ReoFS: A Read-Efficient and Write-Optimized File System for Persistent Memory

Yan Yan*, Kaixin Huang*, Shengan Zheng†, Dongliang Xue* and Linpeng Huang*
* *Department of Computer Science and Engineering*
Shanghai Jiaotong University, Shanghai, China
{*118033910141, Kaixinhuang, xuedongliang010, lphuang*}*@sjtu.edu.cn*
† *Department of Computer Science and Engineering*
Tsinghua University, Beijing, China
{*venero*}*@tsinghua.edu.cn*

2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS) | 978-1-7281-8558-3/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICECCS51672.2020.00028

***Abstract*—In this paper, we present ReoFS, a read-efficient and write-optimized PM-aware (Persistent Memory-Aware) file system that provides strong consistency guarantee. Different from traditional journaling techniques, ReoFS optimizes write requests on the write I/O path and allows concurrent reads to be served opportunistically. To reduce the write overhead for metadata-related operations, we devise a backup buffering mechanism, which keeps a replica of metadata in the backup buffer and supports fast in-place update. To minimize the write overhead for file data modifications, we design a fine-grained short logging mechanism to migrate writes to persistent log blocks and make new writes visible immediately after the log is persisted. With these two designs, ReoFS removes the double write overhead off the critical path of request execution for both metadata and data operations. We conduct extensive experiments on Intel Optane DC PM platform and the results show that ReoFS outperforms existing PM-aware file systems by 1.3x-4x.**

***Keywords*-persistent memory, file system, data persistence, data consistency**

I. Introduction

Emerging persistent memory (PM) such as PCM, STT-RAM and 3D XPoint [1]–[3] is a new class of memory that can provide byte-addressability, non-volatility, large capacity and DRAM-comparable access latency. In April 2019, Intel announced the first commercially-available PM product, Intel Optane DC Persistent Memory [4]. PM can be placed on the memory bus and accessed like DRAM using processor load and store instructions [5]. With its low latency and non-volatility, PM has been attractive for building high-performance persistent storage systems, such as KV-Stores [6] [7], DBMSs [8] and file systems [9]–[12]. This paper focuses on designing PM-aware file system.

Traditional file systems introduce large overhead for read and write operations because of the data copy between the in-DRAM OS page cache and the durable storage (e.g., disk and SSD). To avoid such overhead, the system community has proposed some novel PM-aware file systems, such as BPFS [9], PMFS [10], NOVA [11] and HMVFS [13]. They redesign the file systems in kernel and remove the heavy OS page cache to directly copy data between the user buffer and the PM space.

For persistent memory-based systems, data consistency is a significant issue, because partial writes and reordering writes by modern processor to PM may cause inconsistent data upon system crash [14] [15]. This problem does not exist in traditional file systems since all the data in DRAM is volatile after crash. To maintain the correct persistence order, all current PM-aware file systems have to explicitly enforce memory writes (e.g., *clflush/clwb + mfence/sfence*) to persistent memory in a certain order [16]. This will add non-negligible performance overhead to file systems. Notably, such overhead is absolutely proportional to the number of write operations and forced persistence ordering instructions [17].

Existing PM-aware file systems have widely employed either write-ahead logging (WAL) [18]–[20] or copy-on-write (CoW) [21] techniques to obtain consistency [16]. Write-ahead logging is adopted by PMFS [10], HiNFS [12] and EXT4_DAX [22], which consists of redo logging and undo logging. In either case, a copy for original data or updated data has to be created in the critical path, leading to double write overhead. Other PM-aware file systems such as BPFS and NOVA mainly utilize CoW to achieve consistency. This technique in BPFS may incur a sequence of updates from the affected leaf node to the root node when a write request is committed. Even though CoW doesn't copy the original data or updated data, there still exists the write overhead for the entire data block, which is usually much larger than the modified data size [5]. Such performance issue is called write amplification.

We further compare existing mainstream PM-aware file systems and summarize their features in Table I. We observe that in addition to the overhead caused by double write and write amplification, there are other limitations in current file systems. First, some file systems [10] [12] [22] only support metadata consistency but lack the guarantee for data consistency. Notwithstanding the high performance they can achieve, they may be dangerous for certain deployed applications that require strong data consistency. Second, a few systems ignore the issue of limited write endurance of

978-1-7281-8558-3/20/$31.00 ©2020 IEEE
DOI 10.1109/ICECCS51672.2020.00028

Table I
COMPARISON OF DIFFERENT PM-AWARE FILE SYSTEMS

file system	metadata consistency	data consistency	double write-optimized	write amplification-optimized	endurance optimized
BPFS	√	√	√	×	×
PMFS	√	×	×	√	×
HiNFS	√	×	×	√	√
EXT4_DAX	√	×	×	×	×
NOVA	√	√	√	×	√
ReoFS	√	√	√	√	√

PM and make no optimization for that. Then the PM device supporting those persistent file systems may wear out easily for skewed write workloads.

Motivated by the limitations of existing PM-aware file systems summarized in table I, we develop ReoFS, a **r**ead-**e**fficient and write-**o**ptimized **f**ile **s**ystem for persistent memory. The main purpose of ReoFS is to provide strong consistency for both metadata and data, while minimizing extra performance overhead caused by write amplification, data flush (for persistence) and double write in the critical path. We decouple the write logic for file metadata and data to make best access performance for each. Concretely, we keep a backup buffer zone, in which each backup buffer is a replica of a metadata block. In this way, writes to metadata can be executed using in-place update, without copying old data or new data to the log area. For file data writes, we design per-file DRAM log nodes and persistent log blocks to absorb writes in fine-grained (i.e., cacheline) granularity to avoid write amplification. Furthermore, ReoFS removes the double write overhead for both metadata updates and data writes. In summary, this paper makes the following contributions.

