MPFS: A Scalable User-space Persistent Memory File System for Multiple Processes

Bo Ding, Wei Tong, Yu Hua, Yuchong Hu, Dong Huang, Qiankun Liu, Zhangyu Chen, Xueliang Wei, Dan Feng
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
Email: {boding, tongwei}@hust.edu.cn, ***Corresponding author:*** *Wei Tong*

[1] ***Abstract*—Persistent memory (PM) leveraging memory-mapped I/O (MMIO) delivers superior I/O performance, leading to the development of user-space PM file systems based on MMIO. While effective in single-process scenarios, these systems encounter challenges in multi-process environments, such as performance degradation due to repeated page faults and cross-process synchronizations, as well as a large memory footprint from duplicated paging structures. To address these problems, we propose a Multi-process PM File System (MPFS). MPFS builds a *shareable page table* and shares it among processes, avoiding building duplicate paging structures for distinct processes, thereby significantly reducing the software overhead and memory footprint caused by repeated page faults. MPFS further proposes a *PGD-aligned (512GB) mapping* method to accelerate page table sharing. Furthermore, MPFS provides a cross-process memory protection mechanism based on the *PGD-aligned mapping*, ensuring multi-process data reliability with negligible overheads. The experimental results show that MPFS outperforms existing user-space PM file systems by 1560% in multi-process scenarios.**

***Index Terms*—File System, Persistent Memory, Memory Management**

I. Introduction

Data-intensive applications [11], [12] demand high-capacity memories and storage devices with superior I/O performance. The capacity of DRAM is limited by current manufacturing, and block I/O devices (e.g., SSDs, disks) are significantly slower than DRAM. Persistent Memory (PM) [13], [22], [23] bridges this gap, offering TB-scale capacity with DRAM-like performance [14], [30] and enabling rapid data persistence, which shortens the system I/O path. Additionally, PM is byte-addressable like DRAM, allowing direct access via load/store instructions. PM provides an opportunity for better performance in data-intensive applications. However, since PM's hardware latency is significantly lower than existing block devices, overheads caused by traditional kernel I/O software (VFS, tree-like block index, and so on) account for most of the PM I/O [15]. To maximize PM's performance, existing works [7], [16], [24] have advocated for using Memory-Mapped I/O (MMIO) in place of the lengthy kernel I/O path. MMIO can directly access PM with load/store instructions in user space. Accessing PM with MMIO only incurs the overhead for address translation via the Memory Management Unit (MMU). Thus, MMIO-based file systems (also known as user-space file systems, e.g., SCMFS [26], SplitFS [16], Libnvmmio [7], ctFS [18]) outperform existing kernel PM file systems [10], [25], [28] by over 100%, as shown in Figure 3(a).

[1]This work was supported by the Young Scientists Fund of the National Natural Science Foundation of China under Grant 62302182.

Despite their superior I/O performance, existing MMIO-based file systems struggle to perform efficiently in multi-process scenarios for the following reasons:

Repeated page faults. Modern operating systems assign each process a separate address space, requiring a process-specific page table to map virtual addresses to physical pages. To make PM globally available, PM must be mapped into every process's address space using *mmap()* (allocating a virtual address area), leading to separate page faults (interrupt requests to build paging structures for mapping virtual addresses to physical pages) for every process. This increases the cost of accessing PM with MMIO in multi-process scenarios. Previous work [15] demonstrates that mapping 3TB of PM incurs page faults time ranging from *5.37s* with 2MB (huge page) granularity to *2750s* with 4KB (normal page) granularity. Although not all processes need the entire PM mapped, the time to repeatedly map PM in many processes remains significant. Additionally, the memory footprint of the page table for PM mapping ranges from *12MB* (huge) to *6.01GB* (normal) per process. Due to unpredictable fragmentation [15], it is challenging to map PM entirely at huge page granularity. Thus, the memory footprint of page tables for PM mapping can be very large.

Multi-process Inconsistency. Since each process has a separate PM mapping, changes to PM mappings are not immediately visible to all processes, leading to inconsistent reads/writes in user-space file systems. For instance, in ctFS [18], if one process modifies a file's start address, others may still use the old address, causing different processes to achieve various results and potentially tampering with the data. To prevent this, all processes must synchronize their changes to PM mappings, which increases software overheads and blocks parallel read/write operations. Our evaluation shows that in multi-process scenarios, ctFS's read performance drops to about 1/15 of that in single-process scenarios.

To address the above problems, we propose a scalable user-space PM file system that accommodates multiple processes, termed **M**ulti-process **PM** **F**ile **S**ystem (MPFS).

