HUNTER: Releasing Persistent Memory Write Performance with A Novel PM-DRAM Collaboration Architecture

Yanqi Pan†, Yifeng Zhang†, Wen Xia†‡, Xiangyu Zou†, and Cai Deng†
†School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China
‡Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247940

***Abstract*—We present HUNTER, a POSIX-compliant persistent memory (PM) file system that fully releases PM's write performance. Compared to state-of-the-art ones, HUNTER proposes a novel PM-DRAM collaboration architecture to significantly eliminate/reduce software overheads in the write path. Expensive in-PM metadata are updated asynchronously to hide their performance penalties. Furthermore, in-PM metadata/data are laid out separately for locality awareness, enabling collaboration with asynchronous architecture. HUNTER also adopts several light-weight in-DRAM allocators/indexes to manage PM efficiently.**

Experimental results suggest that HUNTER achieves 2.0–3.4× write bandwidth compared to state-of-the-art PM file systems in write-intensive workloads and shows similar write bandwidth compared to bare PM.

I. Introduction

Persistent Memory (PM), such as PCM [10], ReRAM [8], and 3D-XPoint [16], shows great performance potential by combining the best of memory and storage [4]. With its DRAM-comparable I/O performance and byte-addressability, PM can be attached to the memory bus, sitting alongside the DRAM [20], [22], and accessed directly via CPU loads and stores. Furthermore, given PM's large capacity [22], using PM as storage is also a promising use case.

The advent of PM upends the traditional storage stack due to its much lower latency compared to Hard Disk Drive (HDD) and Solid State Drive (SSD). Therefore, reducing software overheads is critical in building a high-performance PM file system. Many researchers have explored the PM's performance potential by introducing techniques such as *Direct Access* (DAX) [5], [15], [18], [21], hardware-acceleration [4], [12], [23], and user-space accessing [4], [9], [17] to achieve this end. Among them, NOVA and SplitFS are two state-of-art PM file systems. NOVA [21] is an in-kernel file system that builds a hybrid architecture to efficiently manage in-PM metadata logs and written data blocks. SplitFS [9] proposes a split architecture that builds a POSIX-like user-space file system through the memory mapping interface (i.e., `mmap()` syscall) provided by the underlying in-kernel file system.

Nevertheless, existing PM file systems fail to fully exploit PM's performance, resulting in a huge waste of write bandwidth. One key reason for this is that they do not expect how severe performance degradation in-PM metadata updates can incur (observed in §II-B); thus, they force each PM I/O to be persisted immediately, even under standard POSIX interfaces. We argue that such strict data integrity assumptions contradict both POSIX semantics [14] and the general design purpose of mainstreaming file systems [2], [11]. Moreover, inefficient software designs, such as heavy metadata garbage collection (GC) during log extending in NOVA [21], also lead to such waste. Figure 1 reflects the seriousness of the above problems by measuring the sequential write bandwidth of various PM file systems when populating the different size files. SplitFS is omitted since it relies on the underlying in-kernel file system [9]. We observe that more than 50% performance penalties exist in NOVA and PMFS compared to the bare PM, which underestimates the overall value of PM.

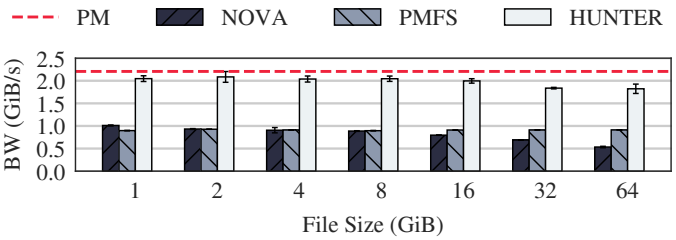


Fig. 1: *Write bandwidth of state-of-the-art PM file systems. The dashed line in the figure shows the write bandwidth of a bare PM on our machine, which is measured by OptaneStudy [22].*

We present HUNTER[1], an in-kernel PM file system that seeks to eliminate such performance penalties and strictly follows POSIX semantics. The novelty of HUNTER lies in that it proposes a novel PM-DRAM collaboration architecture that updates metadata asynchronously and maintains the locality of in-PM metadata to collaborate with the asynchronous mechanism. Compared to prior PM file systems, HUNTER exploits PM's write performance with three specific techniques: ❶ the asynchronous mechanism that routes heavy in-PM metadata updates to the background, ❷ in-PM data-metadata split layout with locality-aware metadata area and GC-free data logs, and ❸ several light-weight in-DRAM data structures to mitigate software overheads. These optimization techniques work together to achieve better collaboration between PM and DRAM. Notably, our explorations on designing HUNTER are driven by considering the following critical questions.