• We analyze the limitations of existing PM-aware file systems, which are 1) the lack of data consistency, 2) double write overhead, 3) write amplification and 4) the risk of limited endurance. To overcome these limitations, we propose ReoFS, a read-efficient and write-optimized PM-aware file system.

• To guarantee efficient metadata consistency, we propose a backup buffering mechanism to support fast in-place metadata update for metadata access operation and file append operations. For data consistency, we also design a fine-grained short logging mechanism to enable flexible file modification (i.e., overwrite) operations, which reduces write amplifications while prolonging the lifetime of PM. Both mechanisms eliminate the double write overhead off the critical path.

• We implement ReoFS as a kernel module of Linux kernel 4.15.0 and evaluate it on Intel Optane DC Persistent Memory platform. The experimental results show that ReoFS outperforms the compared PM-aware file systems by 1.3x-4x in terms of overall throughput for various benchmarks.

The remaining of this paper is organized as follows. Section II presents the background and motivation of our work. Section III provides the design and detailed implementation of ReoFS. Performance evaluations are presented in section IV. Section V discusses related work and we conclude this paper in section VI.

II. BACKGROUND AND MOTIVATION

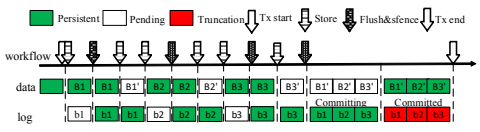
A. Persistent Memory

Persistent Memory (PM) is a new class of memory that supports byte-addressable access like DRAM and survives power outages. Attaching PM to the main memory bus provides a raw storage medium that can be orders of magnitude faster than modern persistent storage medium such as disk and SSD [23]. For PM-based systems, data consistency is a significant issue since partial writes can be persistent after system crash. To make matter worse, modern processors and caches may reorder writes to memory for better pipelining performance, which adds complexity to data consistency. That is, when a write request is issued, an atomic store only ensures 8-byte data to be written to volatile CPU cache and then flushed to the PM controller in arbitrary order [14] [15]. Fortunately, fence instructions such as *mfence/sfence* provided by Intel x86 can avoid memory reordering and force a store visible before subsequent operations are executed. Furthermore, *clflush/clwb* instructions are offered for programmers to explicitly flush a dirty cacheline to memory. With these two types of instructions, a data structure larger than 8 bytes can be stored and persisted into PM in correct order.

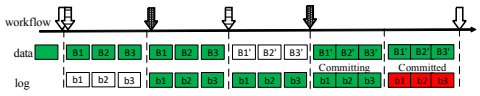
B. Review of Existing PM-aware File Systems

Many researchers have proposed novel file systems for PM [9]–[12] to remove the OS page cache. Apart from these PM-aware file systems, the common Linux ext4 file system also presents a new access model called Direct Access (DAX) to directly access PM without using a page cache.

It is significant for applications to recover to a consistent state after a system crash, which is called consistency. Consistency can be classified into two levels: metadata consistency (i.e., weak consistency, which only guarantees consistency for file metadata), full data consistency (i.e., strong consistency, which guarantees consistency for both file metadata and file data). Among the PM-aware file



(a) Undo logging workflow



(b) Redo logging workflow

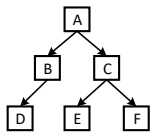
Figure 1. The workflow of two types of write-ahead logging

systems mentioned above, BPFS and NOVA provide strong consistency while the others pursue system performance but sacrifice data consistency. However, BPFS adopts *epoch* command and employs hardware primitives to enforce write ordering, which requires nontrivial modifications to system hardware [9]. NOVA is the only PM-aware file system designed in kernel that guarantees strong consistency without any hardware dependence.

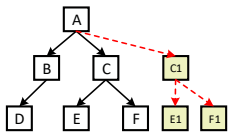
C. Drawbacks of Existing Consistency Mechanisms

Write-Ahead Logging (WAL) is adopted by PMFS, HiNFS and EXT4_DAX. It incorporates two basic logging paradigms: redo logging and undo logging. Fig. 1 shows the detailed workflow of undo logging and redo logging for updating three data blocks. Suppose that data block $B1$ is the original data to be modified. Block $b1$ represents the data copy in the log area and $B1^{'}$ stands for the new data. Fig. 1(a) gives the procedure of undo logging. For each update, undo logging has to create a data copy of original data before performing in-place update. Fig. 1(b) depicts the workflow of redo logging. Each update is intercepted and appended to the redo log area before making modifications to original data. Therefore, both write-ahead logging techniques have to create copies of original or new data in the critical path, leading to double write overhead. Furthermore, each update incurs extra cache line flushing and store ordering instructions. The overhead of persistence ordering for double copies is also prohibitively expensive for system performance.

