Enabling Reliable Memory-Mapped I/O With Auto-Snapshot for Persistent Memory Systems

Bo Ding, Wei Tong, Yu Hua, *Senior Member, IEEE*, Zhangyu Chen, Xueliang Wei, and Dan Feng, *Fellow, IEEE*

***Abstract*—Persistent memory (PM) is promising to be the next-generation storage device with better I/O performance. Since the traditional I/O path is too lengthy to drive PM featuring low latency and high bandwidth, prior works proposed memory-mapped I/O (MMIO) to shorten the I/O path to PM. However, native MMIO directly maps files into the user address space, which puts files at risk of being corrupted by scribbles and non-atomic I/O interfaces, causing serious reliability issues. To address these issues, we propose RMMIO, an efficient user-space library that provides reliable MMIO for PM systems. RMMIO provides atomic I/O interfaces and lightweight snapshots to ensure the reliability of MMIO. Compared with existing schemes, RMMIO mitigates additional writes and extra software overheads caused by reliability guarantees, thus achieving MMIO-like performance. In addition, we also propose an automatic snapshot with efficient memory management for RMMIO to minimize data loss incurred by reliability issues. The experimental results of microbenchmarks show that RMMIO achieves 8.49x and 2.31x higher throughput than ext4-DAX and the state-of-the-art MMIO-based scheme, respectively, while ensuring data reliability. The real-world application accelerated by RMMIO achieves at most 7.06x higher throughput than that of ext4-DAX.**

***Index Terms*—Persistent memory, memory-mapped I/O, PM-aware file system, data reliability.**

I. Introduction

NON-VOLATILE memory (NVM) technologies, such as Resistive RAM (ReRAM) [35], 3D XPoint memory [21], and Phase Change Memory (PCM) [31], achieve the advantages of both DRAM (e.g., low latency and high bandwidth) and disk (e.g., persistency), enabling the durability of data in memory space. Persistent memory (PM) powered by NVMs reduces hardware I/O overhead while suffering from the long I/O path of traditional file systems. To simplify the software I/O stack, recent PM-aware file systems, e.g., PMFS [15], NOVA [40], and ext4-DAX [37], leverage the DAX (Direct Access) [1] technology to remove the additional data copy between PM and the page cache in DRAM. Thus PM-aware file systems can directly access the data in PM and enable in-place updates to PM files. However, DAX-enabled file systems still suffer from the complex indexing structure and lengthy kernel I/O path. To further simplify the I/O path to PM, SplitFS [23] and Libnvmmio [12] propose user-space I/O operations that map PM files into user address space and access data via load/store instructions, termed DAX-style memory-mapped I/O (MMIO). MMIO speeds up I/O operations but keeps the mapped data out of the kernel's protections. Specifically, existing MMIO-based schemes face two main reliability problems: data integrity and consistency.

Data Integrity. While employing MMIO, the file mapped into user address space could be easily overwritten with arbitrary data, called *scribbles*, due to bug-prone user-space software and unexpected hardware errors [25], [41]. In PM systems, scribbles could be more dangerous than those in DRAM systems because of recently discovered bugs in PM programming [27], [28]. In addition, scribbles in PM will permanently corrupt data and exist even after the system reboots. Although previous works, e.g., Btrfs [32] and ZFS [10], provide complete mechanisms to protect file data from scribbles, they have not considered protecting the mapped data. Moreover, the NVDIMM driver that manages PM devices also cannot provide any RAS (recovery, availability, and serviceability) guarantee for the PM data [4]. Snapshots and replicas are widely employed by existing highly reliable file systems [10], [32], [41], to protect data from scribbles. However, snapshots and replicas cause huge write amplification, compromising the performance of MMIO.

Data Consistency. PM does not provide block-level atomicity but only guarantees the atomicity for 8-byte write: any write larger than 8 bytes may be partially lost in the event of a system crash, posing significant challenges to the crash consistency (we refer to consistency) of data in PM [40]. Existing PM file systems provide atomic I/O interfaces to guarantee data consistency for PM. However, MMIO bypasses all I/O stacks of PM file systems and escapes from data consistency protections. To guarantee data consistency for MMIO, SplitFS [23] implements atomic I/O operations by combining WAL (i.e. Write-Ahead-Log) and copy-on-write. Due to the over 50% performance decline, such a strict consistency guarantee is only available in the strict mode of SplitFS. Libnvmmio [12]

Manuscript received 21 December 2022; revised 3 March 2024; accepted 18 May 2024. Date of publication 19 June 2024; date of current version 9 August 2024. This work was supported by the National Natural Science Foundation of China under Grant 61832007, Grant 61821003, and Grant 62172178. Recommended for acceptance by M. Kandemir. *(Corresponding author: Wei Tong.)*

The authors are with Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: boding@hust.edu.cn; tongwei@hust.edu.cn; csyhua@hust.edu.cn; chenzy@hust.edu.cn; xueliang_wei@hust.edu.cn; dfeng@hust.edu.cn).

Digital Object Identifier 10.1109/TC.2024.3416683

proposes fine-grained logging to accelerate the concurrent I/O but there is still a performance penalty due to the double writes of WAL. NOVA [40], a PM-specific file system, relaxes the consistency guarantees for the mapped data by creating a replica only for mapping, resulting in significant write amplification. All existing works suffer from performance overhead caused by data consistency guarantees.

In summary, although MMIO shows better performance than traditional kernel-involved I/O, it still faces the challenge of ensuring the reliability (i.e., integrity and consistency) of memory-mapped data at a low cost. To address these issues, we propose **R**eliable **M**emory-**M**apped **IO** (RMMIO), a user-space I/O library that provides MMIO-like I/O performance while ensuring both data consistency and integrity for PM systems.

To minimize the extra overheads caused by reliability guarantees, RMMIO inherits the matured protection mechanisms of kernel-space file systems [10], [32], [41] by keeping all files in kernel space, without mapping them into user address space. However, existing protection mechanisms limit the performance of user I/O since users have to access PM files via low-speed system calls. Thus, we further employ a large contiguous persistent memory region as a cache layer, called *Persistent Page Cache* (*PPcache*), to accelerate user I/O. The cache directly resides in user space, so accessing *PPcache* is as fast as MMIO to the mapped file. Since *PPcache* is persistent, I/O requests arriving at *PPcache* will be persisted immediately, shortening the I/O path of RMMIO.

Owing to the reliable underlying file system, RMMIO only needs to take charge of the reliability of the data buffered in *PPcache*. To guarantee the consistency of *PPcache*, RMMIO provides atomic I/O interfaces by employing WAL (Write-Ahead Log). Considering that existing WAL has to copy old data as undo log to a specific log region before writing new data, to avoid being overwritten by new data. RMMIO builds a two-level structure to in-place preserve the old data. The new data can be written to another level to avoid overwriting old data, which mitigates the extra write for copying old data to the log region. Moreover, to prevent unrecoverable corruptions caused by scribbles, RMMIO also supports taking a snapshot for the data buffered in *PPcache*. The snapshot provides a consistent backup of *PPcache*. Once the scribble happens, RMMIO can recover the file with unaffected snapshots. To reduce the software overhead induced by snapshots, RMMIO implements incremental snapshots that only record updates to a file, which is much more efficient than full-copy snapshots due to mitigating significant data movements.

Although snapshots can increase data reliability, intemperately taking snapshots will result in a performance decline of I/O service and resource exhaustion of PM. On the other hand, if users do not take a snapshot in time, unexpected scribbles can lead to huge data loss. To trade off the reliability and availability of snapshots, RMMIO provides an enhanced snapshot mode called *autosnapshot*. While enabling *autosnapshot*, RMMIO will automatically take snapshots for *PPcache*. The automatic snapshots always follow the most recent updates to *PPcache* by creating new snapshots and recycling old snapshots in the background, without user intervention. Since auto-snapshot exacerbates the strain on PM resources, we also propose a set of strategies to manage the limited persistent memory resources for RMMIO, including snapshot merging, persistent page pool extension, and garbage collection. Specifically, these strategies are optimized for PM access characteristics (e.g., limited write bandwidth, poor read-write performance, and so on), which only incur negligible performance penalties for foreground I/O services. Our analysis shows that enabling *autosnapshot* only causes a 4.5% performance hit.

We evaluate RMMIO with both microbenchmarks and real-world applications. The experimental results show that RMMIO gains up to 8.49x higher throughput than ext4-DAX [37] and achieves 2.31x higher throughput than the state-of-the-art MMIO-based scheme [12] in write-intensive workloads while guaranteeing data reliability. In the evaluation of real-world applications, RocksDB accelerated by RMMIO achieves at most 706% higher throughput than that of ext4-DAX.

The contributions of RMMIO are as follows:

- We propose a new architecture for MMIO to ensure the data reliability of PM by introducing *PPcache*.
- Based on *PPcache*, we implement a user-space I/O library called RMMIO, which provides MMIO-like I/O performance while ensuring both data consistency and integrity for PM systems.
- We enhance RMMIO with auto-snapshot and efficient memory management strategies that provide better data integrity guarantees while keeping the outstanding performance of RMMIO.
- We provide a comprehensive evaluation to demonstrate the advantages of RMMIO in common I/O operations and real-world applications.

II. Background and Motivation

A. Accesses to Persistent Memory

Non-volatile Memories (NVMs) mitigate the performance gap between DRAM and block devices due to the DRAM-like performance and disk-like persistency. By using non-volatile memory (NVM) as persistent memory (PM), developers can durably store data in the memory space. Intel previously introduced a commercial persistent memory product known as the Optane DC Persistent Memory Module. According to the evaluations of Optane [42], the maximum bandwidth of PM is up to 13.9GB/s for writes and 39.4GB/s for reads, which is much better than that of disks and SSDs.

