A Consistency Mechanism for NVM-Based in-Memory File Systems

Jin Zha
Shanghai Jiao Tong University
qweeah@sjtu.edu.cn

Linpeng Huang
Shanghai Jiao Tong University
huang-lp@cs.sjtu.edu.cn

Linzhu Wu
Shanghai Jiao Tong University
wulinzhu@sjtu.edu.cn

Sheng-an Zheng
Shanghai Jiao Tong University
venero1209@sjtu.edu.cn

Hao Liu
Shanghai Jiao Tong University
liuhaosjtu@sjtu.edu.cn

ABSTRACT

Non-Volatile Memory (NVM) has evolved to achieve non-volatility and byte-addressability with latency comparable to DRAM. This inspires the development of a new generation of file systems, namely *NVM-based in-memory file systems*, which include NVM on memory bus and allow in-NVM data to be directly accessed like DRAM. Meanwhile, an important issue, the *consistency problem*, arises as a new challenge. That is, the direct modification to the in-NVM data can be interrupted by arbitrary crashes in the system, which results in part of the modification being durable and others being lost. Traditional consistency mechanisms assume the existence of DRAM buffering and hence cannot be applied to this hybrid memory architecture. While several consistency methods have been proposed for NVM-based in-memory file systems, most of them have side-effects including unfriendliness to DRAM and penalties on concurrency control, which degrade the system performance.

In this paper, we propose a novel mechanism to guarantee the consistency of NVM-based in-memory file systems. We abstract the storage area as a layered structure and employ a lazy-validated snapshot strategy to achieve a high consistency level. Since every consistency method comes with a cost, we introduce several algorithms to efficiently deal with block-sharing and reduce the overhead of consistency mechanism. The experimental results show that our mechanism incurs negligible consistency overhead and outperforms a state-of-the-art snapshot file system by reducing the latency of snapshot taking and removal by 95% and 60% respectively.

Keywords

File system; non-volatile memory; performance; consistency; snapshot;

CF'16, May 16-19, 2016, Como, Italy
© 2016 ACM. ISBN 978-1-4503-4128-8/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2903150.2903160

1. INTRODUCTION

Byte-addressable, non-volatile memory (NVM) draws growing attention from both industry and academia. Technologies like Memristor [18], Phase Change Memory (PCM) [11] and 3D XPoint [9] provide fine-grained access control and persistence for storage with latency comparable to DRAM. All these benefits prompt the inclusion of NVM in the *processor memory subsystem* where it can provide larger memory capacity than DRAM and still be competitive in terms of performance [11]. Specifically, NVM is placed side-by-side with DRAM on the memory bus, available to ordinary loads and stores by CPU [5]. Based on such hardware architecture, there has been significantly increasing interest in developing a new generation of file systems [12, 5, 19], namely the *in-memory* file systems. By employing NVM as primary storage device, those systems eliminate high overhead imposed by DRAM caching, thus improving I/O throughput significantly. However, consistency problem comes as a challenging issue along with such improvement.

Inconsistency occurs when a durable modification is performed in an incomplete way. That is, part of the modification becomes durable and others are lost. This is usually caused by power failures or arbitrary crashes in the system. In traditional disk-based file systems, modifications to data (including both data and metadata) will firstly be performed in DRAM buffers and then be flushed to the disk via a flushing operation called *sync*. To avoid the occurrence of inconsistency, *sync* is committed as a transactional operation and hence prevents a modification from being halfway completed. However, for in-memory file systems, data is typically updated in NVM without any DRAM buffering. If any crash occurs before the execution of the *sync* operation, the modification to the in-NVM data may become uncommitted, which results in an incomplete modification. Half-written data can corrupt existing files and more importantly, inconsistency between data and *allocation structures* [17] (e.g., bitmaps, free lists) may contaminate the whole system.

Various approaches have been proposed to address the consistency problem for NVM-based in-memory file systems. BPFS [5] uses a tree-based file system structure and performs copy-on-write (CoW) in NVM. File data is updated before *file metadata* [17]. Changes are gathered up to the root of the file system tree until one write is executed to commit all the modifications. By making the last write atomically committed, the consistency of the whole file system is guaranteed. PMFS [12] uses a fine-grained undo journaling method to guarantee the *metadata consistency* [4]. When a file is modified, logs are written first, followed by the modifications to the medatata. Recovery is performed on the next mount to undo any existing changes from uncommitted operations. However, all these approaches do not allow allocation structures to be accessed directly or even stored in NVM (see details in Section 5). Such kind of implementation has two major disadvantages as follows.

The first disadvantage is about the system performance. In-DRAM allocation structures have to be reconstructed every time when the file system is being mounted. This reconstruction is not a simple sequential scan of the NVM, but a logical traverse through the inner index structures of all files. When the in-use NVM space becomes larger, the reconstruction will cause extremely high latency. Furthermore, to minimize the storage cost of DRAM, those derived structures are usually implemented as a linked list for fast insertion and deletion, therefore it does not allow any random access to the list nodes and hinders the concurrency between file operations.