Copy-on-Write (CoW) is used by many PM-aware file systems, such as BPFS and NOVA, to avoid the heavy double copy procedure. Fig. 2 uses a tree structure to describe the copy-on-write (CoW) technique. Fig. 2(a) is an original tree before modifications. In Fig. 2(b), a write request for node E and F is issued and three new nodes marked as $E1$, $F1$ and $C1$ are created to absorb the updates. We make two observations from the procedure of CoW employed in existing PM-aware file systems. First, with the tree-based data organization, CoW will make a cascade of updates from



(a) Original tree A



(b) Updating block node E and F

Figure 2. Copy-on-Write technique

the modified leaf nodes to the top root node. Therefore, like WAL, CoW substantially requires extra data copies in the critical path of execution. Second, when a write operation only modifies a few bytes within the whole data block, the new data block has to copy the old data from original data block, leading to higher write amplification overhead.

Apart from the lack of data consistency and performance overhead caused by double write and write amplification, limited write endurance is another issue in some existing PM-aware file systems. Since skewed workload (e.g., Facebook Memcached workload [24]) may frequently access and update the same data region, certain PM cells may be written many times and tend to wear out easily. Wear-leveling techniques should be developed for PM-aware file systems to relieve the risk of PM wear out. However, existing systems such as BPFS, PMFS and EXT4_DAX just ignore this issue.

III. REOFS

A. Design Goals

Strong consistency guarantee. ReoFS aims to guarantee both metadata consistency and data consistency.

Low latency. ReoFS should enable fast access to file data, with minimizing the write overhead and persistence ordering overhead in the critical path.

High throughput. ReoFS aims to achieve high concurrent throughput by improving file access concurrency.

Wear-leveling support. ReoFS is supposed to be wear-leveling-aware and automatically balance writes to relieve the risk of wear-out for skewed workloads.

Transparency and portability. ReoFS is developed as a kernel module in Linux system and does not require any hardware modification to modern hardwares.

B. Overview

The architecture of ReoFS is shown in Fig. 3. Similar to EXT4_DAX, ReoFS bypasses the generic block layer and copies data between user buffer and PM directly. ReoFS offers an in-kernel PM-aware architecture that services all file operations. Specifically, there are two categories of file operations in file systems: metadata operations (e.g., check file state, modify file size) and data operations (e.g., file overwrite and append). In addition to the metadata area that

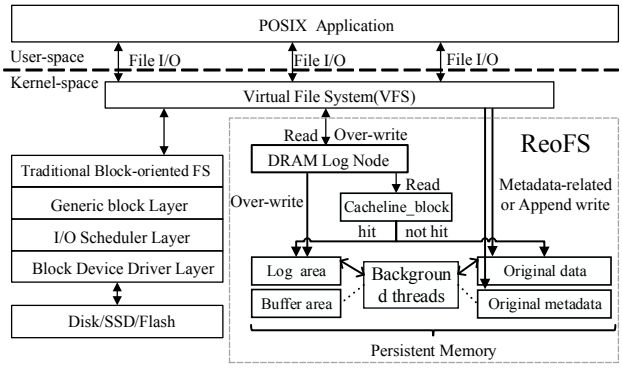


Figure 3. Architecture of ReoFS

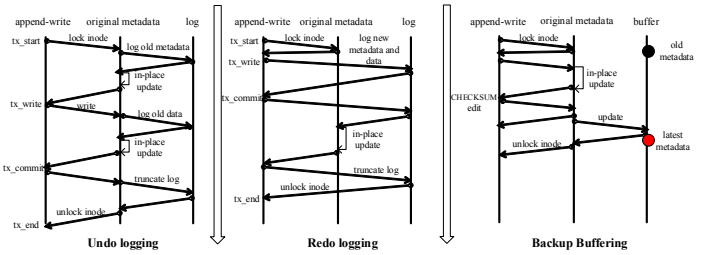


Figure 4. **Workflow comparison of different update techniques.** For viewing convenience, the workflow is used for append-write request. Notice that we only use the workflow to deal with append-write and metadata-related write for single inode.

stores file metadata blocks and data area that contains file data blocks, we also reserve a buffer area and log area in PM. The buffer area is used to speed up metadata operations and the log area is reserved for promoting the performance of data operations. In the meanwhile, they are responsible for metadata consistency and data consistency of ReoFS.

Since the metadata block and data block have different granularities, we design different consistency mechanisms for them to obtain the best performance of each: 1) *backup buffering* (BABU) is devised to guarantee metadata consistency and speed up metadata operations, 2) *fine-grained short logging* (FISOL) is proposed to guarantee data consistency and enable efficient data reads/writes. Both should be assisted with background threads, which are responsible for copying updated data from the original metadata block to the backup buffer and from the log block to original data block, respectively.

C. BABU: Backup Buffering

The main goal of *backup buffering* is to provide metadata consistency while minimizing the overhead caused by double write and persistence ordering. The key idea is to maintain an extra metadata copy in the buffer area. Upon the time a file is created, its metadata (i.e., inode block) is initiated and a corresponding backup buffer in the buffer area is allocated for it. The backup buffer always keeps a consistent old version of that metadata block. Any modification to metadata is performed with in-place update. Then, a checksum for the metadata is calculated and stored into the *CHECKSUM* field, which occupies the last 16 bits of the metadata block. The design of checksum is critical for the correctness of *backup buffering* since it can tell whether the metadata is integrated after crash. After that, a background thread will immediately persist the modified data and copy it to the corresponding backup buffer. As a result, the overhead of both persistence ordering and double write is eliminated in the critical path. The metadata write transaction can quickly return and then process the next transaction.

Fig. 4 compares write-ahead logging techniques with our proposed *backup buffering*. We observe that both undo and redo logging need to create a data copy for either old version or new version of metadata in the critical path. Conversely, *backup buffering* does not require to write to a data copy in the critical path. It even eliminates the overhead of copy creation and deletion. Therefore, ReoFS is able to achieve low-latency metadata access.