In traditional file systems, the bottleneck of data access is the poor I/O performance of block devices. Hence, the page cache residing in DRAM is used as the fast read cache and write buffer to reduce the number of block I/O. However, existing PM-aware file systems, e.g., PMFS [15] and ext4-DAX [37], hold the view that page cache is unnecessary for persistent memory systems, as it brings extra data copy due to the intermediate cache layer. Since software dominates the overheads of I/O operations in PM, PM-aware file systems remove page cache from the I/O path, and directly access the data residing in persistent memory, called direct access (i.e., DAX). DAX significantly reduces the software overhead but

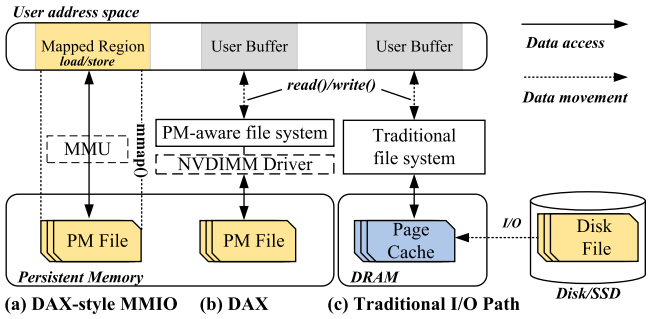


Fig. 1. Comparison between MMIO and Kernel I/O.

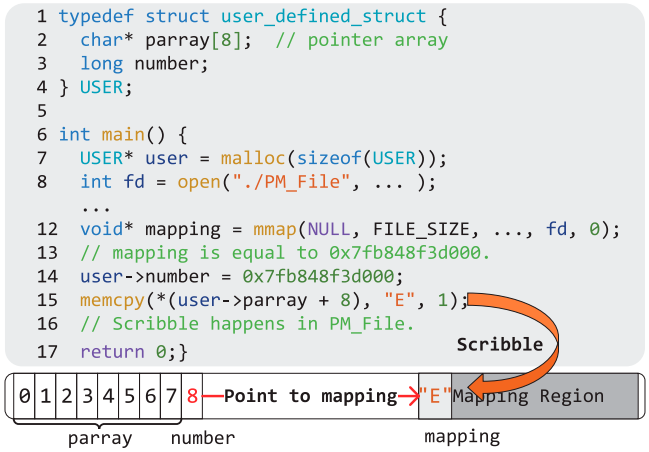


Fig. 2. An example of scribble caused by a user-space bug.

still gets the kernel involved. To further reduce the software overhead of accessing PM, PM-aware file systems propose DAX-style *mmap()*, a new memory-map approach that directly maps the file instead of the page cache. With DAX-style *mmap()*, users can access the target file via load/store, called DAX-style MMIO. Since DAX-style *mmap()* bypasses kernel I/O path (e.g., kernel-user switch, VFS, complex page index), the only overhead involved in DAX-style MMIO is the address translation through MMU. Fig. 1 compares the differences among DAX-style MMIO, DAX and traditional file access.

B. Data Reliability of MMIO in PM

1) Consistency: Persistent Memory extends the persistent domain to memory space. Thus we have to consider the data consistency of memory space in PM systems. To guarantee the data consistency of PM, all I/O operations should be atomic, i.e., to execute in an "all or nothing" fashion. Different from block devices, PM only provides 8-byte atomicity [12], [40] instead of block atomicity (512 bytes). Thus, it is a great challenge for PM programmers to guarantee data consistency for variable-length data in PM.

In systems equipped with persistent memory (PM), PM-specific bugs [27], [28] can also cause data inconsistency. For example, the commit flag of a data page persists before we copy the new data to the persistent memory due to the out-of-order execution [45]. In this case, the old data will be mistakenly treated as the new data, which is inconsistent with the commit flag. Thus programmers have to figure out when and in which order to flush the data in PM, which is nontrivial for developers.

2) Scribbles and Data Integrity: Scribbles are operations that randomly overwrite correct data with arbitrary values. Scribbles are well-known file system errors caused by both unreliable hardware [33], [34] and bug-prone software [9], [27], [28], such as buffer overflows, memory bitflips. Integrity is interpreted as data that has never been modified by unexpected scribbles.

We describe how scribbles compromise data integrity with the description of a bug instance shown in Fig. 2. *USER* is a user-defined structure composed of an eight-element pointer array (*parray*) and a long integer (*number*). In a common case, programmers can store a pointer to any memory region in *parray* and set *number* as any value within the range of a long integer. However, such a common instance could cause unexpected data corruptions in PM systems. We can map a PM file into user address space in PM systems. The address range of the memory-mapped file is between stack and heap, which can be easily overwritten by faulty memory accesses. We demonstrate the faulty memory access with the following assumptions: A PM file is mapped into an address space starting at 0x7fb848f3d000 and the programmer also sets *number* to 0x7fb848f3d000. If a programmer mistakenly accesses the element over the range of *parray*, e.g., *(user->parray + 8), he actually accesses the memory region of *number*. Note that the value of *number* is 0x7fb848f3d000, which is equal to the starting address of the memory-mapped file. As a result, a scribble will write "E" to the PM file, as shown in line 15 of the above instance. The data integrity of the PM file will be permanently broken by the scribble.

Such a consequence is unpredictable for new programmers working on persistent memory systems. Although existing works have proposed Machine Check Exception (e.g., ECC [19]) and software debugging tools [6], [7] to improve the system reliability, bugs and errors that cannot be detected by existing techniques still remain in the system. For example, ECC cannot detect 2-bit flipped errors or correct 1-bit flipped errors and GCC can not report faulty memory accesses induced by programmers. So computer systems still suffer from scribbles, especially for the data residing in the user address space due to the lack of protections from the kernel file system.

C. Motivations

Although PM-aware file systems have simplified the I/O path for PM, invoking a system call still introduces much higher overhead than performing MMIO. We measure the software overhead of ext4-DAX with *perf* [5]. The measurement shows that less than 44% execution time of a kernel-involved I/O is spent on copying data, far less than the 100% of DAX-style MMIO.

DAX-style MMIO brings significant performance gains but also exposes the file to the risk of user-space scribbles and

non-atomic I/O operations. Addressing these problems faces a great challenge in balancing performance and reliability. To the best of our knowledge, no previous work has fully addressed these problems. ext4-DAX [29], PMFS [15], Btrfs [32], and ZFS [10] focus on kernel-involved I/O without considerations on MMIO. SplitFS [23], Libnvmmio [12], and NOVA [40] guarantee the data consistency for MMIO but induce significant software overhead due to write amplification. In addition, all the mentioned works neglect to protect MMIO from scribbles. Although Nova-fortis [41] protects the mapped data with snapshots, it does not provide atomic I/O interfaces for MMIO. Therefore, there is a pressing need for providing an efficient and reliable MMIO for Persistent Memory Systems, which ensures both **consistency** and **integrity** for mapped data in PM.

III. RMMIO

This section provides a comprehensive view of RMMIO design. We first outline the design goals of RMMIO. Then, we go into depth about how RMMIO achieves these goals.

A. Desing Goals

Native MMIO outperforms existing kernel file systems [15], [37], [40], [41] on PM in terms of I/O performance. However, the native MMIO disregards the reliability guarantees for PM files. Despite the fact that previous works provided numerous ways to maintain data reliability, such as Write-Ahead-Log [12], [23], log-structuring [40], backup and snapshot [10], [32], [41], all of them impose considerable software overheads, compromising the performance of native MMIO. Thus the main goal of RMMIO is to achieve MMIO-like performance while guaranteeing data reliability. Specifically, RMMIO has three design goals in detail:

- **MMIO-like performance.** RMMIO should provide better I/O performance than kernel file systems, otherwise, users should directly employ kernel file systems.
- **Guarantee data consistency.** RMMIO should make sure that write operations never incompletely update a file or provide a method to undo the incomplete writes, guaranteeing data consistency.
- **Ensure data integrity.** RMMIO should provide redundant data and procedures for recovering a file from data corruption caused by scribbles.

B. Overview

Native MMIO is dangerous since it directly exposes PM files to user-space applications. Scribbles and inconsistent writes, acting like common user I/O, can also in-place update PM files, which cause permanent data corruptions in PM. Thus RMMIO never maps a file into user space and offloads the protections for PM files to the underlying file systems, e.g., NOVA-Fortis [41], which has already provided mature protections for data reliability.

Basic Architecture. The kernel-protected files cannot be mapped into user space, which results in compromised I/O performance. Therefore, RMMIO employs a memory-mapped cache layer upon PM-aware file systems to accelerate user I/O requests, as shown in Fig. 3. The cache is organized at file granularity by using a unique data structure, called *FILE*, for every opened file. *FILE* maintains a persistent page cache (*PPcache*) and a *snapshot list*: ***PPcache*** buffers the most recent updates to the related file; *snapshot list* records the incremental snapshots of the file, which provide necessary data redundancy to recover RMMIO from possible data corruptions. To achieve MMIO-like performance, *PPcache* and *snapshot list* are indexed by ***mapping tables*** located in DRAM, which is optimized for continuous indexing in PM. We present more details about ***mapping tables*** in Section III-C.

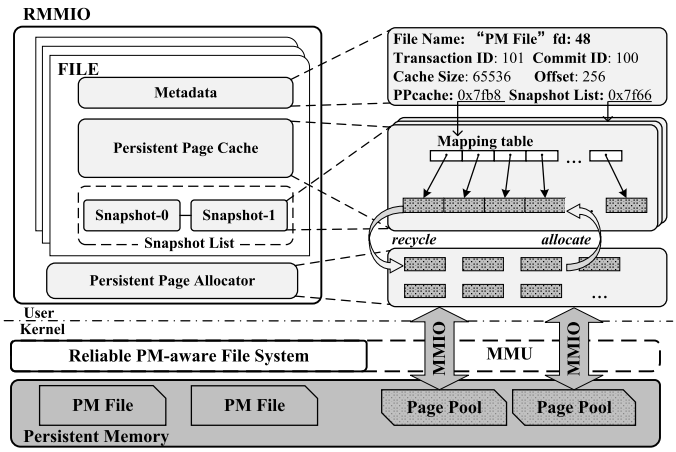


Fig. 3. Overview of RMMIO.