MPFS introduces a shareable page table to enable efficient sharing of PM mappings among multiple processes. This eliminates the need for duplicate paging structures and reduces the software overhead and memory footprint caused by repeated page faults. The shareable page table provides a unified PM mapping view for all processes so that one process can directly access the PM mappings built by another process, eliminating the need for cross-process synchroniza-

978-3-9826741-0-0/DATE25/© 2025 EDAA

tions. MPFS also proposes a ***PGD-aligned*** (512GB-aligned) mapping method to build a well-structured shareable page table, which can be easily mounted to any process's page table by only copying the root of shareable page table. Owing to the aligned paging structures, this approach also reduces paging structure fragmentation and lowers address translation overheads. Additionally, sharing MPFS among processes also induces serious reliability issues due to flaws in single-process memory protection scheme [21]. By utilizing the PGD-aligned mapping, MPFS provides a lightweight ***cross-process memory protection*** mechanism to ensure multi-process data reliability with negligible overheads.

Experimental results show that MPFS significantly reduces software overhead from multi-process page faults and synchronizations, outperforming existing user-space file systems by up to 1560% in multi-process scenarios. Even in single-process cases, MPFS improves performance by about 20% compared to state-of-the-art systems.

II. Background and Motivation

User-space PM File Systems. User-space PM file systems are specialized for PM, leveraging its DRAM-like interface to map PM data into user address space, allowing direct access via load/store instructions, thus outperforming traditional kernel file systems. SCMFS [26] was the first such system, storing files in contiguous areas, enabling sequential access guided by a start address. It uses the page table to index data blocks, which is faster than traditional tree-like indexes [6], [25]. However, SCMFS struggles with file relocation when files expand to overlap other files. ctFS [18] addresses this by using *pswap()*, a system call that swaps the paging structures of two areas, exchanging their mappings without moving data, reducing relocation overhead. In addition to building file systems entirely in user space, SplitFS [16] and libnvmmio [7] use MMIO to accelerate I/O operations for kernel file systems by intercepting I/O system calls (e.g., *read()/write()*) and replacing them with MMIO interfaces. However, these approaches require temporary file mappings each time a file is accessed, leading to overhead from repeated mappings and page faults. DaxVM [1] addresses this by permanently storing the page table that maps the file but still requires remapping when the file size changes, necessitating rebuilding the page table and causing a performance penalty. Though the above-mentioned systems benefit from MMIO, they are all limited while working in multi-process scenarios. Since they all manage the mappings of files in a process-specific manner, the mappings of PM cannot be shared among processes, causing repeated *mmap()* and page faults for accessing the same data in different processes. In SplitFS, Libnvmmio, and DaxVM, the overhead becomes greater because they manage PM mappings at file granularity, which means they have to map every file in every process. Moreover, when the file mapping changes, the updates to the page table have to be synchronized among processes to avoid inconsistent reads/writes to the file. However, none of them provide a solution to efficiently synchronize mappings among processes.

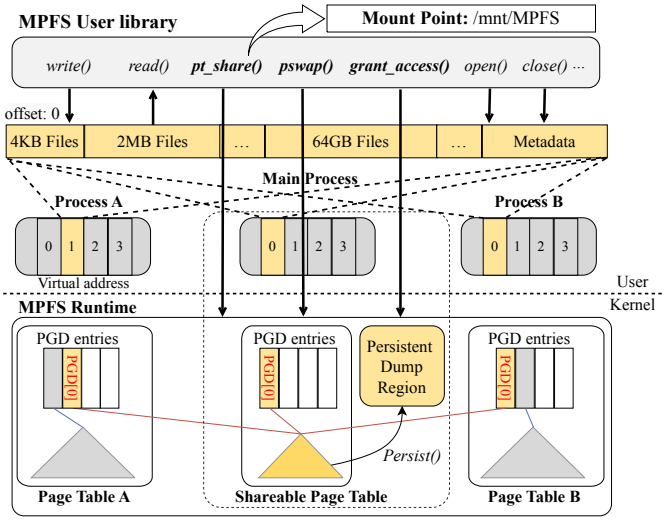


Fig. 1. The overview of MPFS.

Motivation. User-space file systems should be scalable in multi-process scenarios and be available to all processes, just like the kernel file system. However, the limitations of MMIO prevent multiple processes from fully benefiting from user-space file systems. For example, ctFS cannot be mounted to *"/root"* since the root directory must be shared by all processes across different users and daemons. This also complicates tasks requiring file sharing among multiple processes, such as running multiple instances of MySQL [4]. Additionally, common shell operations like *ls*, *cd*, *mkdir*, *cat*, *find*, and *locate*, are not well supported by existing user-space PM file systems due to the long stall time caused by repeated page faults and synchronizations.