Notice that once a metadata write transaction in the front-end has completed, a background thread will take over the remaining work, copying the modified metadata

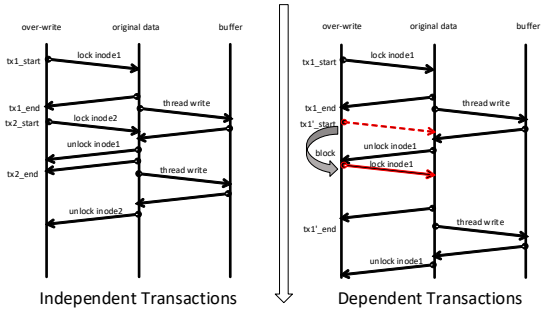


Figure 5. Dependent journaling and independent journaling

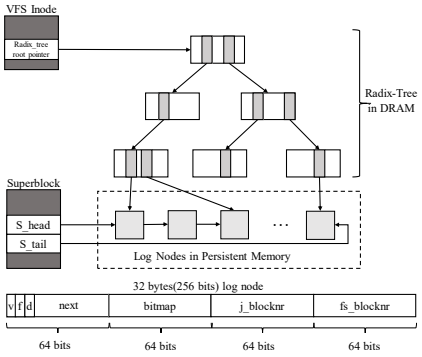


Figure 6. Block Node Index

to its backup buffer. Only when the updated metadata is persisted in the backup buffer, the metadata write transaction is considered as committed and all the modifications are visible to future reads.

Access collisions may occur when a new metadata write transaction starts after the last transaction ends but before the last transaction commits. We address this challenge with the design of a dependent list. A dependent list includes an inode set, in which all the metadata blocks are inconsistent with their corresponding backup buffers. It indicates that either in-place update or background copying is being performed for these metadata blocks. A dependent transaction is a metadata read/write that accesses an inode that belongs to the dependent list. If the new transaction does not overlap with the dependent list, it is called an independent transaction.

Fig. 5 illustrates how independent transactions and dependent transactions are processed. An independent transaction can be performed immediately whenever it issues a read or write request to a metadata block. However, a dependent transaction has to wait until the original metadata block is consistent with corresponding backup buffer (i.e., the relevant transactions are all committed).

D. FISOL: Fine-Grained Short Logging

The *fine-grained short logging* mechanism is designed to guarantee data consistency and support efficient access to file data. It mainly serves file overwrite operations. In ReoFS, each inode has its own data log. This contributes to high access concurrency across multiple files. We keep the log blocks in PM to record updated data and maintain an index structure in DRAM to enable fast lookup for latest data. The *fine-grained short logging* optimizes traditional redo logging scheme with two distinguishing techniques: fine-grained recognition and eager commit. The fine-grained recognition technique can decrease the write amplification overhead by only recording the modified cachelines of original data blocks. The eager commit technique allows the transaction to commit immediately after the modified data is persisted in the log block. We name this as short-path logging because there is no double write in the critical path. In this way, the update is visible to other transactions earlier and supports higher concurrency for reads and writes. The modified data in the log block will be copied back to original data block by a background thread in a lazy method. This also helps to coalesce the data updated to the same file and reduces the number of writes to original data blocks, which optimizes the endurance performance of PM. The illustrations for DRAM log node, fine-grained logging technique and eager commit technique are provided as follows.

DRAM Log Node. Each inode owns a radix tree in DRAM space to support fast indexing of valid log blocks in persistent memory. Radix tree is feasible and efficient for search, insert and delete operations [25]. The index structure of the DRAM Log Node is depicted in Fig. 6. The leaf nodes of radix-tree, however, are stored in PM for data consistency. ReoFS finds one log node by the logic file offset which is aligned with the data block size. The log node acts as the metadata of log blocks and each occupies 32 bytes. As illustrated in Fig. 6, the leftmost 3 flag bits are used to label the attributes of a log block. The *v* bit represents whether the matched log block is valid ('1') or invalid ('0'). The *f* bit is used to label whether the matched log block is occupied ('1') or free ('0'). The *d* bit tells whether the matched log block and the log node are deletable ('1') or non-deletable ('0'). The subsequent 61 bits are used as *next* field, which points to another log node if there exists one. The remaining bits are divided into three parts equally, including a 64-bit bitmap, 64-bit log block number and 64-bit original block number. The root of radix tree is stored in the kernel node of virtual file system and all the log nodes are linked as a log node list. A head pointer points to the first log node and a tail pointer points to the last node. Both of these two pointers are stored in the file system superblock.

Fine-grained Recognition. Both conventional buffer management for block-oriented file systems and PM-aware file systems are coarse-grained (e.g., 4KB block granularity

Table II
PLATFORM CONFIGURATIONS

CPU	Intel Xeon Gold 6240×2
DRAM	32GB×8
PM	Intel Optane DC Persistent Memory 128GB×4
OS	Ubuntu 18.04, Linux 4.13.0, Linux 4.15.0

Table III
FILE SYSTEMS FOR COMPARISON

EXT4_DAX	EXT4 with Direct Access (DAX).
EXT4_DBA	EXT4 with strong consistency.
PMFS_UNDO	PMFS with data consistency using undo logging.
NOVA	A PM-aware file system with copy-on-write technique.
ReoFS_0	It only has the *fine-grained short logging* mechanism in comparison with ReoFS.