I/O Path. Introducing a cache layer lengthens the I/O path of existing systems since we may need to frequently synchronize data between the cache and underlying file systems to ensure the persistency of data. However, owing to the persistency of PM, *PPcache* built in PM avoids frequent synchronization. RMMIO removes synchronization from the critical I/O path by deferring the writeback of data in *PPcache*. RMMIO never writes the data buffered in *PPcache* back to the underlying file system until the idle time or when users actively write data back with *fsync()*. Although the *PPcache* extends the overall I/O path of RMMIO, it also provides fast persistency perfectly matching modern RPC-based systems [26]. Likewise, reads to a PM file will first be routed to the *PPcache* since the latest data may only be buffered in the *PPcache*. Then, if reads do not achieve any data from *PPcache*, RMMIO will invoke a system read to fetch the data from the underlying file system and buffer it in the *PPcache*.

Memory Management. The cache layer shares the same memory region with the Reliable file system. RMMIO employs a persistent page allocator to manage the persistent memory resources for *PPcache* and *snapshot list*. The persistent page allocator acquires memory resources by creating a page pool and mapping it into user space so that it can be directly accessed via load/store. The persistent memory resources are allocated and recycled at page granularity. Section III-H and Section III-G will go into further depth on memory management.

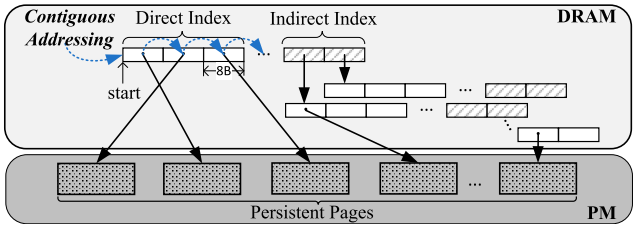


Fig. 4. Mapping table.

POSIX-like Interfaces. By introducing the persistent page cache, RMMIO currently provides six main interfaces to accelerate the I/O request in PM systems, i.e., ***read(), write(), snapshot(), open(), close(), fsync()***., whose usages are aligned with *POSIX* APIs except for *snapshot()*. More details about *snapshot()* will be shown in Section III-F.

The basic architecture of RMMIO ensures the reliability of kernel-protected files. However, massive data will be buffered in *PPcache* under contiguous I/O requests. We cannot diminish the reliability guarantees of data buffered in *PPcache* since corrupted data in *PPcache* will eventually be written back to the underlying file system. Thus, we have to take action to prevent inconsistent writes and scribbles to *PPcache*. In the worst case, *PPcache* may be broken by inconsistent writes and scribbles. RMMIO has to detect the corrupted data and recover the *PPcache* to a consistent and integrated state. In summary, we cannot contaminate the file with corrupted data in *PPcache*. In Section III-D to III-F2, we describe how RMMIO implements atomic I/O interfaces to ensure data consistency of *PPcache* and how RMMIO recovers corrupted *PPcache* from scribbles.

C. Mapping Table

RMMIO uses a flat range-optimized *mapping table* to index persistent pages for ***PPcache*** and snapshot. The design goal of *mapping table* is to simulate the addressing mode of native MMIO, i.e., short path walk and efficient range query. To achieve this goal, *mapping table* inherits the block-level addressing strategy from ext2/ext3, which places addresses of blocks in contiguous entries of a table. Fig. 4 shows the layout of a mapping table. A *mapping table* consists of *Direct Index* and *Indirect Index*. The *Direct Index* entry stores the pointer to a persistent page. Yet a *Indirect Index* entry stores the pointer to the beginning of a next-level *Direct Index* to extend the *mapping table*. Both *Direct Index* and *Indirect Index* are constructed from contiguous memory regions allocated from DRAM.

Short Path Walk. The tree-like index structures employed by previous works, e.g., radix tree [12] and extent tree [37], have to walk through numerous intermediate nodes to reach leaf nodes, which is a big drag on PM featuring low-latency. Thus *mapping table* in RMMIO removes intermediate nodes by indexing leaf nodes with offset. If the offset is within the range of *Direct Index*, RMMIO can directly calculate the address of the queried leaf node with the following method:

$$address = start + \frac{offset}{PAGE\ SIZE} * 8Bytes \quad (1)$$

If the offset is out of the range of *Direct Index*, RMMIO has to first get the start address of the next-level index by accessing *Indirect Index* with a specific index number. The method of calculating the index number of *Indirect Index* is:

$$index = \frac{offset}{Range\ of\ Direct\ Index} - 1 \quad (2)$$

After getting the start address, RMMIO can simply achieve the queried leaf node by following the above method1. At last, RMMIO can directly access the target page with the address stored in the leaf node. The time complexity of query in *mapping table* is practically constant. When only the direct index is activated, the time complexity is only O(1), similar to MMIO, which significantly reduces the software overheads of indexing a page. Even if the indirect index is activated while writing, according to our evaluation, indirect indexing will only result in a 5% additional software overhead at most.

Efficient Range Query. Although flat index structures, such as hashing tables, can also achieve short walk paths, they are unable to deliver satisfactory range query performance. A *mapping table*'s virtual address space is continuous since it is constructed using continuous memory regions. By using the above methods, RMMIO can calculate the start and end addresses of the nodes indexing the queried persistent pages. Though the addresses of persistent pages are not contiguous, the addresses of nodes that index these pages grow linearly. Even if the pages' range is out of the current *Direct Index*, RMMIO only needs one more calculation for the address range.

D. Ensure Data Consistency With Two-Level Page Cache

As described in Section II-B1, we have to consider the data consistency of *PPcache* residing in PM. Since a system crash can happen at any time, we need to keep the data consistent in PM all the time. We assume that the data in *PPcache* is consistent at the beginning. RMMIO has to provide atomic I/O interfaces to avoid breaking the consistency of *PPcache*.

The biggest challenge of atomic I/O comes from the limitation of PM systems. Modern processors support only 8-byte atomic writes for persistent memory [45]. However, RMMIO has to provide atomicity for writes with arbitrary lengths. To overcome this problem, previous works, e.g., Libnvmmio [12] and SplitFS [23], employ WAL (i.e., write-ahead-log), which induces extra PM writes for logging. Though NOVA addresses consistency problems by using log structuring, removing the extra writes from the critical I/O path, it still needs laborious garbage collection.

Since the persistent memory system is sensitive to the efficiency of software, RMMIO needs to get rid of extra writes and additional software overheads. To achieve this goal, RMMIO implements an optimized WAL mechanism that removes additional writes from the critical path by reusing old data as undo log. As shown in Fig. 5, RMMIO builds a two-level *PPcache* to maintain two versions of data for each page, i.e., the older data and the newer data. When a writer thread writes data to *PPcache*, the coming data overwrites the older data (smaller TID) but preserves the newer data (larger TID) as undo log. Once the ongoing write is interrupted by a system

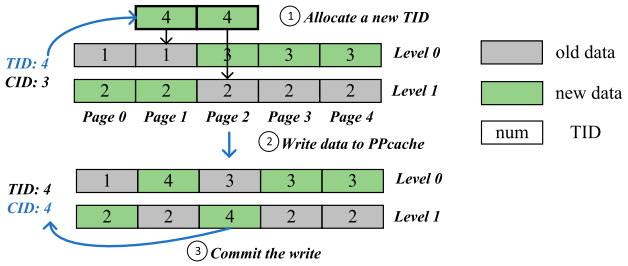


Fig. 5. Write 8KB data to the two-level persistent page cache.

TABLE I
THE EFFECT OF 4K ALIGNMENT ON THE WRITE PERFORMANCE OF PM/DRAM

Memory	I/O Pattern	Bandwidth
PM	16KB + 4K-aligned	4270MB/s
PM	16KB + 4K-non-aligned	3220MB/s
PM	8KB + 4K-aligned	4310MB/s
PM	8KB + 4K-non-aligned	2690MB/s
PM	4KB + 4K-aligned	4280MB/s
PM	4KB + 4K-non-aligned	1950MB/s
DRAM	4KB + 4K-aligned	5830MB/s
DRAM	4KB + 4K-non-aligned	6550MB/s

crash, RMMIO abandons the incompleted write and recovers from the undo log. To identify the incompleted write, RMMIO introduces a timestamp-based commit mechanism, inspired by Libnvmmio [12]. The timestamp-based commit mechanism includes two main components, i.e., a logical timestamp called TID (Transaction ID) and a global timestamp that indicates the latest transaction called CID (Committed ID). As RMMIO divides every write into several page writes. Pages within a write transaction will be marked with the same TID to indicate when they are written. Furthermore, RMMIO records the timestamp of the latest transaction using CID to figure out which page has been committed. All pages with a TID smaller than or equal to the CID will be identified as committed. Therefore, we can simultaneously mark any number of pages as committed by increasing the CID. Since CID is an 8-byte file-specific variable, every writer thread can update it atomically. With the timestamp-based commit mechanism, RMMIO tolerates data inconsistency after a system crash happens. Because CID is designed for orderly growth, a page with a TID larger than CID must be uncommitted. By comparing their TIDs with the CID, the inconsistent pages can be easily identified and removed. Therefore, RMMIO can always keep *PPcache* consistent even across a system crash.

Fig. 5 demonstrates an example of how RMMIO guarantees data consistency in a single-thread instance. The blank square represents a page in the *PPcache*, and the number inside the blank indicates the TID of the page. If a writer thread is trying to write 8KB data to *Page1* and *Page2*, RMMIO first allocates a new TID for the working thread by increasing the TID from 3 to 4. And then the writer thread marks two pages of data with TID-4. Second, RMMIO writes these two pages of data to *level0 of Page1* and *level1 of Page2*, respectively. Because the data in *level0 of Page1* and *level1 of Page2* is older than that in the other level. During the write, the data in *level1 of Page1* and *level0 of Page2* becomes the undo log for this write transaction. After all the data has been written to the related pages, the writer thread updates CID to 4 with an atomic write. Meanwhile, all pages with TID-4 are simultaneously identified as committed. Throughout the write transaction, *PPcache* is always consistent.