III. MPFS Based on Shareable Page Table

We introduce MPFS, a user-space PM file system designed for multi-process scenarios. As shown in Figure 1, the MPFS user library stores files in contiguous areas of the user address space, offering POSIX compatible interfaces (e.g., *write()*, *read()*, *open()*, *close()*). Specifically, *write()* and *read()* are implemented by moving data between user buffer and a file area with *memcpy()*, similar to SCMFS [26] and ctFS [18]. Differently, MPFS's interfaces are cross-process, enabling files to be correctly addressed across different processes via offsets relative to MPFS's start address, even if they are mapped to different addresses in different processes. The offset addressing is built based on a ***shareable page table***, which guarantees that different addresses with the same offset can be mapped to the same physical address. However, building a shareable table is challenging because the page table is process-specific, and PM mappings in existing systems cannot be directly shared among processes due to the following reasons:

Arbitrary Mapping. Existing MMIO-related works, like SplitFS [16], Libnvmmio [7], and ctFS [18], follow the mapping rules set by the memory management subsystem (4KB-aligned) or the NVDIMM driver (2MB-aligned) without imposing additional restrictions on PM address space. As a result, PM can be mapped into arbitrary areas in the user space,

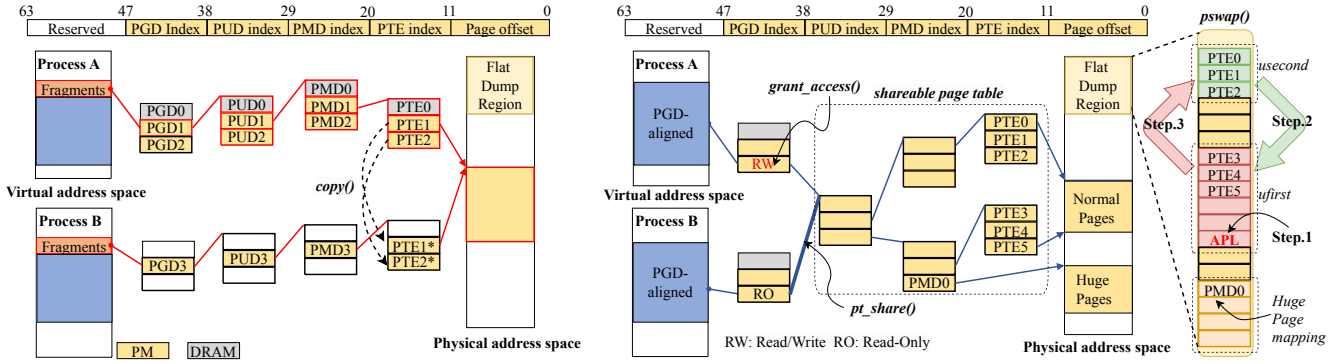


Fig. 2. The PGD-aligned page table and fragmented paging-structure entries.

leading to an irregularly shaped page table. This irregularity means that paging structures mapping PM may also include DRAM mappings since existing systems allocate paging structures in groups of 512, not individually. For example, as shown in the red square on the left of Figure 2, the root of the paging structures for fragments is *PGD1*, pointing to a group of PUD entries. However, only *PUD1* and *PUD2* map PM, while *PUD0* maps DRAM, which cannot be shared, as does *PMD0*. Thus, these fragmented paging structures cannot be shared by other processes. To share the page table only for PM mappings, each fragmented page's faults must be handled individually by allocating new PUD and PMD groups and linking them to PTE entries with the page's physical address. In the worst case, if a PTE group contains DRAM mappings, a new PTE group must be allocated, and PTEs copied individually to prevent unintentional sharing of private DRAM data across processes. This approach necessitates traversing the entire page table, leading to significant performance degradation. Additionally, different processes may allocate different address spaces, making the page table sharing even more complex.

Scattered Mappings. Furthermore, existing systems [1], [7], [16] map PM at file granularity. Files are mapped separately, leading to separate and non-adjacent paging structures in the page table. As a result, the paging structures for PM mapping are scattered throughout the page-table tree, complicating the process of sharing page tables. Even if adjacent structures are initially allocated, they may still need to be relocated or rebuilt as files grow or shrink, as discussed in Section II. Therefore, file-granularity PM mapping is not suitable for efficient page table sharing among processes.