The second disadvantage is the high storage cost of DRAM. Consider PMFS [12] as an example. It needs 32 bytes for containing a free block in its free list. Suppose there are totally 512GB free space scattered into 4KB blocks, it will require 4GB exclusive DRAM space for node slab cache. Such a proportion is unacceptable especially when the current DRAM-based main memory systems have been starting to hit the limit of power and cost [11].

In order to achieve reliability with better overall performance and being DRAM-friendly, we introduce a new consistency mechanism for NVM-based in-memory file systems. By leveraging a layered abstraction of storage area, we design a novel per-block structure and based on which, we propose a lazy-validated snapshot strategy and several efficient block-sharing algorithms to address the consistency problem. We implement our mechanism in Hybrid Memory File System (HMFS), which is an open-source in-memory snapshot file system based on a NVM/DRAM hybrid memory architecture, available in [2]. The experimental results show that comparing to a disk-based snapshot file system, latencies of snapshot functionalities are reduced dramatically. Also, our proposed mechanism provides almost the same performance as in-memory file systems, which proves that minor cost is imposed to provide high reliability.

We summarize the contributions of this paper as follows.

- We propose a novel mechanism to guarantee the consistency of NVM-based in-memory file systems. We abstract the NVM space into different kinds of blocks and group them into several logical layers, based on which we employ a lazy-validated snapshot strategy to achieve a high consistency level.
- To achieve high performance, we introduce a new per-block structure and several block-sharing algorithms that can reduce the overhead of snapshot management significantly.
- We conduct extensive experiments to evaluate the performance of our consistency mechanism. The results show that our method saves more than 90% and 65% overhead for snapshot taking and removal respectively, compared to the state-of-the-art consistency methods.

The rest of the paper is organized as follows. Section 2 gives an overview of HMFS including its system architecture physical layout, logical structure and DRAM facilities for consistency maintenance. In Section 3, we introduce our consistency mechanism. In Section 4, we evaluate how this mechanism works in practice. Section 5 provides related work and Section 6 concludes the paper.

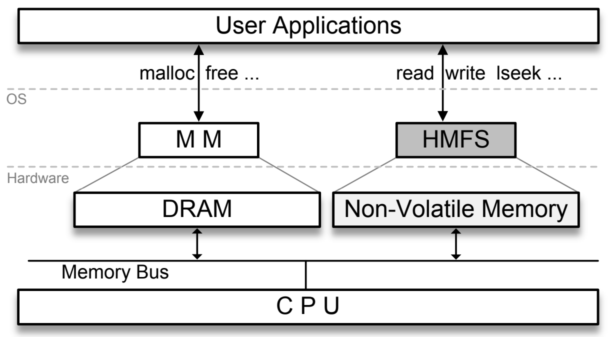


Figure 1: HMFS Architecture

2. HMFS OVERVIEW

In this section, we provide an overview of our prototype system, the Hybrid Memory File System (HMFS). Specifically, we include the system architecture, physical composition and the layered structure that are the basis of our consistency mechanism.

2.1 System Architecture

Figure 1 shows the system architecture of HMFS. In the hardware level, NVM is connected to memory bus and allows direct access from CPU. In the operating system (OS) level, Memory Management (MM) manages the DRAM space and HMFS is responsible for NVM. In user space, applications can use HMFS like traditional disk-based file systems with normal file routines.

2.2 Physical Layout

In HMFS, the minimal unit of storage is a 4KB *block*. Several (512 by default) contiguous blocks compose a *segment*. Block allocation in a segment is performed sequentially. The size of a segment is configurable but fixed when the volume is formatted. By employing this indirect management of blocks, we are able to allocate space of different sizes efficiently and reduce journal overhead effectively.

Figure 2 shows the physical layout of HMFS for one volume. It consists of one area for in-place updates (random writes) and another area for CoW updates (sequential writes).

In the random write area, there are three sub-areas:

- **Superblock (SB)** stores the basic volume information. It takes up one extra block for redundancy backup.
- **Block Information Area (BIA)** is the allocation structure. It contains start version number, type, valid bit and owner information of every block in Main Area. Start version number identifies the snapshot that validates this block. Owner information is used for optimizing reverse lookup and modification. Valid bit