E. Enable High Scalability for RMMIO

Since PM is byte-addressable, it is easy for DAX-style MMIO to achieve high scalability. However, challenges come with opportunities. Because of the lack of thread isolation, the native MMIO cannot safely handle multi-thread I/O requests.

Fine-grained Lock. To guarantee thread safety, RMMIO employs a reader or writer lock for thread isolation. In addition, we note that the file-grained lock used in VFS blocks concurrent operations on a shared file [39], which is a waste of the byte-addressable PM. Thus RMMIO needs a fine-grained lock to achieve high scalability. To determine the most appropriate granularity of a lock, we evaluate the native MMIO with different I/O patterns in PM, as shown in Table I. The experimental results show that the maximum bandwidth appears when the I/O is 4KB-aligned. According to the evaluation, the granularity of a lock should be 4KB or a multiple of 4KB to take full advantage of PM. However, automatically determining the specific granularity of the lock for different workloads is out of the scope of this paper. Therefore, we configure the default granularity of a lock as 4KB to fully expose the raw performance of PM.

Atomic Primitives. RMMIO also ensures thread safety with atomic primitives. As locking may fall into the kernel, locking for every thread-safety operation will cause a significant performance decline. Thus, RMMIO employs atomic primitives provided by *glibc* to deliver TID and update CID for working threads. Since these atomic primitives guarantee the thread safety of the operand, every writer thread will get the unique TID by using FAA (i.e., *atomic_fetch_add_explicit*). Moreover, we use CAS (i.e., *_sync_bool_compare_and_swap*) to ensure the linearizability consistency of RMMIO write. In other words, while several threads holding different TIDs are working together, RMMIO needs to make sure that every thread commits its writes in the order of the TID. Since CAS only updates the old value when the old value matches the given value, RMMIO sets the given value as "TID minus 1" (TID is the current logical timestamp of a thread). Thus, a thread will not update CID until the last thread (thread holding TID-1) has committed its write. The atomic CAS makes sure that write transactions from different threads will be committed one by one.

F. Recover Corrupted Files With Incremental Snapshots

Scribble is a serious issue for RMMIO because scribbles break the data integrity of *PPcache*. According to Ganesan et al., [17], in modern distributed storage systems, a single file-system fault can cause catastrophic consequences. However, scribbles are inevitable and unpredictable as we described in Section II-B2. To combat data corruptions caused by scribbles, RMMIO maintains a CRC32C checksum for every page (shown in Fig. 6) and examines the checksum to check the integrity of

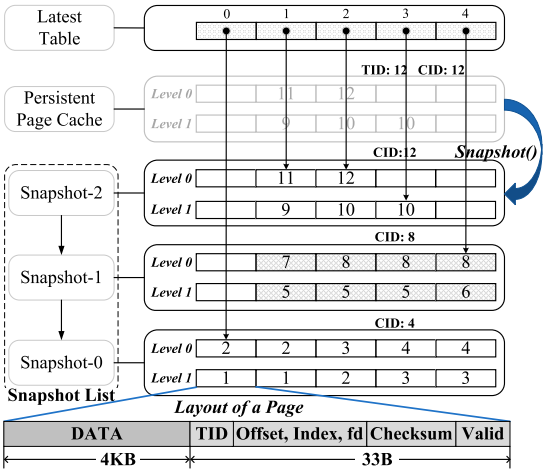


Fig. 6. The layout of snapshots and pages in RMMIO.

a page. The mismatch of checksum indicates data corruptions. To recover from data corruptions, existing works [10], [32], [41] has proposed two strategies, i.e., full-copy backup and snapshots, to recover the corrupted file from data corruptions. However, all of them incur a huge account of data movements, causing performance loss for MMIO. Yet, a system without backups and snapshots will compromise data reliability. To overcome this challenge, RMMIO proposes an incremental snapshot mechanism that dramatically decreases the software overhead of snapshots.

1) Incremental Snapshots: As shown in Fig. 6, the incremental snapshot is built based on *PPcache*. When the user takes a snapshot, RMMIO directly packs the data buffered in *PPcache* as a snapshot. So each snapshot only retains the updates to a file instead of the full copy of a file, which significantly decreases the software overhead on snapshot capture. Furthermore, we note that the *mapping table* (Section III) records pointers to persistent pages buffering the updates of a file. So if we want to build a snapshot with these persistent pages, we only need to copy the pointers to these pages without copying the data. However, simply copying pointers will lose the layout of data. Rebuilding a new *mapping table* and migrating data from *PPcache* to the new *mapping table* can address this problem but induces non-negligible software overheads caused by reindexing all pages in the *PPcache*. To further reduce the software overhead of taking a snapshot, RMMIO directly converts the *PPcache* of into a snapshot. Specifically, once a user invokes *snapshot()*, RMMIO first links the *PPcache* (snapshot-2 in Fig. 6) to the *snapshot list* and then builds a new *persistent page cache* with the empty *mapping table*. As a result, *snapshot()* in RMMIO only needs to initialize a new *mapping table* without any data copying.

Once a scribble happens in a file, RMMIO can recover the file from snapshots that have not been corrupted. The recovery cannot restore the corrupted data but can return the file back to a historical version to minimize the impact on the whole system. Upon detecting data corruption and initiating a version rollback, RMMIO sends a SIGUSR1 (user-defined signal) to the current process using the *kill()*. Users can register a signal handler using signal() to address the data corruption with the information provided by RMMIO, including the corrupted file name, TID and the corrupted data address. More details about the recovery of RMMIO will be presented in Section III-F2.

Latest Table. We also note that taking snapshots in RMMIO may cause the performance decline of reads. Because snapshots may still buffer the latest data of a file, which increases the difficulty of *read* to query the latest data. As shown in Fig. 6, reads must traverse all snapshots to find the latest data. Such an inefficient traversal operation extends the critical path of RMMIO read, which goes against RMMIO's design goals. To avoid traversing these snapshots, RMMIO builds a *latest table* to store the pointer to the latest data for each page. The *latest table* will be updated along with the updates to persistent pages. Since taking a snapshot in RMMIO does not move any persistent page, the pointer to the latest page will always be constant during snapshot capturing. So RMMIO can always get the latest data by accessing the *latest table*. The time complexity of querying a page using *latest table* in RMMIO is O(1) which is much more efficient than traversing all snapshots.

2) Recovery: The recovery of RMMIO follows two steps: First, we remove the corrupted data whose checksum mismatches the stored one. Then, RMMIO rolls the corrupted file back to a historical version that does not contain any corrupted page. Note that all the data with TID larger than the corrupted one should be discarded and the *roll-back* should be aligned with the snapshots.

We further demonstrate how *roll-back* works with a case based on the *PPcache* and *snapshots* shown in Fig. 6. If a scribble happens in *data-10*, RMMIO should discard all pages with a TID larger than 10 and relink the *latest table* to snapshot-1. We do not retain *data-11* and *data-12* because storing the data with TID larger than that of the most recent available data violates the consistency guarantees, as we described in Section II-B1. In addition, we also do not roll back to the version with TID as 9 because the data with TID-9 could have been partially overwritten by data-10. Thus, rolling back to the TID-9 also goes against the data consistency. That is why the *roll-back* should be aligned with the snapshots.

Recovery across a System Crash. In the event of a power failure, the *mapping table* residing DRAM will be lost. We cannot find the persistent pages buffering the data of a file without the *mapping table*. Thus, RMMIO appends the location (i.e., *offset, index, fd* in Fig. 6) of a page after the data area of each persistent page. By traversing the locations of the pages in the page pool after a system crash, RMMIO can rebuild the *mapping table* snapshots of a file and recover the file. However, the file descriptor (*fd*) will be reset after a system reboot. We cannot find the target file correctly by using the appended *fd* because the *fd* may be allocated to a different file after the system reboots. Although we can directly append the absolute path of a file after every page, the lengthy path name causes non-negligible additional writes. To address this problem, RMMIO creates a persistent *file descriptor table* to permanently record file descriptors' relationship with the absolute path of an opened file. Thus, RMMIO can exactly know where to write these pages

back by looking up the *file descriptor table*. In addition, the latest data will also be identified by comparing the TID of pages so we can even recover the *latest table* by accessing the metadata area of persistent pages. Maintaining metadata for a page only takes 33 Bytes but helps us recover from a system crash.

G. Automatic Snapshot

RMMIO sacrifices the persistence of some data to maintain data integrity because data corruptions are more dangerous than data loss. For example, if a user is modifying "*grub.cfg*" with RMMIO, a corrupted modification will cause the system to fail to start. If data is lost, the user only needs to rewrite it once.

However, huge data loss is also unacceptable for users. The existing snapshot in RMMIO has to be manually built by users with *snapshot()*. However, users may be confused about when to build a snapshot since users are not aware of the layout of the data in *PPcache* and the recovery mechanisms of RMMIO. If users lazily take snapshots, there will be a huge data loss when scribbles happen. Because the most available snapshot could be far away from the latest data. Furthermore, if users frequently build snapshots without recycling them, the memory space will soon run out. Despite RMMIO can provide an interface for users to recycle useless snapshots, the complex management of snapshots may still trouble users and lead to non-negligible performance decline. To address these issues, RMMIO devises an enhanced snapshot mode (we refer to *auto-snapshot*) that automatically captures a fixed number of snapshots for a file and keeps updates snapshots to follow the users' updates. The *auto-snapshot* consists of an automatic capture strategy and snapshots management strategy.