[11]) for data modifications. However, such management scheme is not efficient in two respects. (1) a write operation may only modify a few bytes within the whole data block, but the new data block has to copy all the content from original data block to the new version and the need also arises to flush the data in the entire block to persistent memory. (2) When an unaligned write is issued, a fetching process will catch the persistent data to log block first and then the new data is written to the log block.

To ease the buffer management, we propose a fine-grained recognition technique with cacheline-level management. A 64-bit *bitmap* is used to record the states of all the cachelines in a data block. If the Mth bit is 1, the Mth 64 bytes need to be flushed into original data block. When a write operation modifies a few bytes, instead of the whole data block, ReoFS only needs to fetch the related cachelines. As such, the fine-grained recognition technique reduces many unnecessary copies and eases the write amplification overhead.

Eager Commit. *Fine-grained short logging* mechanism employs a background thread to defer the heavyweight cache flush and memory fence operations (e.g.,*clflush/clwb + mfence/sfence*) and eliminates any copy in the critical write I/O. Once the updated data is persistent in the log block, the write transaction can be regarded as committed and upper-layer applications have the chance to access the latest data version earlier. For instance, a read transaction for a file with committed logs can be processed in three steps: 1) locate the related log nodes; 2) find dirty cachelines from the cacheline bitmaps; 3) read latest data from both original data block and relevant dirty cachelines from valid log blocks. Therefore, eager commit can improve the concurrency of ReoFS, especially when multiple threads operate on the same file. Furthermore, the combination of background thread and cacheline bitmap is able to coalesce writes to the same block and cancel the writes that are later deleted, thus reducing the number of writes to original data block and improving the endurance of PM.

E. Buffering and Persistence

Buffer area. ReoFS uses B-tree data structure to store the metadata (i.e., inode) information of files.

Log node buffer. ReoFS adds a log node layer to modify the I/O execution path, which maintains a linked list to track the recently-overwritten data. When an overwrite transaction is performed, the *fine-grained short logging* will calculate the delta between the old log node and new log node. The delta and new data is written to a new log block while the raw log block remains unchanged.

Background threads. ReoFS creates the background kernel threads when the system is mounted. They are used to persist data and commit a transaction in the background. Typically, two kernel threads are utilized to complete these operations: one backup thread for copying data from original metadata block to the corresponding backup buffer and one log thread for copying data from the log block to original data block.

The backup thread will be waken up once the write request completes the in-place update and checksum calculation in the metadata block. As for the log thread, two cases may trigger the process of it. First, the time interval threshold, 5 seconds by default, is reached. Second, the number of free log nodes falls below a threshold value, 10% by default. When the log thread is waken up, it will deal with the log nodes from head to tail pointer in the superblock. For each log node, its flag bits determine the subsequent operations. For example, if the *d* flag is 1, the log node is useless because it is deleted or coalesced. If the *v* flag is 1, the log thread will write back the dirty cachelines to original data block via the memory interface (i.e.,memcpy_mcsafe()). After that, the corresponding log node and log block will be reclaimed for future write transactions.

F. Concurrency and Consistency

Concurrent read. A file in ReoFS can be accessed simultaneously by multiple processors with shared read lock. They will read the latest data from both log blocks and original data blocks.

Concurrent write. ReoFS uses mutex lock mechnism for concurrent read and write operations. If there is another transaction executed on this metadata/data block, we just defer the current transaction until the lock on this file is released by the concurrent pending transaction.

Crash recovery. When a system crash occurs during meta-related or append-write operation, ReoFS first calculates the checksum of the metadata block. If the checksum result matches the value in the *CHECKSUM* field, then current metadata block is the latest consistent version. ReoFS will overwrite the corresponding backup buffer in the buffer area. Otherwise, the metadata block should be

Table IV
BENCHMARKS AND DESCRIPTION

Benchmark	Read/Write Ratio	Request Size	Running Time	Description
Fileserver	$1/2$	$16KB$	$60s$	A file server to perform create, append, read, write and delete operations
Webserver	$10/1$	$16KB$	$60s$	A web server consists of file reads and log append operations.
Varmail	$1/1$	$16KB$	$60s$	A email server contains read and delete operations, create-append and read-append with sync flag.
Sequential write	$0/1$	$1KB \sim 4KB$	$60s$	Sequential write for a file data
Random write	$0/1$	$1KB \sim 4KB$	$60s$	Random write for a file data

overwritten by the corresponding backup buffer. In either case, the metadata block will enter a consistent state, which satisfies the requirement of metadata consistency.

When system crash occurs during overwrite operation, ReoFS first scans the journal. If an uncommitted transaction is found, an undo procedure should be performed: 1) copy the corresponding backup buffer to original metadata block, 2) mark the valid log nodes which point to the modified data blocks as deletable, and 3) for each deletable log node, traverse the linked list, find the log nodes that has the same *fs_blocknr* field and mark the last log node as valid. The situation may occur that one transaction is committed but the metadata block is not consistent with the backup version. For this case, a copy operation from metadata block to backup buffer is enough. Finally, a recovery process is performed to deal with all the log nodes stored in linked list.

Wear-leveling. In order to avoid localized writes to the same block, ReoFS defers the writeback of log blocks and keeps a list of log blocks in persistent memory. When different versions of the same block exists in log blocks and frequent write requests for that block is issued, we only write the new data and delta data to log block, therefore reducing the number of writes to the log block and original data block.