A. Shareable Page Table with PGD-aligned Mapping.

To address these challenges, MPFS uses device-granularity mapping to map PM into a large contiguous area with mandatory alignment constraints on user space addresses. This approach consolidates the paging structures for PM into a single subtree under the process's page table. As shown in the right of Figure 2, this subtree is constructed by PUDs, PMDs, and PTEs, with a PGD entry as root which stores the physical address of the subtree. To create a well-structured subtree, MPFS introduces a new *mmap()* flag, ***MAP_PGD_ALIGNED***, which ensures that the virtual address returned by *mmap()* is strictly PGD-aligned, meaning it starts at an integer multiple of 512GB. MPFS also guarantees that the allocated region size is a multiple of 512GB, confining PM paging structures to one or more subtrees. DRAM mappings are excluded from these subtrees, even if the aligned region remains empty. This doesn't waste physical memory and is also not a significant issue, as 64-bit systems provide up to 256TB of address space per process, far exceeding the physical memory capacity in current systems. In MPFS, the PGD-aligned subtree for PM mappings is termed the ***shareable page table***. The shareable page table can be mounted to any process's page table by copying its root to the process's PGD entries via *pt_share()* (taking 7000 to 10000 nanoseconds), as shown in Figure 2. Since the shareable page table stores physical addresses based on the offset of the address, the same offset within MPFS in different processes will map to the same physical address. This ensures that the same PM data can be accessed by different processes using the same offset without fixing the address in multiple processes. Since each subtree can map only 512GB of PM, MPFS uses a multi-tree solution to extend the PM mapping range. The kernel runtime currently contains 6 subtrees, mapping up to 3TB of PM, and can be extended for larger mappings.

The shareable page table can be shared by multiple processes simultaneously. The concurrent page faults from different processes are handled by existing systems via locking. Additionally, MPFS provides an atomic *pswap()* to ensure concurrency safety when relocating paging structures across multiple processes, further detailed in Section III-B. MPFS initializes and manages the shareable page table in the main process shown in Figure 1. MPFS follows Linux's lazy page fault strategy, which populates paging structures only when the page is accessed, amortizing page fault overheads across multiple processes and reducing MPFS's memory footprint.

B. Persisting Shareable Page Table in Flat Dump Region.

MPFS must ensure the persistency of the volatile page table to rebuild file mappings after a reboot. Existing systems [1], [18] use a persistent page table, synchronizing every modification with the volatile page table. However, this approach requires traversing multiple levels (PGD, PUD, PMD, PTE), resulting in at least three 8B-sized PM accesses per update, which is inefficient for PM due to performance degradation of sub-256B accesses [27]. Thus, MPFS introduces a ***flat dump***

region, as shown in the right of Figure 2, to persist only the last-level entries (PTE for normal pages, PMD for huge pages) of the shareable page table and indexes PTE/PMD entries in the flat dump region using the offset relative to MPFS's start, avoiding persisting upper-level structures. Upper-level entries (PGD, PUD) can be easily rebuilt after reboot because they do not directly store the physical addresses of pages but work as intermediate nodes to index PTE/PMD. Though the flat dump region needs more capacity than persisting the page table tree when PM is not fully mapped, it significantly accelerate the persistency of page table. Storing 512GB mappings (for every subtree) requires 1GB which is only 0.03% of PM capacity.

MPFS also provides an **atomic *pswap()*** to ensure the consistency of the shareable page table even if a system crash occurs during *pswap()*. The atomic *pswap()* can atomically swap the PTE/PMD entries in the two areas of the dump region to move the file to a larger area. The key of atomic *pswap()* is an 8B *Atomic Persistent Log (APL)*, which can be written to PM atomically because PM supports 8B atomic write. The APL contains 4 fields: (1) The *magic word* (2B) to identify whether it is a valid entry; (2) The *status* (1B) to indicate the status of current *pswap()*. (3) The *npgs_enum_index* (1B) to record the number of normal pages to be swapped in the *pswap()*. (4) The *usecond_index* (4B) that records the location of the second area (usecond) participating in the *pswap()*. Since 8B is too small to contain the locations of the two swapped areas, MPFS places the APL at the end of the first area participating in the *pswap()* to indicate the locations of *ufirst*, as shown in Figure 2. Since the *ufirst* is always larger than the *usecond*, the last entry of the *ufirst* will always be empty without overlapping any existing entry. Thus, MPFS can calculate the location of the *ufirst* by the address of the APL and *npgs_enum_index* in the APL. The atomic *pswap()* follows the three steps shown in Figure 2. The status of the *pswap()* will be updated to the APL immediately after every step. Therefore, whenever a system crash occurs during the *pswap()*, the recovery process can easily identify the status of the *pswap()* and continue the *pswap()* from the last step. In addition, since APL can be atomically updated, it also provides concurrency safety for *pswap()* in multiple processes. Any concurrent *pswap()* to the same region will be blocked by the APL. The atomic *pswap()* provides both consistency guarantees and concurrency safety for MPFS.