1) Automatic Capture: While enabling *auto-snapshot*, RMMIO captures snapshots at fixed time intervals. The time interval is a predefined gap between two logic timestamps (i.e., TID). In other words, RMMIO will capture a snapshot after a fixed number of writes. The automatic capture should guarantee the data consistency of captured snapshots. So RMMIO does not capture a snapshot until the foreground ongoing write has been committed. Furthermore, RMMIO has to lock the PPcache to guarantee there are no more writes to the *PPcache*. Thus, RMMIO captures snapshots in foreground threads to avoid the software overheads caused by synchronization between foreground I/O threads and the background thread. Since taking a snapshot has to initialize a new *mapping table*, frequent *auto-snapshot* will greatly increase the tail latency of I/O requests. RMMIO mitigates the software overheads of initializing a *mapping table* by batch preallocation and initialization. Therefore, the foreground *auto-snapshot* only needs to acquire a preallocated *mapping table* without a long time stall.

2) Snapshots Merging: The memory footprint of snapshots is constrained under the set threshold by RMMIO's automatic recycle of snapshots frequently captured by both *auto-snapshot* and *snapshot()*. Therefore, RMMIO tightly enforces a threshold on how many snapshots are allowed to be captured for a file. When the number of snapshots is over the predefined threshold, RMMIO takes charge of shrinking snapshots. However, writing the oldest snapshots back to the underlying file system will cause a non-negligible time stall for the foreground I/O service. Although we can apply a background thread to recycle snapshots to mitigate the foreground time stall, the limited bandwidth and poor read-write-mixed performance of PM [42] still bothers the foreground I/O services. To overcome this challenge, RMMIO proposes a lightweight recycling strategy called *snapshots merging*, which avoids access to PM, i.e., both read and write.

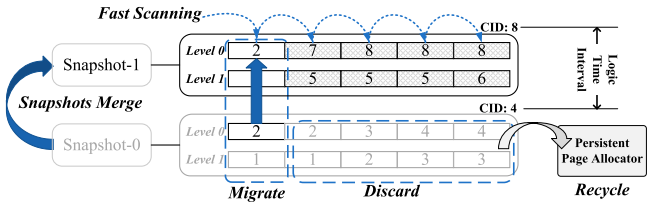


Fig. 7. Snapshots merging in RMMIO (snippet from Fig. 6).

Snapshots merging merges the oldest two snapshots of a file into a new snapshot by retaining the newer data and discarding the older data for every page. The new snapshot still maintains the latest data of two merged snapshots but recycles some redundant data to restrict the memory footprint of snapshots. The *snapshot merging* works in a background thread but is triggered by the foreground I/O services. In addition, we further propose two strategies to mitigate the reads and writes to PM, i.e., fast scanning and shadow migration, respectively.

Fast Scanning. As the instance shown in Fig. 7, if the foreground thread has created more snapshots than the set threshold (e.g., 2 snapshots), the background merging thread starts to merge the oldest two snapshots, i.e., snapshot-0 and snapshot-1. We may need to scan every page in both snapshots-0 and snapshot-1 and compare their TIDs to decide whether a page should be discarded or migrated to snapshot-1. However, scanning all pages consumes too much execution time and results in numerous PM reads when accessing TID. In fact, we do not have to traverse all pages'TID in these two snapshots since the data in snapshot-1 is always newer than that in snapshot-0. Based on this observation, we propose a fast scanning strategy to accelerate the snapshot merging. The fast scanning only traverses the newer snapshot, i.e., snapshot-1, to get the pointer to the latest data of snapshot-1. If the pointer is not NULL, RMMIO directly skips this page in snapshot-1 and discards this page in snapshot-0, as page1-4 shown in Fig. 6. On the contrary, if the pointer is NULL, it means snapshot-1 has never updated this page but the data may be updated by snapshot-0. Thus RMMIO has to further get the latest data of snapshot-0 and merge it into snapshot-1, as page0 shown in Fig. 6. Throughout the whole scanning, RMMIO does not induce any PM read as the mapping table is built in DRAM (Section II-C).

Shadow Migration. While merging two snapshots, RMMIO has to migrate pages from snapshot-0 to snapshot-1. However, directly copying data from snapshot-0 to snapshot-1 will cost a high write bandwidth of PM. To address this problem, RMMIO employs *shadow migration* to migrate data between

two snapshots without any PM writes. *Shadow migration* is based on RMMIO's hybrid memory architecture shown in Section III-C. We only store recovery-related data and metadata in PM (as shown in Fig. 6) but build the run-time data structures in DRAM. Specifically, we build a mapping table in DRAM to index a snapshot and store pointers to PM pages in the entries of the *mapping table*. Thus migrating data between two snapshots only takes a pointer exchange in DRAM, without the need to induce any write to PM. Moreover, the software overhead of *snapshots merging* is greatly reduced by *shadow migration* since it mitigates numerous store instructions from the critical path of merging.

H. Memory Management

RMMIO builds global persistent page allocators for all working threads to dynamically allocate and recycle persistent pages. The default size of the page pool for each thread is 4GB, which is inadequate for data-intensive applications, e.g., RocksDB. Thus, to satisfy the excessive demand for persistent pages, RMMIO proposes two strategies, i.e., Garbage Collection and Pool Extension, to provide enough pages for every busy thread.

Garbage Collection. The data buffering in *PPcache* and snapshots can be divided into two kinds of data, i.e., useful data and useless data. The useful data refers to (1) the data that is linked to the *latest table*, e.g., *level0 of Page0* in snapshot0 of Fig. 6, which may be read by future user requests. (2) The useful data also includes the newest pages in each snapshot even if they are not linked to the *latest table*, e.g., *level0 of Page1* in snapshot1 of Fig. 6, since they are necessary for the recovery of RMMIO. Pages that do not belong to useful data are called useless data, e.g., *level1 of Page0* in snapshot0 and *level1 of Page1* in snapshot1 of Fig. 6. They all used to be the undo log for a page write. The undo log is not necessary anymore after a write commit. RMMIO permanently maintains useful pages but recycles useless pages with garbage collection. We create a background thread to look for useless pages in *PPcache* and snapshots. To recycle useless pages, RMMIO simply resets the valid bit of a page and then pushes it back to the persistent page allocator. The reset valid bit indicates that the data in this page is not valid anymore. Thus, RMMIO does not need to reset every bit of the page, reducing huge store instructions. Moreover, we also do not need to copy data from those useless pages to the underlying file systems since the data in these pages has already been overwritten by the new data. In addition, the recycling of these useless pages is similar to the *shadow migration*, which only needs to move the pointer of the useless page to the allocator queue, without any PM write. The garbage collection can release at most 50% persistent pages of *PPcache* and snapshots.

Pool Extension. Although RMMIO recycles useless pages, a persistent page pool can easily run out in data-intensive workloads. In this case, RMMIO extends the persistent page allocator by allocating new page pools, instead of writing useful data back to the underlying file system. Because allocating 4GB persistent memory (1.14s) is almost 32x as fast as 4GB data migration (36.52s) from *PPcache* to the underlying file system. In addition, RMMIO extends the persistent page pool before it becomes full (at 70% utilization), which avoids a long tail latency of RMMIO. Since RMMIO is designed as an extension of the underlying PM file system and guarantees data reliability, maintaining data in user-space *PPcache* is also a practical strategy for permanent storage.

IV. Evaluation

This section presents a comprehensive evaluation of RMMIO through both microbenchmarks and real-world applications. We also compare RMMIO with state-of-the-art PM-aware file systems and two MMIO-based works, including NOVA [40], PMFS [15], ext4-DAX [37], SplitFS [23] and Libnvmmio [12]. In the following sections, we demonstrate RMMIO's performance in common I/O workloads (Section IV-B), software overhead caused by reliability guarantees (Section IV-B), scalability (Section IV-C), performance decline caused by *auto-snapshot* (Section IV-D), and performance in the real-world applications (Section IV-E).

A. Experimental Setup

We implement the experimental evaluation on a system equipped with 2-socket Intel Xeon 6230R, 12 * 16GB DDR4 and 12 * 128GB Optane DC Persistent Memory. To enable persistency and high bandwidth of PM, all Optane DC Persistent Memory Modules are configured as App Direct Mode with interleaving [22]. We use *numactl* to bind all working threads and memory regions to the same NUMA node to avoid cross-node memory access. Finally, our evaluation is performed on Linux kernel 4.13 with FIO [8] as microbenchmark and RocksDB as the real-world evaluation platform.

B. Single-Thread Evaluation

As shown in Fig. 9(a), we evaluate RMMIO in read/write with both sequential 4KB I/O and random 4KB I/O. Since the I/O path of RMMIO is much shorter than any competitor, the sequential write throughput of RMMIO is 2.54x, 1.18x, 1.46x, 1.98x, and 1.37x higher than that of NOVA, SplitFS, PMFS, Libnvmmio, and ext4-DAX, respectively. The random write performance shows similar results. In read evaluations, the throughput of RMMIO is slightly lower than that of MMIO-based works (i.e., Libnvmmio, SplitFS). Because RMMIO must calculate the checksum of every page to examine data integrity while the other MMIO-based works dismiss the data integrity guarantees. The read performance of RMMIO is expected to be close to other MMIO-related works by using a better CRC32C accelerator.