The first question is how to build in-DRAM structures that efficiently collaborate with PM I/O to fully release PM's write performance. According to our observation in §II-B, in-PM metadata updates cause severe performance penalties in the write path due to its small/random pattern. Fortunately, asynchronous I/O supported by POSIX semantics helps hide such penalties. POSIX semantics allows file systems to delay PM writes while using `fsync()` to assure that data are persisted immediately, which is leveraged by many mainstreaming file systems [2], [11], as illustrated in Figure 2a. Many real-world applications without strict data integrity requirements, such as LASR [13], can benefit from the efficient asynchronous I/O through POSIX interfaces. HUNTER also follows POSIX semantics to maximize its write performance but makes some PM-aware improvements. For example, HUNTER only

[1]We call it HUNTER because it reduces any software overheads that might cause I/O performance degradation, like a hunter hunts down its prey.

performs expensive in-PM metadata updates asynchronously, causing much fewer background interferences and alleviating severe double-copy overheads. Besides, HUNTER adopts several optimized in-DRAM structures to mitigate software overheads, such as using dynamic arrays [1] with the *workload-aware pre-allocation* technique for fast indexing.

The second question is how to design an in-PM layout that collaborates with our designed in-DRAM structures, or in other words, how to minimize the asynchronous interferences by well-organized in-PM data/metadata. HUNTER tends to extend the log-structured file system (LFS) to achieve the goal since LFS holds strong crash consistency guarantees and efficient data management approaches. However, extending LFS is not ideally straightforward due to the following two problems: ❶ **Poor metadata locality**. Traditional LFS appends metadata/data in the same log. Thus, asynchronous metadata updates break I/O sequentiality, resulting in non-negligible performance degradation. ❷ **Heavy log cleaning overheads**. Log cleaning adds significant overheads since it causes numerous in-media data/metadata copying and updating [11], [21]. The cleaning procedure is sensitive in PM file systems since PM has much lower I/O latency. HUNTER introduces an elegant, locality-aware, and GC-free data-metadata split PM layout to address the above problems. This layout manages in-PM data and metadata separately to collaborate with the asynchronous mechanism. Moreover, cleaning overheads are eliminated by efficiently reusing invalid space at run time.

We implement HUNTER in Linux kernel 5.1.0, supporting mostly common POSIX-compliant file operations. We further evaluate HUNTER with several file system benchmarks, such as FIO [7] and Filebench [6]. The evaluation results suggest that HUNTER achieves up to 3.4× write bandwidth compared to NOVA using FIO and shows the lowest data write latency in various Filebench workloads. We also observe that the write bandwidth of HUNTER can match that of a bare PM, proving that HUNTER can fully exploit PM's write performance.

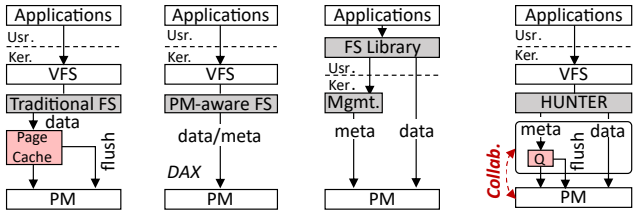
To summary, this paper makes the following contributions:

- A detailed performance breakdown of existing PM file systems in their write paths and a revelation of the huge performance penalties for synchronous metadata updates.
- A novel PM-DRAM collaboration architecture that updates metadata asynchronously collaborated with a locality aware PM layout to release PM's write performance.
- The design and implementation of an POSIX-compliant in-kernel PM file system, HUNTER, with a PM-DRAM collaboration architecture, GC-free schemes and lightweight data management structures.
- Experimental evaluations that suggest HUNTER outperforms state-of-the-art (in-kernel) PM file systems, especially in write-intensive workloads.

II. Background and Motivation

A. Persistent Memory and PM File Systems Architectures

Emerging PM technologies bring a groundbreaking breakthrough to the existing storage stack. A PM usually has 128–512GiB capacity and achieves similar write latency while 2–3× read latency compared to DRAM [20], [22]. Traditional



(a) Traditional FS (e.g., EXT4) (b) PM-aware FS (e.g., NOVA) (c) User-space FS (e.g., SplitFS) (d) HUNTER (Ours) (Release PM Perf.)

