# RMMIO: Enabling Reliable Memory-Mapped I/O for Persistent Memory Systems

Bo Ding, Wei Tong, Yu Hua, Zhangyu Chen, Xueliang Wei, Dan Feng
*Wuhan National Laboratory for Optoelectronics,*
*Huazhong University of Science and Technology, Wuhan, China*
Email:{boding, tongwei, csyhua, chenzy, xueliang_wei, dfeng}@hust.edu.cn
*Corresponding author: Wei Tong*

2022 IEEE 40th International Conference on Computer Design (ICCD) | 978-1-6654-6186-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCD56317.2022.00110

***Abstract*—The byte-addressable persistent memory (PM) is coming to be the next-generation storage device for better I/O performance. As the traditional I/O path is too lengthy to drive PM featuring low latency and high bandwidth, prior works have 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 user-space scribbles and non-atomic I/O interfaces, termed reliability issues. Since existing reliability schemes cause significant extra overheads, we propose RMMIO, an efficient user-space library that provides reliable memory-mapped I/O interfaces for PM systems. RMMIO achieves a good balance between efficiency and reliability by introducing a memory-mapped cache layer upon kernel file systems. The cache layer accelerates I/O requests and carries the file system's responsibility for data reliability by data isolation. In addition, RMMIO further employs lightweight snapshots and efficient atomic I/O interfaces to guarantee the integrity and consistency of the data in the cache layer at low costs. The experimental results show that RMMIO achieves 8.49x higher throughput than ext4-DAX and 2.31x higher throughput than state-of-the-art MMIO-based schemes for PM while ensuring data reliability.**

## I. Introduction

Non-volatile memory (NVM) technologies, such as Phase Change Memory, and Resistive RAM, achieve the advantages of both DRAM (e.g., low latency, high bandwidth, and byte addressability) and disk (e.g., persistency). Persistent memory (PM) powered by NVMs enables the durability of data in memory space. These remarkable PM features significantly reduce the hardware I/O overhead, but stick out the software overhead in traditional file systems. To simplify the software I/O stack, recent PM-aware file systems, e.g., PMFS [4], NOVA [10], and ext4-DAX [8], leverages the DAX technology to bypass page cache. As a result, PM-aware file systems can directly access the data in PM. However, DAX-enabled file systems still suffer from the complex indexing structure and redundant VFS I/O path. To further simplify the I/O path to PM, SplitFS [5] and Libnvmmio [3] 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 kernel's protections. Specifically, the file mapped into user address space could be easily overwritten with arbitrary data, called *scribbles*, due to bug-prone software and unexpected hardware errors. In PM systems, scribbles could be more dangerous than those in DRAM systems because of recently discovered bugs in PM programming. In addition, PM only guarantees the atomicity for 8-byte write: any write over 8 bytes could be incomplete upon a system crash. Thus even regular I/O requests can result in data inconsistency due to the lack of atomic I/O interfaces. Moreover, data corruptions caused by scribbles and non-atomic I/Os can permanently corrupt data in PM and exist even after the system restarts, which dramatically threatens the reliability of data in PM.

To the best of our knowledge, existing works [2]–[7], [11] do not fully guarantee the reliability of the mapped data or do not achieve a good balance between performance and reliability. To address these issues, we propose **R**eliable **M**emory-**M**apped **IO** (RMMIO), a user-space library that efficiently guarantees both data consistency and data integrity for the memory-mapped data.

To minimize the extra overheads caused by reliability guarantees, RMMIO inherits the matured protection mechanisms of kernel file systems [2], [7], [11] by keeping all files in kernel space, without mapping them into user address space. Instead, we use a large contiguous persistent memory region as a cache layer, called *Persistent Page Cache* (*PPcache*), to accelerate I/O operations. The cache directly resides in user space, so access to PPcache is as fast as MMIO to the mapped file. Since *PPcache* is persistent, I/O requests arriving at *PPcache* are treated as completed and 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 cached in PPcache. To guarantee the consistency of *PPcache*, RMMIO provides atomic I/O interfaces by employing WAL (Write-Ahead Log). Specifically, RMMIO builds a two-level structure for reusing old data as undo log, which significantly decreases the write amplification in existing WAL. 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 target file from the unaffected snapshots. To simplify the software overhead of snapshots, RMMIO implements incremental snapshots that only record updates to a file, which is much more efficient than full-copy snapshots by avoiding unnecessary data movements.