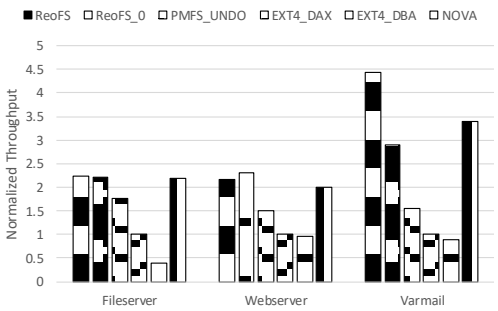


Figure 7. Overall performance

IV. PERFORMANCE EVALUATION

A. Experimental Setup

We conduct our evaluation experiments on Intel Optane DC Persistent Memory platform (OptanePM). The configurations of OptanePM are listed in Table II, in which Linux 4.13.0 is used for NOVA. All the experiments are developed on a separate persistent memory device and threads are pinned on local NUMA nodes. In all cases, we use *clwb* instead of *clflush* instruction for cacheline flush.

We list the file systems for comparison in Table III. EXT4_DAX and NOVA are designed to access the persistent memory directly. EXT4_DBA adopts journal mode in which both metadata and data are written to log area while NOVA takes copy-on-write technique. ReoFS is developed on the basis of PMFS. Since PMFS doesn't guarantee data consistency, we add an undo logging-based consistency mechanism in PMFS, for fairness of comparison. In addition, we also implement a comparative version, ReoFS_0 without *backup buffering* mechanism((i.e., metadata consistency is guaranteed by undo logging).

We use benchmarks [26] (filserver, webserver and varmail) to measure the performance of file systems. Table IV shows their characteristics and descriptions. For these benchmarks, the default configuration is used of 50 threads issuing requests to the fileserver.

B. Overall performance

In Fig. 7, normalized throughput from three multi-threaded benchmarks is presented.

Fileserver. Compared with EXT4_DAX, EXT4_DBA and NOVA, ReoFS gains 1.03× (for NOVA) to 5.7× (for EXT4_DBA) higher throughput. It mainly benefits from *backup buffering* and *fine-grained short logging* mechanism, which remove the double-copy overhead off the critical path and minimize the write amplification overhead. EXT4_DBA has the worst performance because of the penalty of double-write overhead and of generic block layer.

Webserver. Webserver is a read-dominated benchmark. Comparatively, ReoFS is 9% better on average than NOVA and 128% better than EXT4_DBA. EXT4_DAX has the same performance as the EXT4_DBA because of the DRAM

page cache. We can see that the performance of ReoFS_0 outperforms ReoFS since the log append operations are in the critical path as well as an additional overhead for *CHECKSUM*.

Varmail. A large portion of operations is synchronous in Varmail, which involve both reads and writes. ReoFS outperforms the other file systems by 1.3× to 4× as all the writes in this case are append operations, which will be written to file directly without the double-write overhead. It is also observed that the performance of ReoFS is better than ReoFS_0, because Varmail consists of many metadata operations, *backup buffering* in ReoFS can eliminate any copy in the critical path and improve the performance. Meanwhile, the performance of ReoFS_0 is also lower than NOVA because NOVA uses one log for each inode to maintain file's metadata, avoiding the duplicate writes overhead. Therefore, ReoFS_0 is more suitable for read-intensive operations. Furthermore, users can decide to mount ReoFS_0 or ReoFS for best performance.

C. Scalability and Sensitivity

Scalability to the number of threads. We vary the number of threads to compare the throughput of ReoFS with the mainstream file systems. The results are shown in Fig. 8.

From Fig. 8(a) and Fig. 8(c), we observe that ReoFS outperforms the other systems for fileserver and varmail benchmarks. This is because 1) ReoFS buffers the backup version and log blocks to enable efficient data reads/writes. 2) The data hit radio will increase when the number of threads increases from 1 to 9. The throughput of EXT4_DBA is stable at approximately 500MB/s because of the constraints incurred by the double write overhead and the generic block layer. However, for the webserver benchmark (shown in Fig. 8(b)), the performance of ReoFS is a bit lower than NOVA for throughput. The reason is that webserver is a read-dominated benchmark. ReoFS accesses the latest data from both original data block and log blocks while NOVA designs a powerful and comprehensive index structure to quickly perform search operations [11].

Sensitivity to the I/O size. The write I/O path supports high concurrency for mixed read/write operations. Therefore, in this part, we ignore the discussion for the performance of read-only operations. The experimental results for sequential and random writes are given in Fig. 8(d) and Fig. 8(e). For brevity, we only keep ReoFS_0, EXT4_DBA with write-ahead logging and NOVA with copy-on-write technique.