Fig. 2: *Architecture comparison of different PM file systems. HUNTER and traditional file systems are designed with asynchronous mechanisms following POSIX semantics. However, HUNTER does not buffer data in DRAM since data flushes result in significant double-copy overheads [5], [21].*

file systems designed for disks and flash memories with page cache based (asynchronous) architecture [2], [11], as shown in Figure 2a, no longer suit efficient PM due to their heavy software overheads, such as double-copy issues [5], [21]. Note that their metadata are also cached in DRAM, such as inode [2], [11], and flushed when `fsync()` is evoked. But they are not shown in the figure due to space limits.

Efficient file system I/O is critical to high-level storage applications, such as databases and key-value stores. Therefore, many new PM file system architectures have been proposed to address the problem [4], [5], [9], [12], [13], [15], [17], [18], [21], [23], as illustrated in Figures 2b and 2c. Among them, only PMFS, NOVA, SplitFS, and ctFS are publicly available and well-tested. These file systems utilize different techniques to improve I/O performance. PMFS leverages DAX to prevent double-copy overheads, but its data operations are not atomic. NOVA builds a hybrid log-structured file system to provide strong crash consistency but still suffers heavy GC overheads. SplitFS proposes a new split architecture that implements file system operations on top of `mmap()` interface to largely enhance write performance but it is limited by the underlying in-kernel file system. ctFS is a user-space file system that leverages memory management unit (MMU) to accelerate indexing, which is similar to SIMFS. However, such a strategy requires file systems to organize the in-PM data as page table-like structures and causes frequent in-PM accesses, thus harming the limited lifecycle of PM.

Moreover, one common problem for these PM file systems is that they try to design and implement under standard POSIX interfaces but use some stronger semantics than POSIX. For example, NOVA and PMFS always persist PM writes immediately, making `fsync()` meaningless. This observation motivates us to propose a novel PM-DRAM collaboration architecture in HUNTER, as illustrated in Figure 2d, which follows POSIX semantics and guarantees strong crash consistency. Note that HUNTER only delays metadata updates to the background, which alleviates severe double-copy overheads compared to the traditional ones (in Figure 2a).

B. Observed Performance Bottlenecks in PM File Systems

Generally, a `write()` syscall of PM file systems consists of several bookkeeping steps: reserving space on PM (*allocate*), writing data from DRAM buffer to PM (*write-data*),

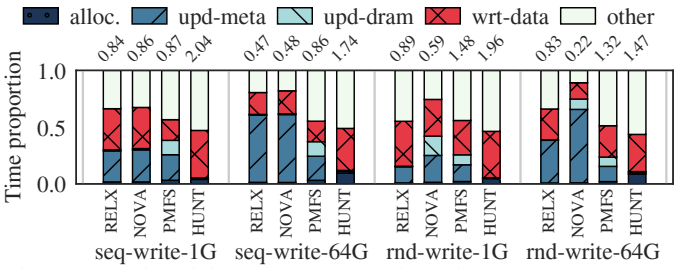


Fig. 3: *Time breakdown of sequential/random writes of PMFS, NOVA-RELAX (RELX), NOVA, and HUNTER (HUNT). The number above each bar is the bandwidth in GiB/s. NOVA's GC overheads are involved in* update-metadata.

updating corresponding in-PM metadata to reflect changes (*update-metadata*), and updating in-DRAM data structures to make these changes visible (*update-dram*). The key premise of a high-performance PM file system is to reduce overheads of *allocate*, *update-metadata*, and *update-dram* steps. We then make the following observations to understand the write performance bottlenecks of existing PM file systems.

Figure 3 shows the time breakdown of sequential and random writes of PMFS, NOVA, and NOVA-RELAX. The only difference between NOVA-RELAX and NOVA is that NOVA-RELAX does not use the copy-on-write mechanism for data atomicity. All the file systems suffer from heavy in-PM metadata update overheads because metadata are randomly distributed in these PM file systems. Note that NOVA performs worse when the size grows because heavy synchronous GC overheads degrade the overall performance (the GC overheads are involved in *update-metadata*). In random write workloads, *update-dram* overheads increase largely since NOVA needs to perform overlay checks on the index and harms performance.

Existing PM file systems have significant software overheads in their write path, which underestimates the potential of PM. HUNTER is motivated to address these bottlenecks by leveraging our proposed PM-DRAM collaboration architecture, GC-free data logs, and many optimized light-weight data structures. Thus, HUNTER can eliminate or reduce these software overheads in the write path.