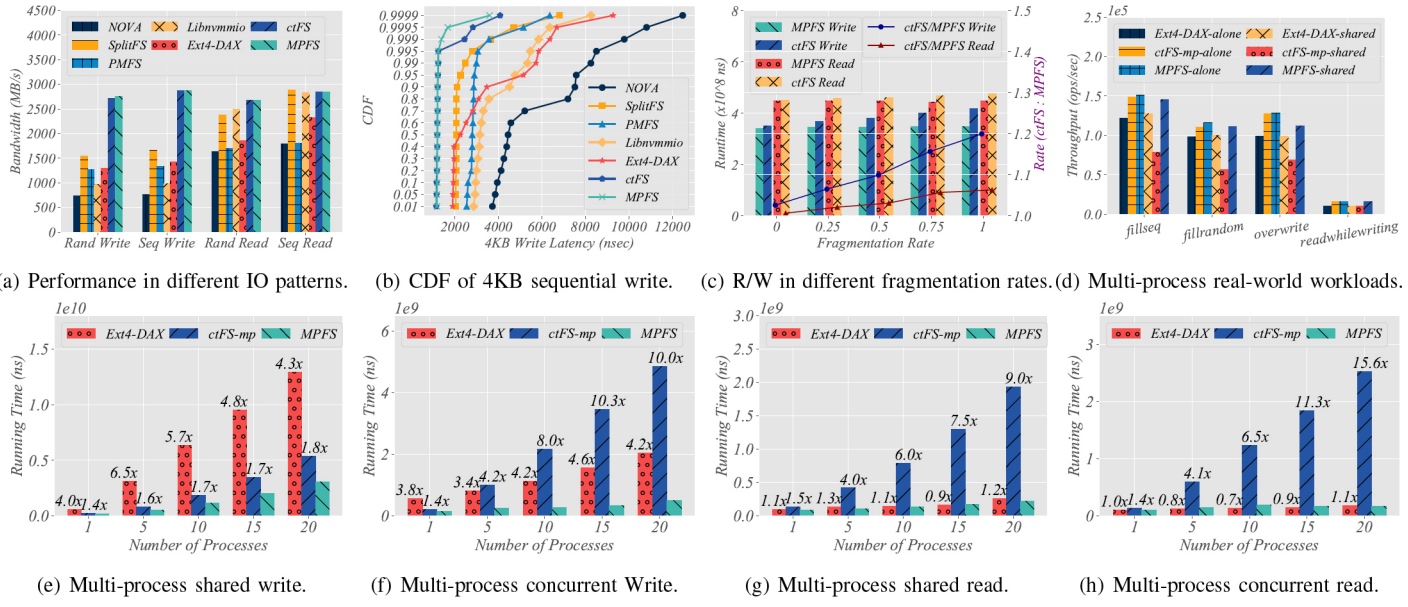
C. Optimized pswap() based on PGD-aligned Mapping.

The PGD-aligned mapping also optimizes the *pswap()* operation and reduces the addressing overhead of MPFS. Existing *pswap()* implemented by ctFS [18] suffers from the fragmentation of paging structures due to the irregularly shaped page table. Fragments divide huge pages into a large number of normal pages, leading to many more paging structures that need to be moved during *pswap()* and increasing the overheads of addressing. By building a PGD-aligned PM mapping, MPFS can implement an optimized *pswap()*, minimizing the number of paging structures moved by *pswap()*. Considering a PGD-aligned PM region is also PUD/PMD/PTE-aligned, MPFS user library ensures that files are placed in PUD/PMD/PTE-aligned addresses with aligned sizes. Thus, paging structures moved by *pswap()* are reduced to only one or a few monotypic structures, and no fragmentation occurs after *pswap()*.

D. Cross-process Protections for PGD-aligned Region.

User-space file systems face significant reliability issues due to scribbles (i.e., stray writes), a known problem in file systems [10], [17], [29] that overwrites data with arbitrary values. Scribbles become more serious in user-space file systems [5], [8], [9], [18], [24] because they directly map PM into the user address space, making the mapped data vulnerable to software bugs [3], [19], [20] that can induce scribbles. Existing MMIO-based works [5], [9], [18] use Intel's Memory Protection Key (MPK) to prevent unintended PM access. MPK uses a Protection Key and the PKRU register to partition the address space into 16 regions with distinct access rights. The access rights for each region are determined by 2-bit values in the PKRU register. For instance, setting the rights of key1 to 00 allows full access, while setting it to 10 makes the region read-only, triggering a SIGSEGV error on writes. MPK safeguards PM regions by setting them as read-only via modifying PKRU with the non-privileged instruction. In multi-process workloads, using MPK is risky due to its excessive flexibility. The PKRU register is not globally unique, requiring individual settings for each core, making it difficult to unify access rights across processes or threads, even when they map the same memory region. While Libmpk [21] offers inter-thread synchronization to maintain consistent PKRU values within a process, it cannot enforce this across all processes, as user processes can opt out of using any library. Consequently, user-space file systems may lose control over PM region access rights. A malicious process can modify PM data by setting the region's access to writable without being known by any other process, a vulnerability we have verified in practice.