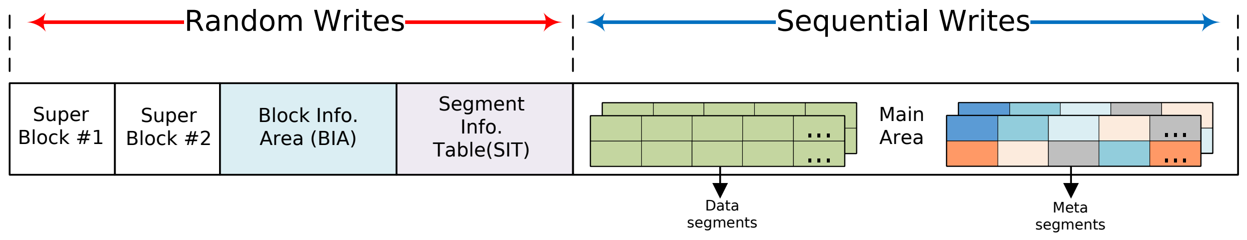


Figure 2: Physical Layout of HMFS

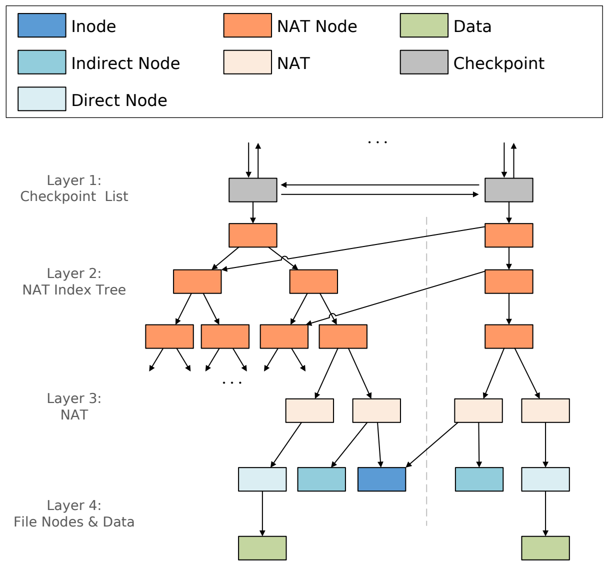


Figure 3: Logical Hierarchical Structure

indicates whether this block is valid or not. BIA only accepts in-NVM updates and thus will not be cached in DRAM.

- **Segment Information Table (SIT)** records usage information like the last update time and how many blocks are valid of every segment in Main Area. This information is consistent with the latest snapshot and will be used to select victim segment during cleaning process in garbage collection.

Block allocation is based on BIA and SIT. During mounting, segments with no valid block will be marked as empty in a in-DRAM bitmap. Every time a segment is full, a new empty segment will be found in the bitmap. When a block is allocated, the corresponding BIA entry will be updated *softly*, which means information like start version number, block type and owner information is recorded in the entry, but valid bit is not set.

Sequential write area, or Main Area, is filled with 4KB blocks. Block and segment allocations in Main Area are all log-structured. There are two kinds of segments in this area, *data segments* and *meta segments*. Data segment contains only data blocks of files (or directories). Meta segment is more complicated. It contains 6 kinds of blocks:

- **Inode Block** represents a file (or a directory). **Indirect Node Block** records *node id* (NID) of child direct nodes. **Direct Node Block** has block addresses pointing to child data blocks. HMFS uses the same pointer-based file indexing structure with F2FS [8]. We do not further detail such structure in this paper.

- **NAT Node Block** is a non-leaf node in the index structure of Node Address Table (NAT), *NAT Index Tree*. One block contains 512 addresses of its child node blocks.

- **NAT Entry Block** is a leaf node of NAT Index Tree. It contains several NAT entries. Given a NID, a *file node* (including inode, indirect node and direct node) block can be located through a NAT entry.

- **Checkpoint Block** identifies a snapshot in HMFS. It stores a unique version number, system state information, connections in checkpoint list and the block address of the root node in NAT Index Tree.

2.3 Logical Structure

Based on the physical layout, the logical hierarchical structure of Main Area blocks is shown in Figure 3.

In the first layer, checkpoint blocks are linked pair-wisely and compose a *Checkpoint List*. In the second layer, NAT node blocks form a *NAT Index Tree*. The whole NAT is too large to make a complete copy for every snapshot. By sharing intact nodes between successive snapshots, NAT Index Tree effectively reduces such overhead. In the third layer, there are the leaf nodes of NAT Index Tree, blocks of *NAT*. Grouped by NID, NAT entries are merged in to those blocks. As is shown in Figure 4, NID can be divided into two parts. Higher 23 bits indicate the per-level offsets of child block addresses in parent NAT node blocks. Lower 9 bits record the inner-block offsets of NAT entries in NAT entry blocks. Block addresses in the NAT entries point to inode, indirect node and direct node blocks in the *Index-Structured File*, and addresses in those blocks finally point to file data blocks.

Such a hierarchical structure achieves a very ideal abstraction of Main Area. Layer 4 absorbs all the file modifications directly in NVM. Layer 3 is cached in DRAM on demand for transactional flush process like disk-based file systems. Layer 2 guarantees a reasonable overhead of each snapshot. Layer 1 keeps track of every integrated or inconsistent but recoverable snapshot.

Varied consistency techniques are employed in those layers. In Layer 2, 3 and 4, we do *CoW* in Main Area and *Soft Updates* in BIA. Deferred block validation in BIA can be done very efficiently because it only requires a sequential scan on newly allocated NAT entries. Involved segment entries will be *Journaled* during flushing. This guarantees the integrity between BIA, SIT and the latest snapshot. Blocks

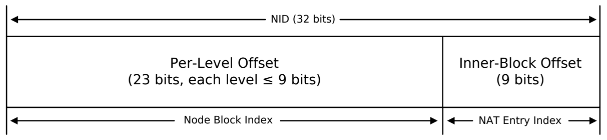


Figure 4: Structure of NID

in Layer 1 keep track of snapshots. On every file system mounting, a *File System Check* will be done on system state information in latest snapshot and inconsistencies will be fixed.