III. HUNTER: Design and Implementation

A. Design Goals

Maximal write I/O performance. HUNTER aims to fully release PM's write performance by eliminating and reducing software overheads in the write path as much as possible.

Asynchronous metadata updates. HUNTER aims to eliminate metadata update overheads in the write path through an asynchronous architecture. This hides many slow small and random PM I/Os and improves overall write performance.

POSIX-compliant semantics. HUNTER enables efficient (metadata) asynchronous I/O to high-level applications with strong crash guarantees. While users can also assure data integrity by timely calling `fsync()` provided by HUNTER.

Light-weight logging and data structures. HUNTER aims to build GC-free data logs with strong crash consistency. It also maintains light-weight management structures to reduce further software overheads (e.g., dynamic arrays as indexes).

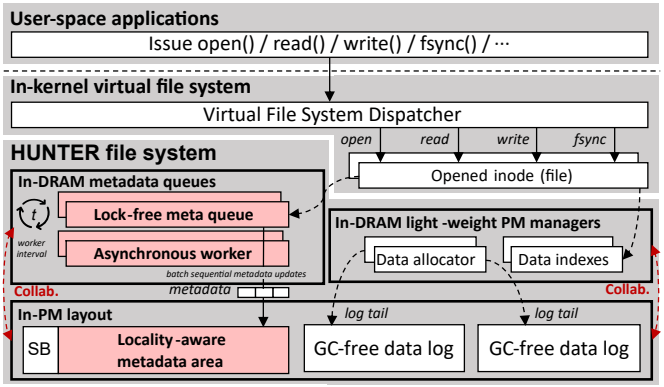


Fig. 4: *HUNTER Overview. Each file is associated with a lock-free queue for asynchronous metadata updates. For open(), HUNTER creates a dynamic array as index for the file. read() locates the offset in the index and copies in-PM data directly to the user buffer without buffering. write() is a little complex, see details in §III-C. At last, fsync() synchronously flushes the file's metadata in the queue to PM to guarantee data integrity*.

TABLE I: Techniques used in HUNTER. All of these work together to largely reduce software overheads in the write path.

Techniques	*Contributions*
PM-DRAM collaboration architecture	Low-overhead data management, efficient asynchronous metadata updates
Data-Metadata split layout	Optimized metadata updates, crash tolerate, well collaboration with async architecture
Imm-eviction and *gap-write*	GC-free log space allocation
Light-weight data structures	Low-overhead allocation and indexing

In-kernel implementation. HUNTER aims to leverage the in-kernel virtual file system to provide efficient and POSIX-compatible PM access interfaces for high-level applications.

B. Overview

Figure 4 presents the overall architecture of HUNTER and Table I summarizes used techniques and their contributions.

PM-DRAM collaboration architecture. HUNTER stores data and metadata in PM and manages them in DRAM with light-weight data structures. To eliminate significant in-PM metadata update overheads, HUNTER routes metadata updates to the file-associated (dequeue) lock-free queues in DRAM and deploys four workers to flush them periodically to PM. Flushing might interfere with foreground data operations, so each worker flushes at most 2^{18} metadata updates each time. Therefore, If each metadata update represents 4KiB data, then the flushing process persists at most 1GiB data per run.

Data-Metadata split layout. In HUNTER, data logs and metadata are stored in separate areas to improve metadata locality. This is the key design to collaborate with the above asynchronous architecture since we can flush small metadata updates in batch and thus have thg potential to achieve sequential asynchronous writes on PM. Further, HUNTER manages multiple (per-CPU) data logs to provide high concurrency, while traditional LFSs [19] apply only a single log since the underlying media below has poorer concurrency (e.g., some

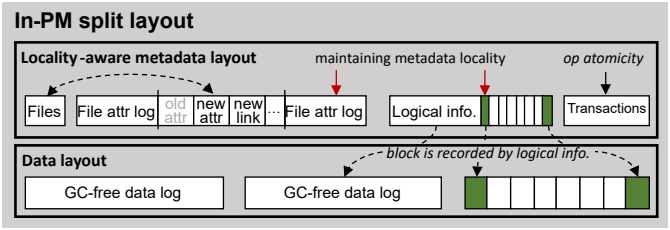


Fig. 5: *In-PM layout. HUNTER splits data and metadata into two areas. The locality of metadata is maintained to collaborate with asynchronous architecture. In the file attr log, new attr entry evicts the old attr entry by using imm-eviction.*

NOR/NAND flashes) compared to PM devices.