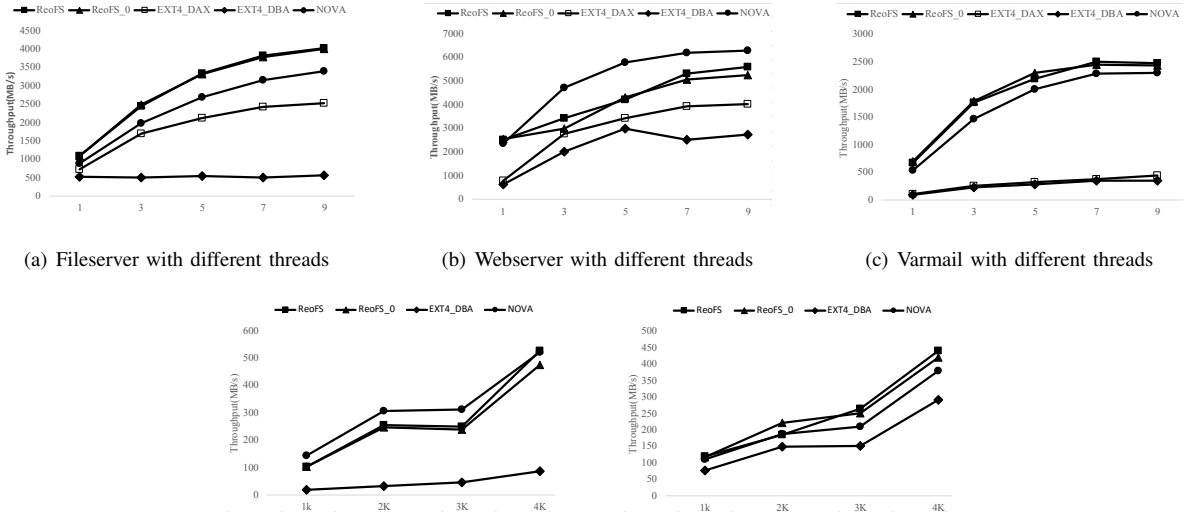
Fig. 8(d) shows that the throughput of NOVA is better than ReoFS when the size is less than 4KB. This occurs because 1) the fine-grained structure for log copying in ReoFS will degenerate to write the entire page for sequential write while only a few bytes are modified, 2) ReoFS has almost the same performance as ReoF_0, which means the double write for backup buffer is in the critical path. All cases show that the throughput of ReoFS exceeds EXT4_DBA because of minimal data copying and write I/O by bypassing the generic block layer.

In Fig. 8(e), the throughput of ReoFS outperforms NOVA by up to 25%. When the I/O size is 2KB, we find that the performance of ReoFS_0 is better than ReoFS because of the higher hit ratio and the constraint of backup thread. The results demonstrate that ReoFS is more suitable for handling random writes.

V. RELATED WORK

PM-aware File Systems. Many researchers have proposed different in-kernel file systems for persistent memory. BPFS [9] uses a combination of shadow paging technique and 8-byte atomic in-place update to ensure consistent update while requiring a hardware approach (i.e., epoch) to force correct persistence ordering. PMFS [10] is a lightweight POSIX-based file system built to bypass the OS page cache and access persistent memory directly. HiNFS [12] buffers the lazy-persistent writes in DRAM to hide the long write latency. EXT4_DAX [22] extends EXT4 with a new access model called Direct Access(DAX) to directly read from or write to persistent memory. However, PMFS, HiNFS and EXT4_DAX sacrifice data consistency and ignore the double-write overhead in the critical path. NOVA [11] is a log-structured file system that guarantees both metadata and data consistency with copy-on-write technique. However, it performs page-level copy-on-write and incurs many wasteful copies when only a few bytes are modified. In constrast, ReoFS provides strong consistency and minimizes the extra performance overhead caused by write amplification and double write in the critical path. Moreover, ReoFS uses *fine-grained short logging* mechanism to coalesce multiple writes to the same data block and improves the endurance of PM.

Buffering and Caching. Buffering and caching are widely employed to improve the performance of storage systems [27]–[29]. For flash-based storage systems, a flash-aware SSD cache is introduced to improve the efficiency of write and erase operations. H. Jo et al. [27] use a flash-aware write buffer and optimizes the traditional LRU policy to select a victim based on its page utilization. S. Kang et al. [28] emulate a block device using flash memory and offers *Cold and Largest Cluster Policy* to accommodate both temporal locality and cluster size. However, these flash-based cachings manage the memory space at the page granularity and match the characteristics of flash (e.g., slower erase operation). Therefore, these write buffer managements are not suitable for authentic persistent memory. E. Lee and H. Bahn [29] present a new buffer cache management scheme for PCM-based storage systems. However, this cache technique still maintains OS page cache and assumes that persistent memory is attached to the I/O bus. In contrast, ReoFS bypasses the OS page cache and employs a DRAM log node buffer layer



(a) Fileserver with different threads (b) Webserver with different threads (c) Varmail with different threads

(d) Sequential write (e) Random write

Figure 8. Scalability and Sensitivity

while persistent memory is placed on the memory bus. Furthermore, with the high performance of emerging persistent memory technologies, system designers should care about the write and copy overhead for persistent memory buffering. Taking advantages of PM's byte-addressability and non-volatility, we devise *backup buffering* and *fine-grained short logging* mechanisms to eliminate the double write overhead off the critical path.

VI. Conclusion

We develop a novel PM-aware file system named ReoFS, which can achieve efficient read and lightweight write while guaranteeing strong data consistency. We propose a backup buffering mechanism for metadata writes to support fast in-place updates and opportunistic concurrent reads. We design a fine-grained short logging mechanism for data writes to migrate writes to persistent log blocks and enable eager commit. Both of these two mechanisms remove the double write overhead off the critical path, which tend to achieve low latency and high throughput. The experimental results demonstrate that ReoFS gains better performance than the compared PM-aware file systems.

Acknowledgements

This work is supported by the National Key Research and Development Program of China (No.2018YFB1003302), the China Scholarship Council (No. 201906230180) and SJTU-Huawei Innovation Research Lab Funding.

