[14] O. Rodeh, "Deferred reference counters for copy-on-write b-trees," Technical Report rj10464, IBM, Tech. Rep., 2010.

[15] Filebench, http://filebench.sourceforge.net.

[16] J. Katcher, "Postmark: A new file system benchmark," Technical Report TR3022, Network Appliance, Tech. Rep., 1997.

[17] Ext4, https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[18] C. Min, S.-W. Lee, and Y. I. Eom, "Design and implementation of a log-structured file system for flash-based solid state drives," *Computers, IEEE Transactions on*, vol. 63, no. 9, pp. 2215–2227, 2014.

[19] S. C. Tweedie, "Journaling the linux ext2fs filesystem," in *The Fourth Annual Linux Expo*, 1998.

[20] C. Czezatke and M. A. Ertl, "Linlogfs-a log-structured file system for linux." in *USENIX Annual Technical Conference, FREENIX Track*, 2000, pp. 77–88.

[21] A. Hunter, "A brief introduction to the design of ubifs," *Rapport technique, March*, 2008.

[22] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system." in *USENIX Annual Technical Conference*, vol. 15, 1996.

[23] R. Nagar, "Windows nt file system internals: a developer's guide," 1997.

[24] C.-H. Wu, T.-W. Kuo, and 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.

[25] D. Malkhi, M. Balakrishnan, J. D. Davis, V. Prabhakaran, and T. Wobber, "From paxos to corfu: a flash-speed shared log," *ACM SIGOPS Operating Systems Review*, vol. 46, no. 1, pp. 47–51, 2012.

[26] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 5–5.

[27] Z. Peterson and R. Burns, "Ext3cow: a time-shifting file system for regulatory compliance," *ACM Transactions on Storage (TOS)*, vol. 1, no. 2, pp. 190–212, 2005.