GC-free space management with *imm-eviction* and *gap-write* techniques. HUNTER avoids GC by reusing invalid PM space at run time. For metadata, we reserve each file a fixed-size metadata log in PM (i.e., *file attr log*). Attributes updates of the file are recorded in this log. Since the attribute type is fixed, we can immediately evict the stale attribute entry when a new same-type attribute is written. This is referred to as *imm-eviction*. Although *imm-eviction* might cause fragmentation, the log is small and updates can hit the buffer with the locality.

For data, HUNTER appends them to the data log and records their logical information in a reserved space, together with the metadata logs. For brevity, we refer to data's logical information and file attributes as metadata. Once the data log is full, HUNTER performs *gap-write* that scans corresponding logical information to find invalid data blocks quickly and allocates these space for new data writes. The scanning is efficient since the logical information occupies at most 1.5% of PM size (i.e., given logical information 64B for each 4KiB block, $\frac{64}{4096} = 1.5\%$ PM size continuous space is reserved). Figure 5 illustrates the in-PM layout of HUNTER.

Light-weight data structures. Allocation and indexing are frequent operations during `write()`. For allocation, HUNTER moves the log tail forward in most cases, which consumes only one add operation. Note that the tail is maintained in DRAM; thus add operation has low overhead. To accelerate *gap-write*, we cache an interval double list in DRAM to efficiently index reusable space. Interval double list is also used for file number allocation since we do not care about allocation orders. For indexing, HUNTER creates a dynamic array for each opened file. Thus, searching for an offset takes only one array access. To prevent frequent extending, HUNTER pre-allocates 2MiB space for each file, which supports index $\frac{2\text{MiB}}{8\text{B}} * 4\text{KiB} = 1\text{GiB}$ file at initialization.

Crash consistency and recovery. HUNTER populates an undo transaction area in PM for complex file operations, such as `rename()`. Each transaction contains a sequence of pointers to the involved metadata. Transaction guarantees metadata consistency, providing trustworthy information for crash recovery: HUNTER first undoes unfinished transactions and then scans logical information and validates them against the corresponding metadata based on timestamp. If data are newer or belonging to a deleted file, then the data are dropped for consistency. Note that the timestamps are stored in both logical information and attributes during each file operation.

C. Let's Handle Write I/O

We present the details of the write operation in this subsection, and see how HUNTER accelerates it.

Assuming we are writing a character "a" to an empty file "/test". First, HUNTER allocates an empty data block at the tail of a data log (say p) corresponding to the current CPU and then moves that tail forward (i.e., $tail = p+blk\_size$). Second, HUNTER copies data "a" from the user buffer to the allocated data block. Third, HUNTER packs metadata update request and sends it to the metadata queue based on file number. The request contains two fields, including the logical information of the allocated block and the changed attributes of "/test" after writing (e.g., size/mtime/ctime). Finally, HUNTER efficiently assigns p to index[0] to make the data visible to the user.

When the worker wakes up, it grabs and extracts a batch of requests from the queue and then flushes them to PM in order. Note that logical information is always flushed before file attributes, which guarantees the data is only trustworthy after the metadata is persisted, which helps crash consistency.

D. Implementation

We implement HUNTER on top of Linux kernel 5.1.0, which NOVA supports. This subsection presents some specific details of HUNTER's implementation.

PM-DRAM collaboration architecture. HUNTER carefully separates PM and DRAM operations to avoid unnecessary PM access. Specifically, we maintain frequently changing structural information such as log tails, queues, and indexes in DRAM and rebuild them during mount time. While the metadata and data are stored in PM for persistence.

Data-Metadata split layout is illustrated in Figure 5. For metadata, HUNTER reserves a 2^{20}-entries table in PM to store the file's inode (256B). For each inode, we also reserve a 256B space for *file attr log*. For simplicity, HUNTER supports two attribute entries: *attr-entry* for size/time/mode/owner changes and *link-entry* for link changes. Each entry is aligned to 64B, so 256B is sufficient for the log with *imm-eviction* technique. We also reserve each 4KiB block a 64B space for its logical information, which contains block's file number, logical offset, and timestamp. At last, per-CPU transactional areas are reserved. Each transaction area holds 4KiB space. In summary, given a 32 cores machine with 256GiB PM, the overall metadata space occupies 1.76% of this PM. Note that for data logs, HUNTER also manages them in a per-CPU manner, i.e., there are 32 data logs for 32 cores.