2.4 Volatile Data Structures

To be DRAM-friendly, data blocks, super blocks and BIA requires direct updates in NVM. However, some metadata still need to be cached in DRAM. Here we detail why these metadata need to be cached and how they are managed in DRAM.

(1) NAT: NAT is cached in DRAM on demand to exhibit flush process like disk-based file systems. As is shown in Figure 3, the in-NVM organization of NAT is a index tree. But in DRAM, cached NAT entries is managed by a *NAT Radix Tree* keyed with NID. On system mounting, only NAT entries of root directory will be added to the tree. When a certain file node is read, written or allocated, its entry will be added to the radix tree. This on-demand growth guarantees a moderate cache cost of DRAM. Also when a file node is modified, corresponding NAT entry will be inserted into the *Dirty NAT List* in ascending order of NID.

(2) SIT: As mentioned before, direct updates in NVM are not allowed in this area. Hence we use a *Segment Entry Array* to absorb modifications from segment-related operations (e.g., garbage collection or new block allocation). Involved segments will be added to *Dirty SIT List* and flushed back to SIT after snapshot taking.

(3) Checkpoint: Once been written to NVM, checkpoints are seldom modified. But when removing a snapshot or doing garbage collection, the validity of a certain version number is frequently queried. Thus we cache checkpoints of active snapshots in *Checkpoint Radix Tree* to optimize such work. This tree is keyed by the version numbers of snapshots and provides validity checking in O(1).

3. CONSISTENCY MECHANISM

In our mechanism, we employ a lazy-validated snapshot strategy to keep in-NVM allocation structures consistent with blocks in Main Area: Validation for new block allocation is deferred to snapshot taking, invalidation for file deletion or truncation is delayed to snapshot removal.

A new problem, block sharing, arises with such strategy. To deal with it, there are many classic methods like hierarchical reference counting [14](*ref-count* in short), generation chain [6], reference bitmap [7], etc. However, for two reasons, the per-block structure ad hoc basis in those designs cannot be applied to NVM-based in-memory file systems. The first is about the consistency maintenance between this per-block structure and allocation structure. Take ref-count as an example, every time a block is duplicated by CoW updates in common file operations(*create*, *write*, etc), it requires explicit increments in its brother blocks' reference counters. If such increments are reflected to NVM immediately, extra tracing journal must be added for decrement in crash recovery. If such increment is deferred to *sync*, then extra per-block journal should be taken. Either way causes unnecessary overhead. The second reason is that storing the reference relationship is not optimal. Additional space is required by write-ahead logging of batched reference/dereference operations.

In our design, we use a start version number to identify which snapshot validate the block. Hence immediate changes in its brother blocks are avoided in CoW updates. To eliminate unnecessary complexity in write-ahead logging, we use only a valid bit which supports simplified redo operation in crash recovery. Detailed algorithm about reference tracking with this per-block structure will be discussed in the following section.

3.1 File Updating

Three file operations involves consistency concern: write, delete and truncate.

When a file is deleted or truncated, their corresponding NAT entries in DRAM will be set to zero. If the related blocks is still used by old snapshots, real deletion in NVM will be done when removing those snapshots. Thus, truncation and deletion causes no modification in BIA.

There are two kinds of file write in HMFS. If the written area overlaps with existed file content from previous snapshots, then we do block-grained CoW to related data before writing new data in. This is called *Stacking Write*. If those areas do not overlap or related file data is newly created in this snapshot, we use a more efficient in-place update method, *Normal Write*. Normal updates optimize speed of random writes and avoid meaningless inner snapshot garbage.

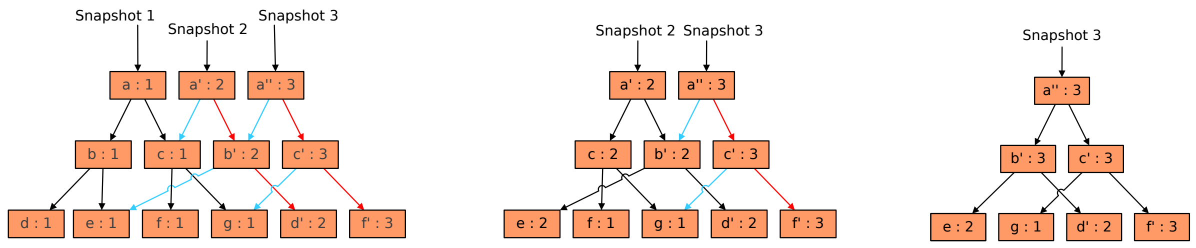
For both kinds of writes, although related data blocks and BIT entries are directly updated in NVM, validation is deferred to snapshot taking process. With this soft update, unexpected crashes before a snapshot taking will not cause any inconsistency, i.e. when a crash occurs between two flushes, all the modifications will be completely discarded and no recovery needs to be done.