References

[1] A. M. Caulfield, J. Coburn, T. I. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*. IEEE, 2010, pp. 1–11. [Online]. Available: https://doi.org/10.1109/SC.2010.56

[2] T. Kawahara, "Scalable spin-transfer torque RAM technology for normally-off computing," *IEEE Des. Test Comput.*, vol. 28, no. 1, pp. 52–63, 2011. [Online]. Available: https://doi.org/10.1109/MDT.2010.97

[3] "Intel and micron produce breakthrough memory technology," https://www.i-micronews.com/intel-and-micron-produce-breakthrough-memory-technology/.

[4] "Intel optane dc persistent memory," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[5] C. Chen, J. Yang, Q. Wei, C. Wang, and M. Xue, "Fine-grained metadata journaling on NVM," in *32nd Symposium on Mass Storage Systems and Technologies, MSST 2016, Santa Clara, CA, USA, May 2-6, 2016*. IEEE Computer Society, 2016, pp. 1–13. [Online]. Available: https://doi.org/10.1109/MSST.2016.7897077

[6] D. Hwang, W. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*, N. Agrawal and R. Rangaswami, Eds. USENIX Association, 2018, pp. 187–200. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/hwang

[7] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018, pp. 461–476. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/zuo

[8] R. Fang, H. Hsiao, B. He, C. Mohan, and Y. Wang, "High performance database logging using storage class memory," in *Proceedings of the 27th International Conference on*

Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany, S. Abiteboul, K. Böhm, C. Koch, and K. Tan, Eds. IEEE Computer Society, 2011, pp. 1221–1231. [Online]. Available: https://doi.org/10.1109/ICDE.2011.5767918

[9] 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 *Acm Symposium on Operating Systems Principles*, 2009.

[10] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *European Conference on Computer Systems*, 2014.

[11] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, A. D. Brown and F. I. Popovici, Eds. USENIX Association, 2016, pp. 323–338. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[12] Y. Chen, J. Shu, J. Ou, and Y. Lu, "Hinfs: A persistent memory file system with both buffering and direct-access," *TOS*, vol. 14, no. 1, pp. 4:1–4:30, 2018. [Online]. Available: https://doi.org/10.1145/3204454

[13] S. Zheng, H. Liu, L. Huang, Y. Shen, and Y. Zhu, "HMVFS: A versioning file system on DRAM/NVM hybrid memory," *J. Parallel Distributed Comput.*, vol. 120, pp. 355–368, 2018. [Online]. Available: https://doi.org/10.1016/j.jpdc.2017.10.022

[14] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering," in *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, W. J. Bolosky and J. Flinn, Eds. USENIX Association, 2012, p. 9. [Online]. Available: https://www.usenix.org/conference/fast12/consistency-without-ordering

[15] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 228–243. [Online]. Available: https://doi.org/10.1145/2517349.2522726

[16] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: lightweight persistent memory," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, R. Gupta and T. C. Mowry, Eds. ACM, 2011, pp. 91–104. [Online]. Available: https://doi.org/10.1145/1950365.1950379

[17] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*. IEEE Computer Society, 2015, pp. 1–10. [Online]. Available: https://doi.org/10.1109/MSST.2015.7208275

[18] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, R. Gupta and T. C. Mowry, Eds. ACM, 2011, pp. 105–118. [Online]. Available: https://doi.org/10.1145/1950365.1950380

[19] A. Kolli, S. Pelley, A. G. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, T. Conte and Y. Zhou, Eds. ACM, 2016, pp. 399–411. [Online]. Available: https://doi.org/10.1145/2872362.2872381

[20] E. Giles, K. Doshi, and P. J. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*. IEEE Computer Society, 2015, pp. 1–14. [Online]. Available: https://doi.org/10.1109/MSST.2015.7208276

[21] J. Gray, P. R. McJones, M. W. Blasgen, B. G. Lindsay, R. A. Lorie, T. G. Price, G. R. Putzolu, and I. L. Traiger, "The recovery manager of the system R database manager," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 223–243, 1981. [Online]. Available: https://doi.org/10.1145/356842.356847

[22] M. Wilcox, "Dax: Page cache bypass for filesystems on memory storage," https://lwn.net/Articles/618064/.

[23] Y. Wang, T. Wang, D. Liu, Z. Shao, and J. Xue, "Fine grained, direct access file system support for storage class memory," *J. Syst. Archit.*, vol. 72, pp. 80–92, 2017. [Online]. Available: https://doi.org/10.1016/j.sysarc.2016.07.003

[24] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, N. Feamster and J. C. Mogul, Eds. USENIX Association, 2013, pp. 385–398. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[25] V. Alvarez, S. Richter, X. Chen, and J. Dittrich, "A comparison of adaptive radix trees and hash tables," in *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, Eds. IEEE Computer Society, 2015, pp. 1227–1238. [Online]. Available: https://doi.org/10.1109/ICDE.2015.7113370

[26] "Filebench file system benchmark," https://sourceforge.net/projects/filebench/.

[27] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee, "FAB: flash-aware buffer management policy for portable media players," *IEEE Trans. Consumer Electronics*, vol. 52, no. 2, pp. 485–493, 2006. [Online]. Available: https://doi.org/10.1109/TCE.2006.1649669

[28] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha, "Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices," *IEEE Trans. Computers*, vol. 58, no. 6, pp. 744–758, 2009. [Online]. Available: https://doi.org/10.1109/TC.2008.224

[29] E. Lee and H. Bahn, "Caching strategies for high-performance storage media," *TOS*, vol. 10, no. 3, pp. 11:1–11:22, 2014. [Online]. Available: https://doi.org/10.1145/2633691