File-associated metadata queues. To avoid insertion contention of dequeue lock-free queues becoming a bottleneck, HUNTER distributes requests into a 64-slots hash table that manages 64 queues. Each file is associated with a queue determined by the equation: $queue\ slot = file\ number \pmod{64}$. In the current implementation, we use infinite queues to trade memory usage for non-blocking foreground requests (e.g., 130MiB memory usage for 64GiB write measured by *free* command). However, this can be optimized in the future.

Improvements on file open. In HUNTER, creating a file

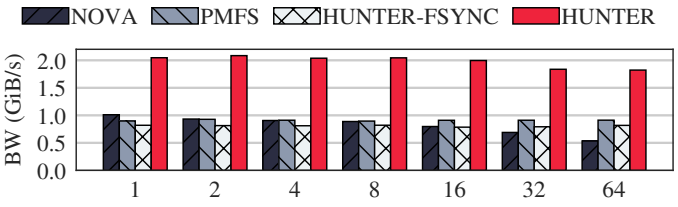


Fig. 6: *Comparison of sequential write bandwidth. HUNTER-FSYNC denotes call fsync() after each write() syscall.*

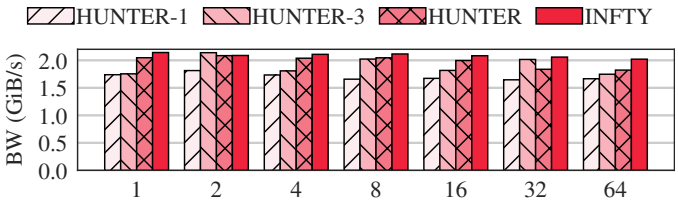


Fig. 7: *Worker interval sensitive test. The default worker interval (HUNTER) is 5 seconds. HUNTER-1 or -3 wakes up workers every 1 or 3 seconds. INFTY never wakes up workers.*

needs to pre-allocate a continuous 2MiB space for indexing, which could introduce high overheads if memory is fragmented. In practice, we observe that the file size is often smaller (e.g., more than 40% files in Linux kernel less than 4KiB). Therefore, HUNTER samples the history file size and speculatively allocates the memory for the index by calculating the average historical size. We call the technique as *workload-aware pre-allocation*. Another improvement is maintaining an in-PM link for each created file with its data blocks. Thus, HUNTER can follow that link to quickly locate the data blocks in a file rather than scan the logical information area.

E. Flexible Configurations

HUNTER is configurable to satisfy flexible requirements. We present some common configurations in this subsection.

Workers interval. The default wake-up interval for each worker is 5s, which is similar to that of EXT4 [2]. For those applications without strict data integrity requirements (e.g., LASR [13]), HUNTER can set a larger interval like the 60s.

Number of files. There are 2^{20} files by default. Users can enlarge the number to support more files at the cost of more metadata space consumption. We can implement a dynamic allocation scheme to prevent space waste in our future work.

Per-file index size and history window. The default size of the per-file index is 2MiB. For some dedicated applications, such as event logs, users can set larger index size to alleviate extending latency. Another configuration is the history window, which is used to restrict the sample range of *workload-aware pre-allocation*. The default window is 1, which means the subsequent index allocation size is exactly the last file size.

IV. Evaluation

In this section, we seek to answer the following questions:

- Can HUNTER fully release PM's write performance compared to existing PM file systems? (§IV-B)
- What workloads are most favored by HUNTER? (§IV-C)
- How do optimizations affect HUNTER? (§IV-D)
- How expensive is HUNTER recovery? (§IV-E)

A. Experimental Setup

The experiments are conducted on a server running on CentOS Stream (Linux kernel 5.1.0) with an Intel Xeon Gold 5218 CPU 2.3GHz with 16 cores, 128GiB DDR4 DRAM, and 512GiB Intel Optane DC Persistent Memory Module (PMM). In the evaluation, we configure PMM to non-interleaved mode.

B. Microbenchmarks

In this section, we use a widely used file system benchmark FIO [7] to evaluate the write I/O performance of HUNTER.