3.2 Snapshot Taking

Taking a snapshot includes re-organizing updated file blocks and validating newly allocated blocks. It contains two parts of work: 1) build up a NAT Index Tree, SIT journal, 2) change the latest snapshot in checkpoint list.

The first part takes up the most time. To make it more efficient, we use a bottom-up merging strategy. Modified NAT entries are merged into NAT blocks and then added to the index tree alternately. As mentioned before, entries in dirty NAT list are ordered by their NID. Thus such merge can be done very quickly. After block address of the root node is recorded in the new checkpoint block, a new NAT Index Tree for the next snapshot is built. Before changing the latest snapshot, entries in Dirty SIT List need to be journaled. The journal overhead is extremely low, which costs only 16 bytes per modified segment (2MB at least). Also, the address of new checkpoint block should be recorded in the latest snapshot's checkpoint block.

In the second part, the 8-bit file system state in latest snapshot's checkpoint block is atomically changed from 'normal' to 'changing the latest snapshot'. A sequential scan will then be conducted on the newly allocated NAT entries to



(a) Add snapshot 2 and 3 (b) Remove snapshot 1 (c) Remove snapshot 2

Figure 5: A scenario of taking and removing snapshots

validate corresponding blocks. When the scan is done, journaled SIT entries are flushed to NVM. After the flushing, the new checkpoint will be added to the checkpoint list with super block pointing to it and henceforth a new snapshot is taken.

A scenario of snapshot taking is illustrated in Figure 5(a), the number in block presents the start version number, the letter denotes the block's alias. Red arrow means CoW, blue arrow means reference. To facilitate understanding, we show only NAT node blocks. Take $Block_{b'}$ as an example. CoW from $Block_b$ to $Block_{b'}$ will bring in a new reference from $Block_{a'}$ to $Block_c$. In our design, this reference requires no explicit change is in $Block_c$. Thus if a crash occurs during the first part, no more recovery work will be needed. All the file modifications are completely lost. SIT and BIA is still consistent with the latest snapshot.

If a crash occurs during the second part of work, corresponding file system state will be detected on next mounting, which means interrupted work is able to be recommitted. We do the recovery by retrieving a newer checkpoint block address from latest snapshot and following the second part work like a usual snapshot taking.

3.3 Snapshot Removal

The main body of the snapshot removal is a tree traverse, and its critical path only covers the blocks that are firstly referenced by the removing snapshot. From the checkpoint block to data blocks, one of the following two operations is taken:

- Invalidate(*blk*): Mark *blk* invalid in BIA. Decrease the valid block counter of corresponding entry in Segment Entry Array.
- Postpone*(blk, version number)*: Change the start version number of *blk* to *version number*.

Algorithm 1 shows the per-block action in the removing traverse. *blk* is the passing-over block. *SV* is the start version number of *blk* in BIA. *CV*, *PV*, *NV* represent the version number of the removing snapshot, its previous and next snapshot in checkpoint list respectively.

The most difficult work in Algorithm 1 is to determine whether a block is further referenced. To deal with it, one more traverse is simultaneously taken on the next snapshot from its chekpoint block in the checkpoint list. This *twinned traverse* imitates every iteration of the removing traverse thus causes negligible performance overhead. If an iteration is not able to be imitated, then the twinned traverse will always stays in NULL. If the twinned traverse and removing traverse were passing on the same block, then this block is further referenced.

Algorithm 1 Per-block procedure in removing traverse

```
1: if $blk$ is not further referenced then
2:     if $SV > PV$ and $SV$ is invalid then
3:         Invalidate($blk$)
4:         iterate children
5:     end if
6: else $\{SV = CV\}$
7:     Postpone($blk, NV$)
8: end if
```

This twinned traverse can only find out shared blocks between successive snapshots. Suppose we're removing $Snapshot_1$ in Figure 5(a). $Block_g$ is referenced by $Snapshot_3$. A twinned traverse on $Snapshot_2$ cannot determine whether this block is referenced by $Snapshot_3$. To avoid such problem, deeper iteration is only allowed for invalidation(Line 4). Traverse will backtrack after postponing a block, leaving its children unchanged. This policy also guarantees the same time-efficiency with hierarchical reference counting [14].

BIA entries of these unchanged blocks will become *erroneous*, which means the start version number identifies a invalid snapshot. Those errors can be corrected by cleaning process in Garbage Collection (detailed in Section 3.4). If not, Line 2 handles this problem with the help of a validity checking from Checkpoint Radix Tree. Related functionality and optimization of such checking is mentioned in Section 2.4.

A detailed scenario is illustrated in Figure 5. We assume that there is no cleaning process throughout the process. In Figure 5(b), $Snapshot_1$ is removed. Only $Block_e$ and $Block_c$ are postponed. $Block_f$ and $Block_g$ are not traversed and contain erroneous version numbers. In Figure 5(c), $Snapshot_2$ is removed. Since $Snapshot_1$ is removed, 1 becomes an invalid version number. Thus $Block_f$ is invalidated. Start version number of $Block_g$ is still 1.