User-space file systems require a cross-process memory protection mechanism to isolate malicious writes in multi-process scenarios. To address this, MPFS introduces a ***kernel-user cooperative memory protection*** mechanism. At the user level, MPFS uses MPK to provide intra-process memory protection, dividing the PM region into file and metadata areas, with both set to read-only during initialization. Writes require opening a *write window* by adjusting the PKRU. The *write window* will immediately close after each write to prevent scribbles. The kernel-level protection is based on the permission bits in paging structures, providing inter-process memory protection. The permission bits of the paging structures are set as read-only when the shareable page table is installed to a new process, to prevent unauthorized processes from writing to the PM region. The permission bits can be set as writable by invoking a system call named *grant_access()* with an authorized key to grant trusted processes access to MPFS. Owing to the PGD-aligned mapping, the *grant_access()* only needs to modify the PGD entries that attach the shareable page table in the current process, as shown in Figure 2. Since permission bits in upper-level paging structures override those in lower-level paging



(a) Performance in different IO patterns. (b) CDF of 4KB sequential write. (c) R/W in different fragmentation rates. (d) Multi-process real-world workloads.

(e) Multi-process shared write. (f) Multi-process concurrent Write. (g) Multi-process shared read. (h) Multi-process concurrent read.

Fig. 3. Performance overview of MPFS.

structures, setting a PGD entry's permission bit to writable makes the entire 512GB PM region it maps writable. Existing works cannot provide the same level of protection because they do not have a well-structured page table, thus they have to modify the permission bits in thousands of PTE/PMD entries one by one. If they directly disable the permission bits of a PGD entry in an irregular-shaped page table will result in DRAM mappings being mistakenly set as unwritable, causing unpredictable errors. MPFS uses *grant_access()* to grant the trusted processes access to PM data when the processes start, which merely takes about 8550 nanoseconds. After initialization, MPFS only needs MPK-based *write windows* to protect PM regions from intra-process scribbles, which only takes 23 cycles. Thus, the kernel-user cooperative memory protection guarantees the reliability of PM regions in multi-process scenarios, while achieving outstanding performance.

IV. Evaluation

We implement the evaluations on a system equipped with an Intel Xeon 6230R, 6 * 16GB DDR4, and 6 * 128GB Optane DC Persistent Memory (PM). To fully exploit the performance of PM, all PM devices are configured as App Direct Mode with interleaving. All evaluations run in Linux 5.7.1 or a modified kernel based on Linux 5.7.1.

A. Single-process Performance Overview

Exp.(a). We first evaluate MPFS and its competitors [7], [10], [16], [18], [25], [28] using FIO [2] as a microbenchmark for common I/O patterns. The evaluation consists of iterative read()/write() operations on a 128 MB file. The results in Figure 3(a) show that in sequential writes, MPFS performs on par with ctFS, both outperforming other competitors. Specifically, MPFS achieves 3.78x, 2.14x, and 2.01x higher bandwidth than NOVA, PMFS, and Ext4-DAX, respectively. User-space file systems bypass the complex kernel I/O stack, which consumes over 50% of the runtime of I/O operations, while NOVA's log-structured design adds significant overhead due to garbage collection. Compared to other MMIO-based systems like SplitFS and Libnvmmio, MPFS still shows performance gains of up to 1.73x and 2.91x, respectively, because these systems rely on Write-Ahead Logging (WAL) for data consistency, which results in additional writes for logging. In contrast, MPFS and ctFS can avoid these extra writes by directly moving the redo log to the file using *pswap()*. The read bandwidths of the four MMIO-based systems are nearly identical, as read operations do not involve consistency and persistency issues. However, due to the complex kernel I/O stack, kernel PM file systems exhibit poorer read performance compared to MMIO-based systems.

Exp.(b). The Cumulative Distribution Function (CDF) of 4KB sequential write shown in Figure 3(b) further reveals the differences in the I/O performance between various file systems. The I/O latency of MMIO-based systems is markedly lower than kernel file systems due to the simplified software stack. In addition, though the average I/O bandwidth of ctFS is almost the same as MPFS, the P999 latency of MPFS is only 1384ns, which is nearly half that of ctFS. This is because ctFS spends much more time on handling page faults while MPFS only needs to inherit the existing page table from the main task, without handling any page faults.