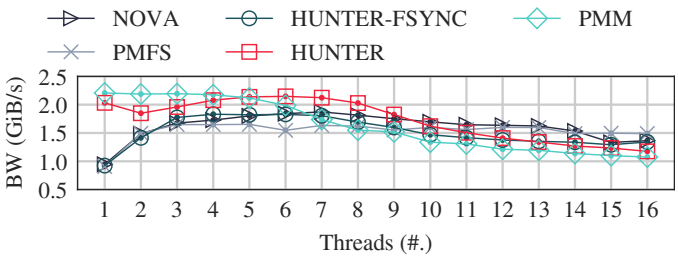


Fig. 8: *Comparison of concurrent write bandwidth.*

We omit the read I/O performance since the read path is so simple that all involved PM file systems show similar results. The I/O size is aligned to a block size (4KiB).

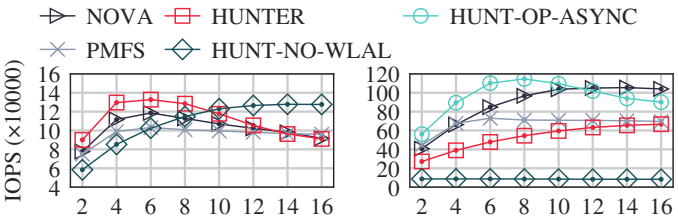
Single thread. Figure 6 shows the write bandwidth when populating different size files with a single thread. HUNTER is 1.0–2.4×, 1.0–1.3× faster than NOVA and PMFS. Note that HUNTER's write bandwidth degrades approximately by 11% with the increasing file size. This is because the number of asynchronous metadata updates increase with the file size, which impacts the foreground write I/O performance.

Sensitive study. Figure 7 studies the write performance of HUNTER under different interval configurations (the benchmark is used as same as a single thread). In the figure, INFTY shows the highest write performance compared to others since it issues no metadata updates during benchmarking.

Concurrency. Figure 8 shows the concurrent write performance of involved PM file systems with multiple threads. These threads populate a total of 4GiB files under the same directory. In the figure, HUNTER always catches up with PM's write bandwidth and shows the highest write performance with 1–9 threads compared to other file systems. The performance degradation of HUNTER is caused by the inner contention of PMM [20], which is also observed in other file systems.

C. Macrobenchmarks

Filebench [6] is usually used to simulate the behaviors of real-world workloads. Figure 9 shows IOPS throughput under two representative workloads, Fileserver and Varmail, with configurations similar to prior works [3], [21]. Each workload is tested under 1–16 threads. In the Fileserver workload, HUNTER is 10–23% and 18–30% faster than NOVA and PMFS when thread number $\leq$ 10. Note that as thread number increases, NOVA, PMFS, and HUNTER all suffer performance degradation due to the inner contention of PMM. However, HUNT-NO-WLAL (without *workload-aware pre-allocation*) shows good scalability even under 16 threads since its heavy



(a) Fileserver (1-16 threads) (b) Varmail (1-16 threads)

Fig. 9: *Comparison using Filebench. HUNT-NO-WLAL does not use the* workload-aware pre-allocation *technique; While HUNT-OP-ASYNC makes all file operations asynchronous.*

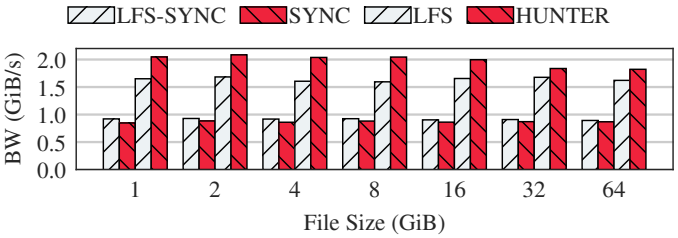


Fig. 10: *Performance comparison between HUNTER and LFS. LFS is a HUNTER's variant that appends its metadata in the data log. SYNC indicates that all file operations become synchronous. LFS-SYNC combines LFS with SYNC.*

software overheads reduce PM accesses frequency and thus alleviate the hardware contention. In the Varmail workload, HUNTER is slower than PMFS and NOVA because there are numerous file operations such as open and create. HUNTER does not specifically optimize these operations. However, we can easily incorporate them into our asynchronous architecture. The evaluation results show that HUNT-OP-ASYNC, with all metadata operations being asynchronous, outperforms PMFS and NOVA by 41% and 13% respectively on average.

D. Evaluation of Techniques Used in HUNTER

Workload-aware pre-allocation. Figure 9 shows the performance of HUNTER with/without *workload-aware pre-allocation* technique. With the technique, HUNTER's IOPS is increased by 3–75% on Fileserver workload when thread number < 10 and by 2.0–7.0× on Varmail workload.