The whole process of snapshot removal is 1) recording snapshot's version number in latest snapshot, changing system state, 2) removing traverse, 3) SIT journaling and updating in NVM, removing the snapshot from checkpoint list, recovering system state.

With this design, except from a check for current snapshot, crash recovery in snapshot removal does the same work of the above step 2 and 3.

3.4 Garbage Collection

By reclaiming invalid and scattered blocks, cleaning process in Garbage Collection(GC) gets empty segments for further allocation. Those emptied segments are called *Victims*. Detailed discussion like how to select a victim will not be mentioned in this paper. In this section, we focus on how to provide consistency guarantee in block migrations.

When moving a block from victim segment to newly allocated area, block address must be modified correspondingly in its parent blocks. Crashes occurred during the modification will cause obsolete references on already reclaimed blocks.

There are two solutions to this problem: keep reclaiming unmoved blocks (Redo) or restore all the modified parent blocks (Undo). We choose the latter one. Because HMFS supports online GC, i.e. blocks for stacking writes can be allocated concurrently with cleaning process. Redo a half-done cleaning process may make those file blocks become new garbage, which will cost more space than GC can reclaim.

Algorithm 2 Per-segment procedure in GC crash recovery

```
1: for each $blk \in victim\ segment$ do
2:   if $blk$ is valid then
3:     continue
4:   end if
5:   $src\_addr \Leftarrow blk\ address$
6:   $tar\_addre \Leftarrow address\ in\ first\ referencer$
7:   if $src\_addr = tar\_addr$ then
8:     return
9:   end if
10:  for subsequent $snapshot \in checkpoint\ list$ do
11:    $ref\_addr \Leftarrow address\ in\ snapshot$
12:    if $ref\_addr = src\_addr$ then
13:      return
14:    else $\{ref\_addr \neq tar\_addr\}$
15:      continue
16:    end if
17:    $ref\_addr \Leftarrow src\_addr$
18:  end for
19: end for
```

Algorithm 2 shows the recovery process for GC undo solution. The main body is a sequential scan of every block in victim segment. To check whether a snapshot is referencing passing-over block, we use a inductive approach. The *source address(src_addr)* can be calculated by victim segment number and block offset. With start verion number and owner information in BIA, we can get a block address from its first referencer. If this address is equal to source address, then GC recovery is done. Otherwise it's the *target address (tar_addr)* of block migration in crashed cleaning process. If the corresponding block address in parent block of the subsequent snapshot*(ref_addr)* is equal to source address, then GC recovery is done. Otherwise, if it is identical to the target address, then current snapshot is not referencing this block, recovery process will move to the next block in victim segment.

In Line 11, an erroneous version number may be used for fetching *(ref_addr)*. When meeting such erroneous number, GC recovery will replace it with the version number of next valid snapshot in checkpoint list.

With crash recovery, file system metadata structures (BIA and SIT) are always consistent with the latest snapshot. With CoW, file writes will not corrupt existing data. Also, NAT of every snapshot matches its node blocks, which means version metadata matches referred data. Thus our mechanism provides high reliability at *version consistency* [4] level.

4. EVALUATION

In this section, we evaluate how our mechanism functions in practice. The whole experiment covers three typical file systems: one NVM-based in-memory file system, PMFS; one legacy snapshot file system, BTRFS; and our prototype file system, HMFS. We use the experimental result of BTRFS as a baseline for comparison.

Our test machine is a commodity server with 64 Intel Xeon 2GHz Processors. It consists of 512GB DRAM, out of which we configured 128GB as ramdisk for BTRFS and another 128GB as simulated NVM for PMFS and HMFS.

We use Filebench [1] benchmark suite to emulate following workloads. All the experiments are conducted on CentOS 6.6 using Linux Kernel 3.11. First we examine the bandwidth of single file I/O in random and sequential access. Then we compare the real performance of those file systems under three common workloads. Lastly, we measure the latency of system *sync* or snapshot taking and removal in corresponding workloads. In the evaluation result, our mechanism provides a performance comparable the NVM-based in-memory file system, demonstrating that higher reliability incurs minor cost. More importantly, comparing to BTRFS, our mechanism has effectively reduced the latency of snapshot taking and removal by 95% and 60% respectively.

4.1 File I/O

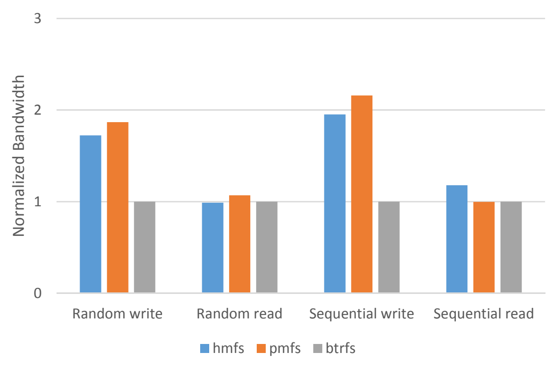
Figure 6(a) shows the benchmark result of read and write bandwidth in a single file. File size is set to 5GB. 8KB for random access and 1MB respectively for sequential access in each I/O operation.