We further evaluate RMMIO's write performance with variable I/O sizes, as shown in Fig. 9(b). Whatever the I/O size is, RMMIO always shows higher write throughput than any related work. Specifically, for 512KB sequential writes, RMMIO achieves a maximum bandwidth of 3818MB/s, which is over 44% higher than that of ext-DAX. Even compared with SplitFS (strict mode), RMMIO obtains performance gains up to 2.19x owing to the lightweight log strategy (Section III-D). With

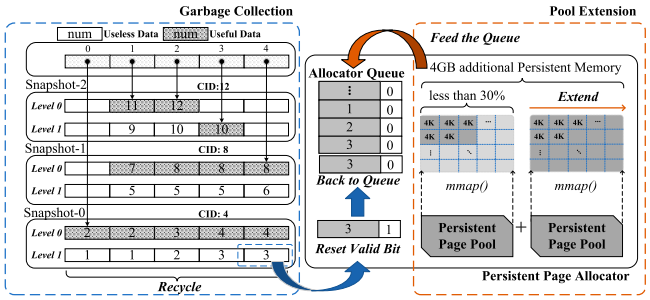
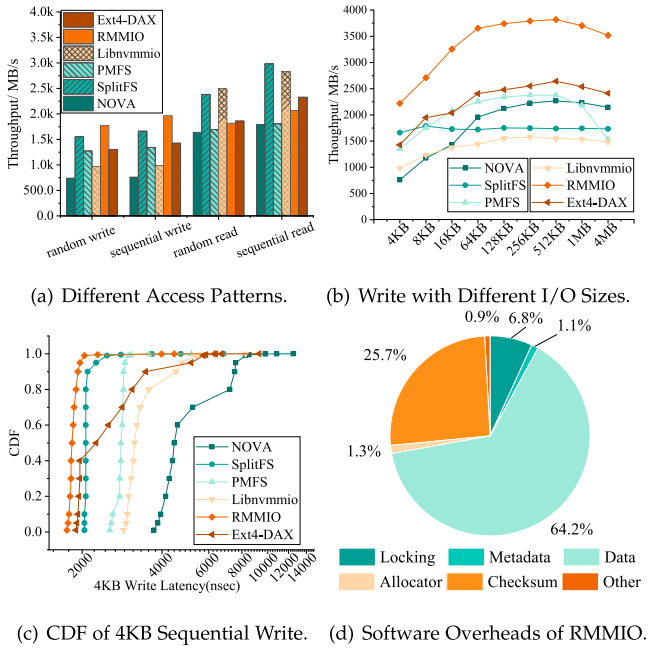


Fig. 8. Memory management of RMMIO.



(a) Different Access Patterns. (b) Write with Different I/O Sizes.

(c) CDF of 4KB Sequential Write. (d) Software Overheads of RMMIO.

Fig. 9. Performance overview of RMMIO.

the increase in I/O size, the performance improvement ratio of RMMIO also exceeds the state-of-the-art works. Because the *mapping table* in RMMIO is more efficient than tree-like index structures, e.g., extent tree in ext4-DAX, radix tree in Libnvmmio, for range querying multiple pages.

As the *Cumulative Distribution Function* (CDF) of 4KB write latency shown in Fig. 9(c), the write latency of RMMIO is lower than all related works. Especially, the P99 latency of RMMIO writes is only 1704ns, which is even lower than the minimum latency of NOVA (3728ns) and PMFS (2544ns). Because RMMIO buffers the data in the *PPcache* to avoid falling into the kernel. In addition, the logging strategy of RMMIO does not have to write log before writing data, removing the extra write from the critical IO path. Thus, RMMIO also outperforms Libnvmmio (2864ns) in write latency, which is also based on MMIO.

In Fig. 9(d), we measure and quantify five major software overheads of RMMIO. The two main components, i.e., data copy and checksum calculation, take up almost 90% of the

TABLE II
THE MEMORY FOOTPRINT OF A 128MB FILE IN PPCACHE

PM Structures	Size	DRAM Structures	Size
data	128MB	mapping table	2.28MB
metadata	1.03MB	thread metadata	3MB
fd table	5.23MB	hash table	781.31KB
FILE	80.625KB		
	Total	140.38MB	

execution time in RMMIO write. Although RMMIO spends over 20% of the execution time on checksum calculation, the data copy still accounts for up to 64%. The experimental results show that RMMIO is still more efficient than ext4-DAX (less than 44% [12]).

As shown in Table II, we evaluate the memory footprint of RMMIO after writing a 128MB file. The data of the file only takes up 128MB, which does not induce any space amplification because RMMIO has recycled the old data in time. Moreover, the other basic components of RMMIO (i.e., fd table, thread metadata, hash table) and file-related data (metadata, mapping table, FILE) only spend 12.38MB of memory space. Note that thread metadata, hash table and fd table will not grow as the number of files and the size of a file. Because they are all preallocated at the initialization of RMMIO. Thus, the space amplification ratio of RMMIO will be less than 9.6%.

C. Scalability

Scalability is another essential advantage of MMIO as it is byte-addressable. In modern multi-core systems, write-intensive applications benefit from highly scalable I/O interfaces. To compare the scalability of RMMIO with other MMIO-based schemes and kernel file systems, we evaluate RMMIO and its competitors with concurrent 4KB-sequential write/read-write I/O to multiple files or a shared file. The experimental results of concurrent I/O are shown in Fig. 10, except for SplitFS. The reason is that SplitFS does not support concurrent writes to a shared file and concurrent read-write mixed workloads in strict mode.

According to Fig. 10(a), the maximum bandwidth of RMMIO concurrent write exceeds that of NOVA, PMFS, Libnvmmio, ext4-DAX by 14.07x, 7.50x, 1.96x, 8.49x, respectively. Since page-grained locks do not block the nonoverlapping I/O operations, the concurrent writes of RMMIO to a shared file could be fully paralleled. Thus, the throughput of RMMIO increases almost linearly with the number of threads. Although libnvmmio also employs page-grained locks for concurrent execution, it does not follow the 4KB-aligned access to PM (Section III-E), resulting in both single-thread and concurrent throughput penalty.

RMMIO follows the reader/writer locking mechanism adopted by VFS thus achieving high scalability in read-write workloads. Furthermore, RMMIO employs a fine-grained reader/writer lock for every page. Thus, multiple reader threads can work on the same page, and multiple writer threads can write different pages of a shared file at the same time,

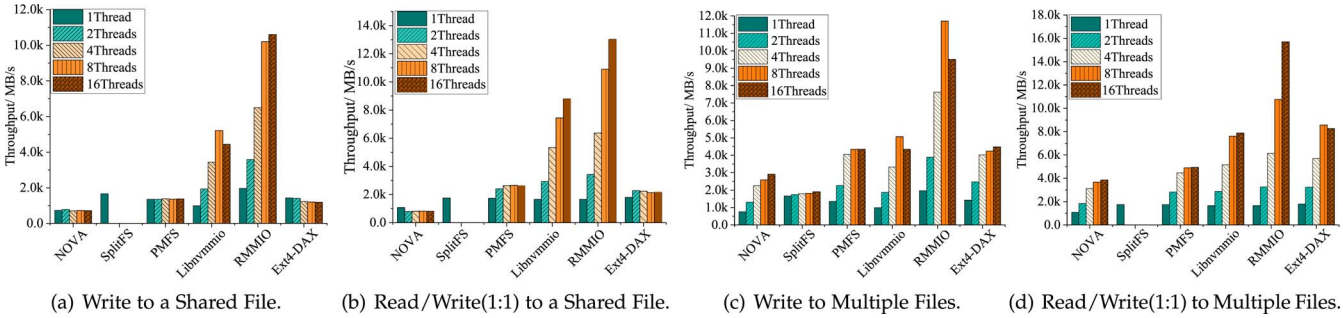


Fig. 10. Concurrent performance of RMMIO.

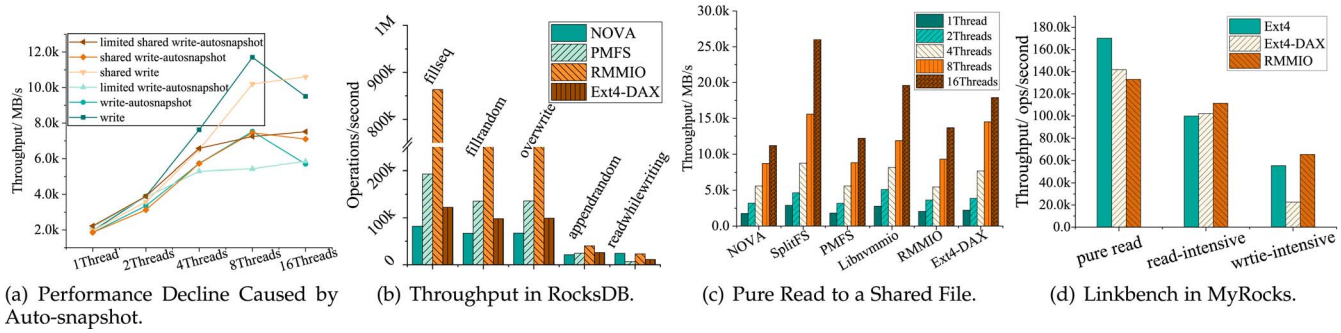


Fig. 11. Auto-snapshot and real-world applications.

which is blocked by kernel file systems (e.g., ext4-DAX, NOVA). Moreover, RMMIO also employs atomic primitives to mitigate frequent locking for thread safety. As a result, RMMIO gains up to 15.94x higher throughput than NOVA and 5.97x higher throughput than ext4-DAX as shown in Fig. 10(b). Although the single-thread read performance of RMMIO is slightly lower than other competitors, the scalability of RMMIO can complement the performance penalty caused by RMMIO read.

In Fig. 10(c) and 10(d), we evaluate the impacts of other limitations for concurrent execution without sharing a file between threads. For concurrent writes to multiple files, RMMIO still exceeds NOVA, SplitFS, PMFS, Libnvmmio, and ext4-DAX by 4.51x, 6.42x, 2.70x, 2.31x, and 2.75x, respectively. By analyzing their software stacks with *perf* [5], the advantage of RMMIO comes from the thread-local log region (i.e., *PPcache*), which has no log contention, compared with the shared log region in other works, e.g., JBD2 in ext4, WAL in SplitFS. Though NOVA employs a per-inode logging strategy, it still spends much time on garbage collection. The slight performance degradation of RMMIO over 8 threads, shown in Fig. 10(c), is due to the limitation of Effective Write Ratio (EWR) [22] in PM. Since different threads access data with different addresses, the concurrent access to PM is similar to random access to PM. Random access to PM will result in frequent built-in cache replacement. Thus, as the number of threads increases, concurrent access to PM triggers more and more unnecessary internal write amplification, resulting in the external performance decline.