Exp.(c). MPFS exhibits superior I/O performance compared to ctFS in aged and fragmented scenarios. Although using huge pages boosts MMIO performance, file system operations that allocate and recycle memory can fragment huge pages into 512 normal pages [15], increasing page fault overhead and indexing complexity. MPFS mitigates this issue with the PGD-aligned shareable page table, reducing fragmentation and eliminating page fault overhead from the I/O critical path. As shown in Figure 3(c), MPFS outperforms ctFS by 20% in write performance at a 100% fragmentation rate. Even though CPU

caching reduces ctFS's performance loss in reads, it still lags behind MPFS by up to 6.5%.

B. Multi-process Evaluation

ctFS-mp. To evaluate the overhead of existing user-space file systems in multi-process scenarios, we developed an enhanced version of ctFS, named ctFS-mp, which supports multi-process operations. ctFS-mp ensures a consistent view of file mappings by synchronizing all page table modifications across processes using a multi-process lock mechanism called *sync lock*. This mechanism locks the page table of all processes using ctFS whenever any process modifies the mapping. Modifications such as *pswap()* and page faults are synchronized immediately, blocking read/write operations until the synchronization is complete to prevent inconsistencies. We evaluated MPFS, ctFS-mp, and ext4-DAX in multi-process scenarios involving shared/concurrent read and write operations. In **shared** read/write, multiple processes access the same file, whereas in **concurrent** read/write, each process accesses its own file. Each process used a single thread to avoid inter-thread overhead, and the file size was 208MB, triggering *pswap()* three times. The file system was reset before each test to ensure consistent conditions.

Exp.(e). In shared write, all processes wrote to the same file, with writes serialized due to a unique writer lock, causing runtime to increase linearly with the number of processes. As shown in Figure 3(e), MPFS outperformed ctFS and ext4-DAX by up to 1.8x and 6.5x, respectively. MPFS's performance advantages stem from two factors: (1) MPFS avoids additional page faults caused by other processes by sharing the page table, leading to an increasing performance gap between ctFS-mp and MPFS as the number of processes grows. (2) MPFS handles I/O operations in user space (except for the initial page fault per page), while ctFS-mp requires kernel involvement for file mapping synchronization, and ext4-DAX handles all I/O operations in kernel space, incurring overhead from kernel-user switches and the kernel software stack.

Exp.(f). In concurrent write, MPFS shows even greater performance advantages over ctFS-mp than in shared write scenarios. Each process writes to a separate file with three *pswap()* operations, and each *pswap()* must be synchronized with other processes. As the number of processes increases, the required *pswap()* operations and their associated overhead grow proportionally to the square of the process count. MPFS avoids this overhead due to its unique shareable page table, which requires no additional synchronization. As shown in Figure 3(f), MPFS outperforms ctFS-mp by up to 10.3x and ext4-DAX by 4.2x. This is because MPFS handles file locking and data transfer entirely in user space, avoiding the need to interact with the kernel except for *pswap()*.

In both shared (***Exp.(g)***) and concurrent read (***Exp.(h)***), MPFS and ext4-DAX demonstrate superior performance over ctFS-mp. The performance bottleneck in ctFS-mp stems from its process-specific page table design, where each process must handle page faults independently while holding a lock on the persistent page table. During page faults, ctFS-mp must verify page allocation status and either populate existing pages or allocate new ones, requiring exclusive locks throughout the read operation. In contrast, MPFS minimizes page faults through page table sharing across processes. While MPFS still requires brief locks during page faults, it enables immediate page sharing once populated, eliminating long-term read blocking. As evidenced in Figure 3(g), MPFS achieves up to 9.0x higher performance than ctFS-mp in shared read scenarios. Since all files in MPFS/ctFS-mp uses the same page table for indexing instead of separate indexes for each file, more files lead to more stalls. Consequently, in concurrent reads (Figure 3(h)), MPFS outperforms ctFS-mp by up to 15.6x by minimizing stalls caused by more files.

Exp.(d): Real-World Workloads. We evaluated MPFS with real-world applications by running a RocksDB [11] instance alongside common shell commands (e.g., *ls*, *cp*, *grep*) that share MPFS. We compared the performance of RocksDB running alone (RocksDB-alone) versus concurrently with shell commands (RocksDB-shared) using throughput as the metric. MPFS was tested with four workloads: *fillseq*, *fillrandom*, *overwrite*, and *readwhilewriting*, each involving 10 million key-value operations with 16B keys and 1024B values. The experimental result shows that the performance of MPFS-shared (running along with shell commands) is almost the same as that when running alone (MPFS-alone) in every workload and outperforms all competitors. This is due to the fact that concurrent read/write to a separate file will not cause any interference to other processes in MPFS. The slight performance degradation of MPFS-shared is caused by the concurrent access to PM, which results in a performance penalty of PM device [30]. However, the performance of ctFS-mp-shared is at most 1.93x lower than that when running alone in *fillrandom*, which contains more write operations due to the RocksDB compactions. As we discussed in the above evaluations, the more writes there are, the more *pswap()* need to be synchronized and the more time needs to be spent on synchronization. Thus, the performance of ctFS-mp-shared is sensitive to the number of writes. Even in the *readwhilewriting*, the performance of ctFS-mp-shared is still 1.59x lower than that when running alone.