The experimental results show that RMMIO gains up to 8.49x higher throughput than ext4-DAX [8] and achieves 2.31x higher throughput that the state-of-the-art MMIO-based schemes [3] in write-intensive workloads, while guaranteeing data reliability. In the evaluation of real-world applications, RMMIO also outperforms all existing PM-aware file systems.

This work was sponsored by National Natural Science Foundation of China under Grant 61832007, Grant 61821003 and Grant 62172178.

2576-6996/22/$31.00 ©2022 IEEE
DOI 10.1109/ICCD56317.2022.00110

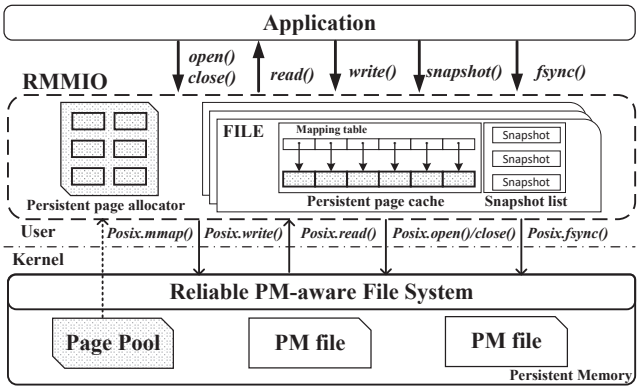



Fig. 1. Overview of RMMIO.

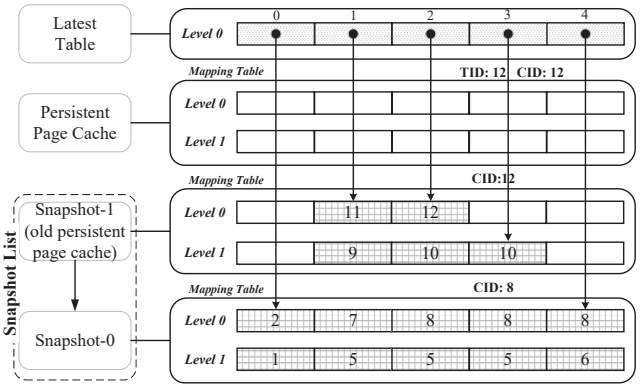



Fig. 2. The Organization Structure of PPcache and Snapshots.

## II. RMMIO

The design goal of RMMIO is to provide efficient reliability guarantees (i.e., data consistency and data integrity) for MMIO in PM. Thus RMMIO offloads the protections for PM files to the underlying file system to reduce software overheads for reliability. However, the kernel-protected files cannot be mapped into user space, which results in compromised I/O performance. Therefore, RMMIO employs a memory-mapped cache region in PM to accelerate the updates to a PM file. As a result, RMMIO only needs to take charge of the cached data. The cache is organized as a unique data structure, called *FILE*, for every opened file, as shown in Figure 1. *FILE* maintains a persistent page cache (*PPcache*) and a *snapshot list* for every file: ***PPcache*** buffers the most recent updates to the related file; ***snapshot list*** records the historical updates to the file, which provide necessary data redundancy to recover RMMIO from possible data corruptions.

For fast indexing, *PPcache* and *snapshot list* are indexed by ***mapping tables*** located in DRAM, which is designed for continuous indexing in PM. In addition, since memory resources are valuable, we never preallocate physical persistent memory pages for the *PPcache*. All memory resources are dynamically allocated/recycled by the ***persistent page allocator***.

RMMIO currently provides six main interfaces for MMIO, i.e., ***read(), write(), snapshot(), open(), close(), fsync()***., whose usages are aligned with *POSIX* APIs except for *snapshot()*. We will present more details about *snapshot()* in Section II-C.

### A. Guarantee Data Consistency with Two-Level Page Cache

To guarantee data consistency in the *PPcache*, RMMIO has to provide atomic I/O interfaces to avoid breaking the consistent state of *PPcache*. Since modern processors support only 8-byte atomic writes for memory [10], RMMIO has to make efforts on providing atomicity for writes with arbitrary lengths. However, existing schemes, like WAL [3], [5] and log-structuring [10], induce extra writes and additional software overheads for garbage collection, respectively.