Asynchronous metadata updates. Figure 10 shows the performance of HUNTER with/without the asynchronous approach when populating different size files. We can see that the asynchronous metadata updates mechanism significantly improves the write bandwidth by 1.1–1.4×.

Data-Metadata split layout. Figure 10 compares our split layout and the traditional LFS layout. We can see that the split layout can better collaborate with asynchronous architecture since it maintains metadata locality, and background workers can leverage that to perform batched and sequential PM writes.

E. Recovery Overheads

We evaluate the recovery overheads of NOVA, PMFS, and HUNTER under a 64GiB write-only workload. During normal recovery, the recovery speed of HUNTER (565ns) is 27.6× and 6.09× compared to that of NOVA (15697ns) and PMFS (3439ns) because HUNTER can restore its simple and small structures efficiently. However, during failure recovery, though HUNTER (2.81s) is still 7.6× faster than NOVA (24.1s), it is 2.64s slower than PMFS because HUNTER must scan whole logical information for data validation, which results in unnecessary PM accesses. In contrast, PMFS only scans those valid metadata blocks.

V. Conclusion

This paper presents HUNTER, an in-kernel and POSIX-compliant PM file system with a novel PM-DRAM collaboration architecture. Experimental results show that HUNTER significantly eliminates/reduces heavy software overheads in the write path and can fully exploit PM's write performance.

Acknowledgment

This research was partly supported by the National Natural Science Foundation of China under Grant no. 61972441; Shenzhen Science and Technology Innovation Program under Grant no. RCYX20210609104510007 and no. JCYJ20200109113427092; Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies (2022B1212010005). Wen Xia (xiawen@hit.edu.cn) is the corresponding author.

References

[1] A. Brodnik, C. Svante, et al. Resizable arrays in optimal time and space. In *Proc. WADS*, 1999.
[2] M. Cao, B. Suparna, et al. Ext4: The next generation of ext2/3 filesystem. In *Proc. LSF*, 2007.
[3] Y. Chen et al. Scalable persistent memory file system with kernel-userspace collaboration. In *Proc. FAST*, 2021.
[4] M. Dong et al. Performance and protection in the zofs user-space nvm file system. In *Proc. SOSP*, 2019.
[5] S. R. Dulloor, K. Sanjay, et al. System software for persistent memory. In *Proc. EuroSys*, 2014.
[6] G. Amvrosiadis et al. Filebench file system benchmark. https://github.com/filebench/, 2016.
[7] J. Axboe et al. Flexible i/o tester. https://github.com/axboe/fio, 2017.
[8] R. Fackenthal et al. 19.7 a 16gb reram with 200mb/s write and 1gb/s read in 27nm technology. In *Proc. ISSCC*, 2014.
[9] R. Kadekodi et al. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proc. SOSP*, 2019.
[10] B. C. Lee et al. Architecting phase change memory as a scalable dram alternative. In *Proc. ISCA*, 2009.
[11] C. Lee, D. Sim, J. Hwang, et al. F2FS: A new file system for flash storage. In *Proc. FAST*, 2015.
[12] R. Li, R. Xiang, Z. Xu, et al. ctfs: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In *Proc. FAST*, 2022.
[13] J. Ou et al. Hinfs: A persistent memory file system with both buffering and direct-access. In *Proc. EuroSys*, 2016.
[14] Y. Qian et al. Metawbc: Posix-compliant metadata write-back caching for distributed file systems. In *Proc. SC*, 2022.
[15] E. H.-M. Sha et al. A new design of in-memory file system based on file virtual address framework. *IEEE Transactions on Computers*, 2016.
[16] R. Smith. Intel announces optane storage brand for 3d xpoint products. https://www.anandtech.com/show/9541/, 2015.
[17] H. Volos et al. Aerie: Flexible file-system interfaces to storage-class memory. In *Proc. EuroSys*, 2014.
[18] C. Wang et al. Lawn: Boosting the performance of nvmm file system through reducing write amplification. In *Proc. DAC*, 2018.
[19] David Woodhouse. Jffs: The journalling flash file system. In *Ottawa linux symposium*, 2001.
[20] L. Xiang et al. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proc. EuroSys*, 2022.
[21] J. Xu et al. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *Proc. FAST*, 2016.
[22] J. Yang et al. An empirical guide to the behavior and use of scalable persistent memory. In *Proc. FAST*, 2020.
[23] J. Yi, M. Dong, et al. HTMFS: Strong consistency comes for free with hardware transactional memory in persistent memory file systems. In *Proc. FAST*, 2022.