We can see that PMFS and HMFS significantly improved the bandwidth of random and sequential writes, roughly as 2.0 and 1.8 times as the BTRFS's. This improvement is brought by the elimination of DRAM buffering. At the same time, HMFS is about 8 percent slower than PMFS. This drop in performance is mainly due to doubled allocation for a free blocks (one for block and one for NID). Since there is only one file and it is empty at first, NAT entries are all set to invalid in NVM. Locality of on-demand growth in NAT Radix Tree cannot function effectively. Also from the view of system reliability, PMFS guarantees no data consistency and requires no CoW for existing file data blocks, which make it perform better in random writes.

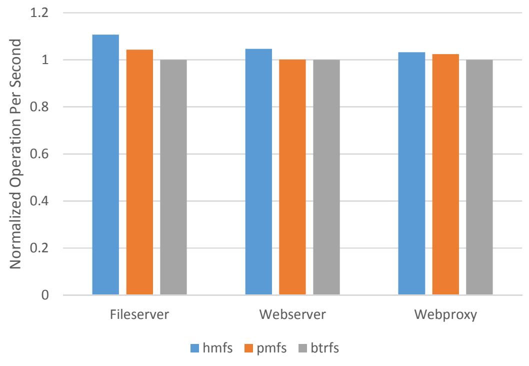
Since DRAM buffering imposed much lower performance overhead in reading than writing, read bandwidth in NVM-based in-memory file systems does not improved effectively like write bandwidth. We can see that HMFS keeps pace with BTRFS and PMFS in random read. Benefited from the locality of NAT Radix Tree, sequential read in HMFS is about 20 percent faster than PMFS.

4.2 Common Workloads

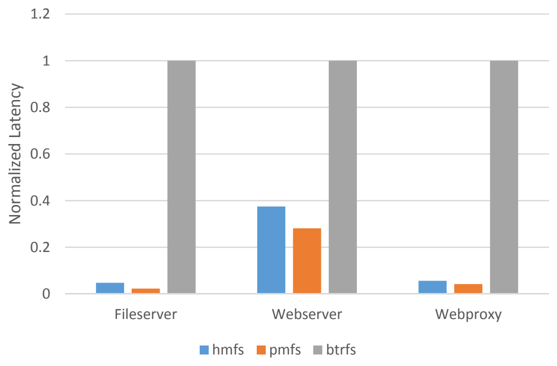
Figure 6(b) presents results from the evaluation of three multi-threaded application workloads from Filebench suite. *Fileserver* emulates a file server on the home directory with multiple threads. Each thread consists of file operations like creates, deletes, appends, reads, and writes. *Webserver*



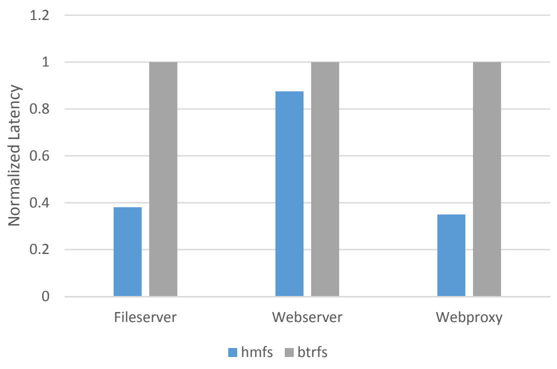
(a) Single file I/O bandwidth



(b) Performance under common workloads



(c) Latency of flushing



(d) Latency of snapshot removal

Figure 6: Evaluation Result

emulates a webserver with activity characterized by a read-write ratio of 10:1. It is a read-intensive workload which involves several lookups followed by whole file reads and appends. *Webproxy* emulates a simple web proxy server hosting directories with depth no more than 1. It exercises a mix of operations simultaneously from a large number of threads, combined with appending a simulated proxy log file. For all three workloads, we use the default configuration in Filebench.

Under those workloads, there is no significant improvement of performance between in-memory file systems and BTRFS. But it's worth noticing that HMFS even outperformance PMFS because of better concurrency between multiple operations. For metadata-intensive workloads like fileserver, HMFS's per-second operations exceed PMFS's for about 7% when data and metadata operation is included at a ratio of 1:1. This improvement is also caused by the unified management of file nodes in our mechanism. For read-intensive workloads, locality in volatile structures once again makes HMFS outperform other systems with a small margin.

4.3 Latency of Snapshot Functionalities

Figure 6(c) shows the flush latency of corresponding common workloads in Section 4.2. For HMFS and BTRFS, we measure the latency by taking a snapshot of a volume. For PMFS, we do a explicit *sync* and record elapsed time. Figure 6(d) shows the normalized latency of removing the created snapshots in HMFS and BTRFS.

As we can see from the pictures, except read-intensive workload like webserver, latency of snapshot related routines is reduced effectively in our mechanism. Snapshot taking is 20 times quicker than BTRFS and is within the same order of magnitude of *snyc* in PMFS. Also, snapshot removal is about 1.5 time quicker than BTRFS under fileserver and webproxy workloads.