Since 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 a novel WAL mechanism that does not require additional writing by reusing old data as undo log. As shown in Figure 2, RMMIO builds a two-level *PPcache* to maintain two versions of data for each page, i.e., the old data and the new data. When a writer thread writes data to *PPcache*, the coming data overwrites the old data but remains the new data as the undo log. Once the ongoing write operation is interrupted by a system crash, RMMIO abandons the incompleted write and recovers from the undo log. To implement this strategy, RMMIO sets up a switch for every page to mark the new data. Each time a write operation is completed, the switch will be toggled to indicate where the new data is.

To guarantee strict data consistency, every write has to be committed atomically. However, it is hard for RMMIO to mark variable-length data across several pages as committed simultaneously due to the limitation of 8-byte atomic write. RMMIO addresses this problem by a timestamp-based commit mechanism, inspired by Libnvmmio [3]. The timestamp-based commit mechanism includes two main components, i.e., a logical timestamp called TID (Transaction ID) and a global timestamp called CID (Committed ID). Since RMMIO divides each write into several page writes, pages within a write transaction will be marked with the same TID to indicate when they are written. To figure out which page has been committed, RMMIO uses CID to record the timestamp of the latest transaction. 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. The inconsistent pages can be easily identified by comparing their TID with the CID.

### B. 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. The native MMIO cannot safely handle multi-thread workloads due to the lack of threads isolation.

To guarantee thread safety, RMMIO employs a reader or writer lock for thread isolation. However, we note that the file-grained lock used in VFS blocks concurrent operations on a shared file [9], 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. 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 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.

RMMIO also ensures thread safety with atomic primitives. As locking needs to fall into the kernel, locking for every atomic 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 that the CID is updated in the order of the TID of different threads. The atomic CAS makes sure that write transactions from different threads will be committed one by one.

### C. *Lightweight Incremental Snapshot*

Scribble is another essential issue of RMMIO because scribbles can break the data integrity of *PPcache*. However, scribbles are inevitable and unpredictable. The only thing that we can do against scribbles is to detect and recover the corrupted data once a scribble happens. RMMIO detects possible data corruptions by examining the CRC32C checksum for every page. The page-grained checksum minimizes the software overhead upon checking the integrity of data. The corrupted data can be recovered from snapshots in RMMIO, which provides a consistent backup for the corrupted data.

RMMIO implements an incremental snapshot that only holds the updates to a file, which is more efficient than the full-copy snapshot in Nova-fortis [11]. The incremental snapshot is built on the basis of *PPcache*. Once a user takes a snapshot, RMMIO directly converts the current *PPcache* into a snapshot and adds the snapshot to the end of the *snapshot list*, as shown in Figure 2. In addition, RMMIO further builds an empty *mapping table* to be the new *PPcache*. As a result, *snapshot()* in RMMIO only needs to initialize a new *mapping table* without any data copying.

We also note that taking snapshots in RMMIO may cause the performance decline of reads. Because snapshots may still buffer the latest data of the target file, which increases the difficulty of *read* to query the latest data. As *Page0* shown in Figure 2, reads must traverse all snapshots to find the latest data. Such an inefficient traversal operation extends the I/O path of RMMIO read, which goes against RMMIO's design philosophy. To avoid traversing these snapshots, RMMIO builds a *latest table* to store the pointer to the latest data for each page. As taking a snapshot in RMMIO does not move any data, the pointer to the latest data will always be constant. So we can always get the latest data by accessing the *latest table* with the time complexity as O(1). The latest table will be updated along with updates to *PPcache* and reset after *fsync()*.

The snapshots help RMMIO to recover from possible data corruptions. Different from backup schemes, snapshots in RMMIO leverage the historical updates to a file as data redundancy. Once the latest data is corrupted, RMMIO rolls back to a historical version by abandoning all affected data. The affected data includes the pages residing in the same *PPcache*/*snapshot* as the corrupted one. As shown in Figure 2, once scribbles happen on *data11*, RMMIO rolls backs back to CID:8. In this case, *data7* can be the successor of *data11* for *Page1*. For pages that have never been updated, RMMIO does not need to worry about them since they are well protected by kernel file systems. Note that the protected file also works as an RMMIO snapshot. In the worst scenario, RMMIO can abandon all updates yet still preserves a file's consistency and integrity, preventing additional disastrous effects.