D. Performance Decline Caused by Auto-Snapshot

To measure the performance decline caused by *auto-snapshot*, we reevaluate RMMIO in two write-intensive workloads, i.e. concurrent write to multiple files (write in Fig. 11(a)) and concurrent write to a shared file (shared write in Fig. 11(a)). The experimental result shows that the single-thread performance decline is only 4.5%. Because the snapshot capture only takes negligible overheads and the background snapshot merging does not block foreground I/O operations. The performance decline grows up with the number of working threads. The maximum performance decline of write and shared write is 39.9% and 33.1%, respectively. Although the snapshot merging does not block foreground I/O operations, the single-thread merging is too slow to merge snapshots for 16 threads at the same time. Since too many snapshots cannot be recycled in time, the persistent page pool will soon run out under the continuous writes. Thus, the background pool extension will be frequently triggered. The foreground I/O services have to wait for the extension thread to provide enough persistent pages. But a single extension thread cannot concurrently extend 16 page pools, slightly blocking the foreground I/O operations. Though it is easy to build more background merging threads and extension threads, RMMIO only employs one for each process to minimize the usage of system resources. Even in this case, the write throughput of RMMIO still outperforms all competitors while enabling *auto-snapshot*, as shown in Figs. 10 and 11(a).

We also evaluate RMMIO with a limited number of CPU cores, which is equal to the number of foreground working threads. As shown in Figure A, the shared-write performance of

RMMIO does not drop down while working in limited cores. Instead, RMMIO achieves higher bandwidth than that of the unlimited workload. It is because the background thread occupies some CPU time of working threads, which also reduces the contention of lock among working threads that share a file. Thus, the background thread actually takes full use of the CPU resources that used to be wasted by lock contention, which also increases the possibility of acquiring a lock for working threads. In other words, the computation resources needed by RMMIO for background snapshot merge are negligible. In concurrent write evaluations, limited RMMIO shows a 27.9% performance decline compared to unlimited RMMIO, while working with 8 threads. That is because the extending thread is fully loaded under continuous write operations. The extending thread and merging thread both compete with foreground working threads for CPU time, which makes the computation bottleneck appear earlier than unlimited RMMIO. However, the bottleneck will not tighten further, because the two background threads at most take up two physical cores. The 16-thread concurrent write performance proves that the two background threads do not further decrease the bandwidth of RMMIO.

E. Real-World Applications

To demonstrate how RMMIO performs in real-world applications, we adapt RMMIO to RocksDB [16] and evaluate it with five built-in benchmarks in *db_bench*, i.e., *fillrandom*, *readwhilewriting*, *fillseq*, *overwrite*, and *appendrandom*. Every benchmark includes 10,000,000 key-value (KV) operations and each key-value pair contains 16B key and 1024B value. Since SplitFS and Libnvmmio have not been adapted to RocksDB, we only compare RMMIO with state-of-the-art PM-aware file systems in this evaluation.

RMMIO provides atomic I/O interfaces so that we avoid inefficient WAL in RocksDB, which resides in the critical path of RocksDB writes and even blocks KV operations in some specific cases. So RocksDB accelerated by RMMIO exceeds all competitors in every benchmark as RocksDB is a write-intensive application. Especially, in *fillseq*, RMMIO accelerates RocksDB by 7.06x compared with ext4-DAX, and 10.54x compared with NOVA. Because RMMIO is optimized for sequential I/O via *mapping talbe*, which is more efficient than the tree-like indexing structures. Randomly inserting KV pairs with RocksDB will cause the reorder of KV pairs, called compaction, which reads KV pairs from the underlying file system and writes them back again after reordering KV pairs. Thus, benchmarks that randomly insert KV pairs, i.e., *fillrandom, overwrite, appendrandom*, will suffer from frequent I/O operations, including both read and write. Although RMMIO is not good at read, the outstanding write performance complements the read penalty. Specifically, in *overwrite*, RMMIO still outperforms NOVA, PMFS, and ext4-DAX by 3.93x, 1.95x, and 2.67x, respectively. Even in the read-write-mixed benchmark, i.e., *readwhilewriting*, RMMIO gains up to 3.39x higher throughput than PMFS, and 2.04x higher throughput than ext4-DAX. The RocksDB accelerated by RMMIO outperforms all competitors in all evaluated benchmarks, indicating that RMMIO can efficiently accelerate write-intensive applications.

Although RMMIO is designed as a write-optimized IO system, we still demonstrate the performance of RMMIO in read-intensive workloads to make sure that the read performance is not a burden on RMMIO. We adopted RMMIO on MyRocks [3], a variant of MySQL, which replaces the default storage engine with RocksDB. We evaluate the performance of MyRocks with Linkbench [2], a database benchmark developed to evaluate database performance for workloads similar to Facebook's production MySQL deployment. Our evaluation includes pure-read, read-intensive (80% read) and write-intensive (80% write) workloads.

As shown in Fig. 11(c), the pure read performance of RMMIO is slightly lower than that of ext4-DAX and other MMIO-based works. Because RMMIO strictly examines the data integrity while reading a page, which caused a significant performance decline. Furthermore, the page-grained lock mechanism also increases the complexity of reading a file. Thus both RMMIO and libnvmmio perform poorer than SplitFS, which only employs a file-grained lock and directly maps files into user address space. As shown in Fig. 11(d), the poor read performance truly decreases the throughput of RMMIO in the pure-read evaluation of Linkbench. However, the ratio of performance decline compared with Ext4-DAX is only 6.1%, which is far less than our expectation due to the performance gap shown in Fig. 11(c). The reason is that the RocksDB in MyRocks can employ a block cache to accelerate read operations, which works like the system page cache. The block cache fills the performance gap between RMMIO and Ext4-DAX. The write-intensive evaluation indicates that RMMIO exceeds both Ext4 and Ext4-DAX as expected. The only unexpected thing is that Ext4 far outperforms Ext4-DAX. We think this is because MyRocks does not immediately write all data back to storage with *fsync()*. While RMMIO and Ext4-DAX directly write data back to PM, without any intermediate DRAM buffer (e.g., page cache in Ext4). RMMIO performs better than Ext4 because RMMIO does not need to enable the WAL in RocksDB. The evaluation results of read-intensive workloads prove that even though the read performance of RMMIO is not as good as other MMIO-based works, the overall performance of RMMIO is still better than Ext4-DAX and Ext4 by 9.1% and 11.7%, respectively.

F. Reliability

Per-page checksum and auto-snapshot make RMMIO able to detect any scribbles and recover from data corruption caused by scribbles. To examine the reliability of RMMIO, we implement a malicious process to inject scribbles into PM while running RMMIO. The malicious process includes several different modes that can inject scribbles with sizes ranging from 1KB to 40KB in up to 16 threads. The number of random injections in each evaluation is 2, 000, 000, 000. The experimental results show that RMMIO can detect all scribbles and recover from them without data inconsistency.

V. Related Works

Although we have discussed some works that are highly related to RMMIO in Section II, there are still some more interesting works that are not mentioned, which inspired us while designing RMMIO. In this section, we will introduce these works and compare them with RMMIO.

BPFS [14] proposes a basic architecture for the PM file system and first considers the data consistency of PM. The most brilliant contribution of BPFS is the short-circuit shadow paging (SCSP), which allows BPFS to use copy-on-write at fine granularity, atomically committing small changes at any level of the file system tree. SCSP significantly reduces the write amplification compared with log-based atomic updates but it relies on a tree-like indexing structure. RMMIO can achieve the same data consistency guarantees as BPFS but with a more efficient indexing structure (*mapping table*).

SCMFS [38] proposes a user-space file system that implements all IO operations in user address space, which is similar to RMMIO. SCMFS maps each file into a contiguous address space and directly indexes files with the virtual address. Although the indexing method of SCMFS is more efficient than the *mapping tables* of RMMIO, it does not take data consistency into consideration. With the help of *mapping tables*, RMMIO can provide atomic I/O interfaces while not inducing any extra writes. In addition, SCMFS also does not provide any data integrity guarantees, which is also a key contribution of RMMIO.

Nova-Fortis [41] and NOVA [40] are log-structured file systems, which can atomically write/update data to files by simply attaching a new log. Although it also does not require extra writes on the IO path, the stale logs must be periodically garbage collected. While the old data logs are automatically overwritten by new data in RMMIO, without any additional overheads. Nova-Fortis notices the data integrity problem of NOVA. Thus the backup and deadzone are employed to ensure the data integrity for PM files while incurring dramatic static space overheads. Furthermore, Nova-Fortis also supports taking a full-copy snapshot of a mapped file to prevent user-space scribbles. However, taking a full-copy snapshot is not as efficient as the incremental snapshot of RMMIO. Because incremental snapshots do not need to copy the unmodified data.

In addition to accelerating file IO operations, NV-Heaps [13], Mnemosyne [36], HEAPO [20] and Pangolin [44] are trying to extend the heap of a program from the volatile region to the persistent region, with the help of DAX-style MMIO. These works provide interfaces for programmers to build persistent heaps on PM and manage persistent objects in the persistent heap. All of them ensure the consistency of persistent object operation with the help of logging [13], [20], [36], log-structure [36] and atomic primitives [36], [44]. Despite these works focusing on the management of persistent objects while RMMIO aims to accelerate file IO operations, RMMIO also learned a lot from these works. Since atomic primitives only support 8B data operations, RMMIO employs atomic primitives to guarantee data consistency of 8B metadata. Moreover, RMMIO employs an undo log to ensure the data consistency of file IO operations but without inducing any extra writes. In some perspectives, the PPcache of RMMIO is organized in the form of log-structure style, but it can automatically overwrite the old logs. We note that Pangolin [44] and Pavise [30] also provide protections for data integrity. Although we both employ the 32-bit checksum to detect data corruption, due to the limitation of parity, Pangolin and Pavise cannot recover from the corruption where two pages in two columns are lost at the same time, while the snapshot-based recovery of RMMIO does not have such a limitation. In addition, the parity has to be updated along with the updates of all related data while the snapshot of RMMIO does not need to be frequently updated, which reduces the software overhead of data integrity guarantees.