5. RELATED WORK

Flash Friendly File System (F2FS) [8] is designed for better performance on modern flash storage devices. Being widely available, it's the first file system designed from scratch to optimize performance and lifetime of flash devices with a generic block interface. It contributes a cost-effective index structure, node address table (NAT), to attack 'Wander Tree' problem [3]. We further exploit NAT in our mechanism to construct a layered abstraction of storage space for a simplified and efficient consistency maintenance.

B-Tree File System (BTRFS) [16] is a renowned snapshot file system which combines ideas from ReiserFS [13] and CoW friendly b-trees suggested by O. Rodeh [14]. Growing more mature and stable, BTRFS is on its way of becoming the default Linux file system [15]. To deal with the problem of block sharing, its design of hierarchical reference counting is the state-of-the-art mechanism in snapshot file systems. By postponing changes in indirectly referenced blocks, it's very efficient but still has a side-effect, Update Storm [6]. Even when comparing to it's refined version, *RefCount Log* [15], our mechanism can provide the same space and time efficiency with less complex interaction with storage device in journaling and crash recovery.

BPFS [5] presents a NVM-based file system and a hardware architecture that enforces the required atomicity and ordering guarantees. It uses short-circuit shadow paging technique to provide consistency and improved performance at the same time. But for CoW and ordered updating, modifications to the allocation structures are scattered and require several commit writes to be executed. Merging those writes and file writes will cause serious write amplification. Thus it does not allow allocation structures to be stored in NVM.

Persistent Memory File System (PMFS) [12] takes advantages of some hardware features like the processor's paging and memory ordering to optimize its consistency guarantees (fine-grained logging) and memory-mapped I/O (transparent large page support). Its implementation of eXecute In Place (XIP) interface efficiently avoids the block device layer and page buffering. These optimization offers a considerable improvement of speed. However, because the huge journal overhead of double copying file data is unacceptable, PMFS allows the existence of corrupted file content after a unexpected system crash. Also, the huge overhead of ordering and journaling per-block data structures at runtime is also unacceptable. Hence, a free block list which is derived from files is kept in the volatile memory. To amortize the latency of initializing this list, PMFS stores the allocator structures in a reserved file on a clean unmount.

Three different levels of consistency concerning system crash is introduced in [4]: metadata consistency, data consistency and version consistency. PMFS guarantees no data consistency, thus it only provides metadata consistency. Both our and BPFS's mechanism guarantee version consistency. However, to ensure the ACID properties for transactions [10], BPFS requires immediate reflection to persistent memory in each file system call, which will impose a huge performance penalty of write amplification especially when file writes are small but scattered.

6. CONCLUSIONS

In this paper, we present a consistency mechanism for in-memory file systems. It's explicitly designed for NVM included memory architecture. This mechanism provides high reliability at version consistency level and powerful functionalities of snapshots. Evaluation results show that the extra cost of this mechanism is reasonable enough to provide performance equivalent to a NVM-based in-memory file system, and the latency of snapshot functionalities is much smaller than a state-of-the-art snapshot file system.

7. ACKNOWLEDGEMENTS

We thank Yanyan Shen, for her meticulous guidance and help. This work is supported by National High Technology Research and Development Program of China (No.2015AA015303) and the National Natural Science Foundation of China (No.61472241).

8. REFERENCES

[1] Filebench. http://sourceforge.net/apps/mediawiki/filebench.

[2] Hmfs source code. https://github.com/timemath/hmfs.

[3] A. B. Bityutskiy. Jffs3 design issues, 2005.

[4] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *FAST*, page 9, 2012.

[5] 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*, pages 133–146. ACM, 2009.

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

[7] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an nfs file server appliance. In *USENIX winter*, volume 94, 1994.

[8] C. Lee, D. Sim, J. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[9] Micron. 3d xpoint technology. https://www.micron.com/about/emerging-technologies/3d-xpoint-technology.

[10] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[11] M. K. Qureshi, V. Srinivasan, and J. a. Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24, 2009.

[12] D. S. Rao, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 15:1–15:15, 2014.

[13] H. Reiser. Reiserfs. http://en.wikipedia.org/wiki/ReiserFS.

[14] O. Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)*, 3(4):2, 2008.

[15] O. Rodeh. Deferred reference counters for copy-on-write b-trees. Technical report, Tech. rep. rj10464, IBM, 2010.

[16] O. Rodeh, J. Bacik, and C. Mason. BTRFS: the linux b-tree filesystem. *TOS*, 9(3):9, 2013.

[17] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Jha. A logic of file systems. In *FAST*, volume 5, pages 1–1, 2005.

[18] S. Vongehr. The Missing Memristor: Novel Nanotechnology or rather new Case Study for the Philosophy and Sociology of Science? *Advanced Science Letters*, 17(1):285–290, 2012.

[19] X. Wu, S. Qiu, and A. L. N. Reddy. SCMFS: A file system for storage class memory and its extensions. *TOS*, 9(3):7, 2013.