V. Conclusion and Future Work

This paper presents MPFS, a novel user-space persistent memory file system optimized for multi-process environments. Through its PGD-aligned shareable page table design, MPFS minimizes overhead from page faults and cross-process synchronization while achieving memory efficiency. MPFS implements cross-process memory protection mechanisms to prevent unauthorized writes. Experimental results demonstrate MPFS's superior performance compared to existing PM file systems across both single and multi-process workloads. However, performance scalability remains a challenge under high concurrency due to hardware limitations of current PM devices [31] - an area targeted for future optimization. The ***source code*** of MPFS and ctFS-mp is available at *https://github.com/BOGEDABUDA/MPFS*.

References

[1] C. Alverti, V. Karakostas, N. Kunati, G. Goumas, and M. Swift, "Daxvm: Stressing the limits of memory as a file interface," in *Proc. MICRO*. IEEE, 2022, pp. 369–387.

[2] J. Axboe, "Flexible i/o tester," https://github.com/axboe/fio.

[3] 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.

[4] F. L. Camargos, "mysqld_multi: How to run multiple instances of mysql," https://www.percona.com/blog/mysqld_multi-how-to-run-multiple-instances-of-mysql/.

[5] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu, "Scalable persistent memory file system with Kernel-Userspace collaboration," in *Proc. FAST*, 2021, pp. 81–95.

[6] D. Chinner, "xfs: Dax support," https://lwn.net/Articles/635514/, 2015.

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

[8] B. Ding, W. Tong, Y. Hua, Z. Chen, X. Wei, and D. Feng, "Rmmio: Enabling reliable memory-mapped i/o for persistent memory systems," in *Proc. ICCD*. IEEE, 2022, pp. 722–725.

[9] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and protection in the zofs user-space nvm file system," in *Proc. SOSP*, 2019, pp. 478–493.

[10] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proc. EuroSys*, 2014, pp. 1–15.

[11] Facebook, "Rocksdb: A persistent key-value store for fast storage environments." https://rocksdb.org/, 2012.

[12] T. A. S. Foundation, "The apache hadoop project develops open-source software for reliable, scalable, distributed computing." https://hadoop.apache.org/, 2012.

[13] J. Handy and T. Coughlin, "Optane's dead: Now what?" *Computer*, vol. 56, no. 3, pp. 125–130, 2023.

[14] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[15] R. Kadekodi, S. Kadekodi, S. Ponnapalli, H. Shirwadkar, G. R. Ganger, A. Kolli, and V. Chidambaram, "Winefs: a hugepage-aware file system for persistent memory that ages gracefully," in *Proc. SOSP*, 2021, pp. 804–818.

[16] 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. SOSP*, 2019, pp. 494–508.

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

[18] R. Li, X. Ren, X. Zhao, S. He, M. Stumm, and D. Yuan, "ctfs: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory," *ACM Transactions on Storage*, vol. 18, no. 4, pp. 1–24, 2022.

[19] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan, "Cross-failure bug detection in persistent memory programs," in *Proc. ASPLOS*, 2020, pp. 1187–1202.

[20] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "Pmtest: A fast and flexible testing framework for persistent memory programs," in *Proc. ASPLOS*, 2019, pp. 411–425.

[21] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel mpk)." in *Proc. USENIX ATC*, 2019, pp. 241–254.

[22] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung *et al.*, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

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

[24] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proc. EuroSys*, 2014, pp. 1–14.

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

[26] X. Wu and A. L. N. Reddy, "Scmfs: A file system for storage class memory," in *Proc. SC*, 2011, pp. 1–11.

[27] L. Xiang, X. Zhao, J. Rao, S. Jiang, and H. Jiang, "Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering," in *Proc. EuroSys*, 2022, pp. 488–505.

[28] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. FAST*, 2016, pp. 323–338.

[29] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proc. SOSP*, 2017, pp. 478–496.

[30] 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. FAST*, 2020, pp. 169–182.

[31] D. Zhou, Y. Qian, V. Gupta, Z. Yang, C. Min, and S. Kashyap, "ODINFS: Scaling PM performance with opportunistic delegation," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 179–193. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/zhou-diyu