We also note that there have been hardware integrity guarantees for PM, e.g., Tvarak [24], which can transparently detect corruption and maintain data redundancy for mapped PM regions at the hardware level. Although Tvarak also employs parity as data redundancy for recovery, it has a fine-grained hardware-managed checksum calculation, which is more efficient than the software page-grained checksum of RMMIO. We expect that Tvarak can soon be publicly available to further reduce the performance overhead of data integrity guarantees for RMMIO.

Previous works also proposed PM transactions to provide atomic durability for PM, with the help of hardware logging. LAD [18] proposed a hardware transaction that maintains a persistent buffer in the memory controller and atomically commits the data in the buffer. However, the size of a transaction is limited within the size of the persistent buffer, which cannot help to guarantee the consistency of large IO operations for file systems. HOOP [11] provides atomic durability by using hardware-assisted out-of-place updates. It first writes updates to a persistent out-of-place region (OOP region) and then changes the data mapping from the home region (the original address) to the OOP region to avoid write amplification for logging. Although HOOP can periodically collect the garbage in the OOP region, the maximum size of OOP region (2MB) still restricts the size of a transaction for executing a large IO operation and decreases the performance of write-intensive workloads. SLPMT [43] proposed a selective logging mechanism to remove redundant data from hardware logging, which can increase the performance of transaction execution. The key insight of SLPMT has already been included by RMMIO since RMMIO only persists the data useful to recovery. In addition, users are forced to specify the execution region with Txbegin and Txend, to achieve atomic durability while using hardware transaction. Thus, existing software has to be extensively modified to benefit from hardware transactions. In contrast, RMMIO provides POSIX-like IO interfaces, which can be compatible with existing software.

VI. Conclusion

We have applied RMMIO to persistent memory systems to address the problems induced by DAX-style MMIO, i.e., lack of guarantees to data consistency and integrity. The key contribution of RMMIO is that we achieve a good balance between the efficiency and reliability of MMIO by introducing *PPcache*

in existing persistent memory systems. Based on *PPcache*, RMMIO proposes atomic I/O interfaces for data consistency and incremental snapshots for data integrity. The experimental results show the atomic IO interfaces of RMMIO can provide 2.31x higher bandwidth than existing log-based schemes and the incremental snapshot with auto-snapshot can strictly guarantee data integrity with only 4.5% additional overhead. Moreover, the overall performance of RMMIO exceeds the most popular PM-ware file system, i.e., ext4-DAX, by at most 849%. The evaluation proves that RMMIO can provide MMIO-like performance while ensuring data reliability.

REFERENCES

[1] "Direct access for files," The Linux Kernel Archives. [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[2] "LinkBench: A database benchmark for the social graph," GitHub. [Online]. Available: https://github.com/facebookarchive/linkbench

[3] "MyRocks: A rocksDB storage engine with MySQL," [Online]. Available: http://myrocks.io/

[4] "NVDIMM block window driver writer's guide," PMem.io. [Online]. Available: https://pmem.io/documents/NVDIMM_Driver_Writers_Guide.pdf

[5] "Linux profiling with performance counters," perf. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[6] "GCC, the GNU compiler collection," Wikipedia. [Online]. Available: https://gcc.gnu.org/

[7] GDB, "The GNU project debugger," Sourceware. [Online]. Available: https://www.gnu.org/software/gdb/

[8] J. Axboe, "Flexible i/o tester," GitHub. [Online]. Available: https://github.com/axboe/fio

[9] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," *ACM Sigplan Notices*, vol. 41, no. 6, pp. 158–168, 2006.

[10] J. Bonwick and B. Moore, "ZFS: The last word in file systems," 2007. Available: https://www.snia.org/sites/default/orig/sdc_archives/2008_presentations/monday/JeffBonwick-BillMoore_ZFS.pdf

[11] M. Cai, C. C. Coats, and J. Huang, "Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory," in *Proc. Int. Symp. Comput. Archit.*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 584–596.

[12] J. Choi, J. Hong, Y. Kwon, and H. Han, "Libnvmmio: Reconstructing software io path with failure-atomic memory-mapped interface," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 1–16.

[13] J. Coburn et al., "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 105–118, 2011.

[14] J. Condit et al., "Better i/o through byte-addressable, persistent memory," in *Proc. Symp. Operating Syst. Princ.*, 2009, pp. 133–146.

[15] S. R. Dulloor et al., "System software for persistent memory," in *Proc. Eur. Conf. Comput. Syst.*, 2014, pp. 1–15.

[16] Facebook, "RocksDB: A persistent key-value store for fast storage environments," RocksDB. [Online]. Available: https://rocksdb.org/

[17] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions," in *Proc. USENIX Conf. File Storage Technol.*, 2017, pp. 149–166.

[18] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 466–478.

[19] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, 1950.

[20] T. Hwang, J. Jung, and Y. Won, "HEAPO: Heap-based persistent object store," *ACM Trans. Storage (TOS)*, vol. 11, no. 1, pp. 1–21, 2014.

[21] "Three types of memory," Intel. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/storage/3-types-of-memory-video.html?language=en_US&wapkw=3DXpoint

[22] J. Izraelevitz et al., "Basic performance measurements of the intel optane DC persistent memory module," 2019, *arXiv:1903.05714*.

[23] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "SplitFS: Reducing software overhead in file systems for persistent memory," in *Proc. Symp. Operating Syst. Princ.*, 2019, pp. 494–508.

[24] R. Kateja, N. Beckmann, and G. R. Ganger, "TVARAK: Software-managed hardware offload for redundancy in direct-access nvm storage," in *Proc. Int. Symp. Comput. Archit.*, Piscataway, NJ, USA: IEEE Press, 2020, pp. 624–637.

[25] H. Kumar, Y. Patel, R. Kesavan, and S. Makam, "High performance metadata integrity protection in the wafl copy-on-write file system," in *Proc. USENIX Conf. File Storage Technol.*, 2017, pp. 197–212.

[26] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proc. Symp. Operating Syst. Princ.*, 2017, pp. 460–477.

[27] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan, "Cross-failure bug detection in persistent memory programs," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2020, pp. 1187–1202.

[28] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "PMTest: A fast and flexible testing framework for persistent memory programs," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2019, pp. 411–425.

[29] A. Mathur, M. Cao, and A. Dilger, "ext4: The next generation of the ext3 file system," *USENIX Assoc.*, vol. 32, no. 3, pp. 25–30, 2007.

[30] H. J. Qiu, S. Liu, X. Song, S. Khan, and G. Pekhimenko, "Pavise: Integrating fault tolerance support for persistent memory applications," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2022, pp. 109–123.

[31] S. Raoux et al., "Phase-change random access memory: A scalable technology," *IBM J. Res. Develop.*, vol. 52, no. 4.5, pp. 465–479, Jul. 2008.

[32] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Trans. Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[33] V. Sridharan et al., "Memory errors in modern systems: The good, the bad, and the ugly," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 297–310, 2015.

[34] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Piscataway, NJ, USA: IEEE Press, 2012, pp. 1–11.

[35] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[36] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 91–104, 2011.

[37] M. Wilcox, "Add support for NV-DIMMS to ext4," LWN.net. [Online]. Available: https://lwn.net/Articles/613384/

[38] X. Wu and A. Reddy, "SCMFS: A file system for storage class memory," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–11.

[39] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2019, pp. 427–439.

[40] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. USENIX Conf. File Storage Technol.*, 2016, pp. 323–338.

[41] J. Xu et al., "NOVA-Fortis: A fault-tolerant non-volatile main memory file system," in *Proc. Symp. Operating Syst. Princ.*, 2017, pp. 478–496.

[42] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. USENIX Conf. File Storage Technol.*, 2020, pp. 169–182.

[43] C. Ye et al., "Reconciling selective logging and hardware persistent memory transaction," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 664–676.

[44] L. Zhang and S. Swanson, "Pangolin: A fault-tolerant persistent memory programming library," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 897–912.

[45] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 461–476.

Bo Ding received the B.E. degree in automation from China University of Geosciences, China. He is currently working toward the Ph.D. degree in computer system architecture with Huazhong University of Science and Technology, China. His research interests include persistent memories and file systems. He has published several papers in international conferences and journals including ATC, ICCD, and TACO.

Wei Tong received the B.E., M.E., and Ph.D. degrees in computer science and technology from Huazhong University of Science and Technology (HUST), China. She is currently an Associate Professor with Wuhan National Laboratory for Optoelectronics, HUST. Her research interests include computer architecture, non-volatile memory & storage, and software-defined storage. She has more than 20 publications in international conferences and journals including IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, *ACM TACO*, ISCA, DAC, DATE, ICCAD, ICCD, ICPP, MSST, and LCTES.

Yu Hua (Senior Member, IEEE) received the B.E. and Ph.D. degrees from Wuhan University, China. He is currently a Professor with Huazhong University of Science and Technology, China. His research interests include cloud storage systems, file systems, non-volatile memory architectures, etc. His papers have been published in major conferences and journals, including IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, OSDI, FAST, MICRO, ASPLOS, VLDB, USENIX ATC, SC, and HPCA.

Zhangyu Chen received the B.E. degree in computer science and technology from Huazhong University of Science and Technology (HUST), China, where he is currently working toward the Ph.D. degree. His research interests include persistent memories and debugging. He has published several papers in international conferences and journals including ASPLOS, ATC, DAC, TACO, etc.

Xueliang Wei received the B.E. and Ph.D. degrees in computer science and technology from Huazhong University of Science and Technology (HUST), China. He is currently a Postdoctoral Researcher with HUST, China. His research interests include non-volatile memory, persistent memory, crash consistency, and memory security.

Dan Feng (Fellow, IEEE) received the B.E., M.E., and Ph.D. degrees in computer science and technology from Huazhong University of Science and Technology (HUST), China. She is currently a Professor and Dean with the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 80 publications to her credit in journals and international conferences, including IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, JCST, USENIX ATC, FAST, ISCA, ICDCS, HPDC, SC, ICS, and ICPP.