## III. Evaluation

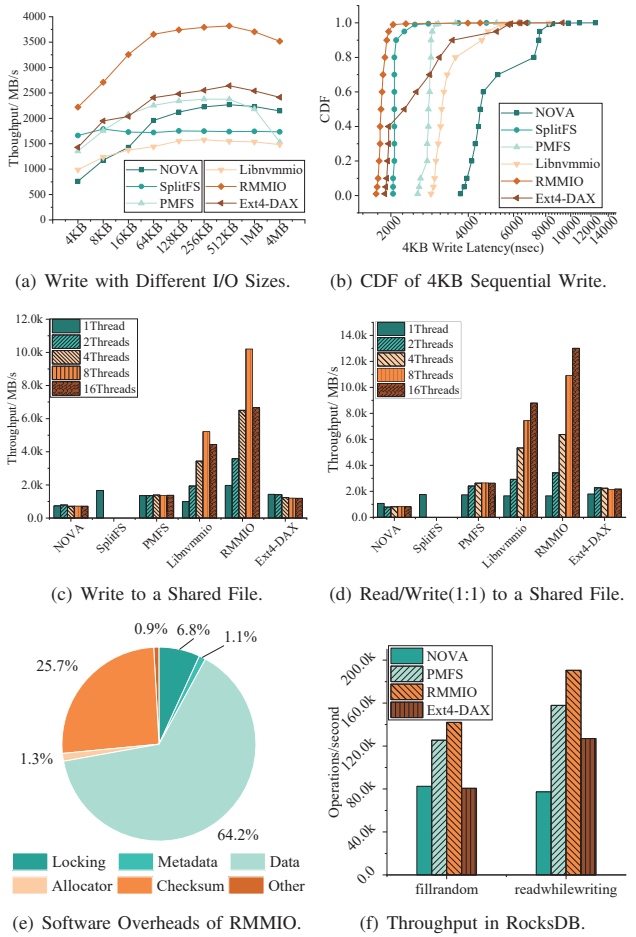
We implement and evaluate the proposed designs on a server equipped with a 2-socket Intel Xeon 6230R, 12 * 16GB DDR4 and 12 * 128GB Optane DC Persistent Memory configured as App Direct Mode with interleaving. We note that all evaluated works are not NUMA-optimized. So we employ *numactl* to bind all working threads and memory regions to the same NUMA node. Finally, our evaluation is performed on Linux kernel 4.13 with FIO [1] as microbenchmark.

**Bandwidth.** We evaluate RMMIO's write performance with variable I/O sizes, as shown in Figure 3(a). 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 3,818MB/s, which is over 44% higher than that of ext4-DAX. Compared with SplitFS (strict mode), RMMIO obtains performance gains up to 2.19x owing to the lightweight log strategy. With the increase of 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 the tree-like index structures, e.g., extent tree in ext4-DAX, radix tree in Libnvmmio, for contiguous indexing multiple pages.

**Latency.** As the *Cumulative Distribution Function* (CDF) of 4KB write shown in Figure 3(b), 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 (3,728ns), PMFS (2,544ns) and Libnvmmio (2,864ns). Because RMMIO permanently buffers the data in the *PPcache* to avoid kernel overheads. In addition, the logging strategy of RMMIO does not introduce any extra write to PM, which further reduces the latency of atomic I/O.

**Reliability overheads.** In Figure 3(e), we measure and quantify five major software overheads of RMMIO. The two main components, i.e., data copy and checksum calculation, take up more than 80% of the execution time in RMMIO write. Although RMMIO spends over 20% of time on checksum calculation, the execution time of data copy is still up to 64% in write, which is much more efficient than ext4-DAX.

**Scalability.** The native MMIO is efficient for concurrent executions owing to byte-addressable PM. To examine the scalability of RMMIO, we further evaluate RMMIO and its competitors with concurrent 4KB-sequential writes and 4KB-read-write mixed I/O to a shared file. Since SplitFS does




(a) Write with Different I/O Sizes. (b) CDF of 4KB Sequential Write.

(c) Write to a Shared File. (d) Read/Write(1:1) to a Shared File.

(e) Software Overheads of RMMIO. (f) Throughput in RocksDB.

Fig. 3. Performance Overview of RMMIO.

not support concurrent execution in strict mode, we only present the single-thread performance of SplitFS. According to Figure 3(c), the maximum bandwidth of RMMIO concurrent writes exceeds that of NOVA, PMFS, Libnvmmio, and ext4-DAX by 14.07x, 7.50x, 1.96x, and 8.49x, respectively. The huge performance gain is because RMMIO employs the page-grained lock for concurrency control instead of the VFS's file-grained lock. The advantage of page-grained lock is evident in the concurrent writes to a shared file since multi-thread writes could be fully paralleled. We also note that the write performance decreases when the thread number exceeds 8. The analysis of *perf* indicates the high lock contention to the same page, which only happens in thread-intensive workloads. In read-write mixed I/O, the performance advantage of RMMIO is more pronounced than that in the write-only workloads, considering that reader threads can share a page at the same time. According to Figure 3(d), RMMIO gains up to 15.94x higher throughput than NOVA and 5.97x higher throughput than ext4-DAX. In addition, due to the higher single-thread performance, RMMIO also outperforms Libnvmmio by 1.47x, which also employs a page-grained lock. Given the 4KB-aligned I/O is nearly twice as fast as the 4KB-unaligned I/O (Section II-B), RMMIO outperforms Libnvmmio by building a 4KB-aligned data region (i.e., *PPcache*).

**Real-world Application: RocksDB.** We adapt RMMIO to RocksDB and evaluate RMMIO with two benchmarks in *db_bench*, i.e., *fillrandom* and *readwhilewriting*. Both two benchmarks contain 10,000,000 entries with 16B key and 1024B value. As shown in Figure 3(f), the throughput of RMMIO outperforms ext4-DAX by 1.72x with *fillrandom*. Because RMMIO accelerates the compaction in RocksDB, which blocks the foreground insert operation. According to the statistics of *db_bench*, RMMIO only triggers two times of *level0_slowdown_with_compaction* while ext4-DAX triggers up to nine times. Figure 3(f) indicates that RMMIO gains up to 2.46x throughput of NOVA, 1.2x higher throughput than PMFS, and 1.49x higher throughput than ext4-DAX, with *readwhilewriting*. The advantage of RMMIO becomes greater with *readwhilewriting*. Such a great advantage does not come from the fine-grained reader/writer lock of RMMIO since we only use single-thread compaction in this evaluation. As *readwhilewriting* is a mixed read-write benchmark, the outstanding performance of RMMIO read contributes to the high throughput of *readwhilewriting*. Since RMMIO reads are routed to the *PPcache* instead of the underlying file system, RocksDB reads are accelerated by MMIO.

## IV. Conclusion

We have applied RMMIO to PM-aware file systems to address the reliability issues induced by MMIO in PM, i.e., the lack of consistency and integrity guarantees for the mapped data. The key contribution of RMMIO is that we achieve a good balance between the efficiency and reliability of MMIO by introducing *PPcache*. The experimental results indicate that RMMIO can accelerate existing IO-intensive workloads by employing MMIO while considering the reliability of data. Moreover, RMMIO also shows better scalability and reliability than prior MMIO-based works.

## References


[1] Jens Axboe. Flexible i/o tester. https://github.com/axboe/fio.

[2] Jeff Bonwick and Bill Moore. Zfs: The last word in file systems, 2007.

[3] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing software io path with failure-atomic memory-mapped interface. In *Proc. USENIX ATC*, pages 1–16, 2020.

[4] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proc. EuroSys*, pages 1–15, 2014.

[5] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proc. SOSP*, pages 494–508, 2019.

[6] Avantika Mathur, Mingming Cao, and Andreas Dilger. ext4: the next generation of the ext3 file system. *Usenix Association*, 32(3):25–30, 2007.

[7] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[8] Matthew Wilcox. Add support for nv-dimms to ext4. https://lwn.net/Articles/613384/, 2014.

[9] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proc. ASPLOS*, pages 427–439, 2019.

[10] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. FAST*, pages 323–338, 2016.

[11] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proc. SOSP*, pages 478–496, 2017.