P²Cache: Exploring Tiered Memory for In-Kernel File Systems Caching

Zhen Lin, *Binghamton University;* Lingfeng Xiang and Jia Rao,
The University of Texas at Arlington; Hui Lu, *Binghamton University*

https://www.usenix.org/conference/atc23/presentation/lin

This paper is included in the Proceedings of the
2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the
2023 USENIX Annual Technical Conference
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

P²CACHE: Exploring Tiered Memory for In-Kernel File Systems Caching

Zhen Lin*, Lingfeng Xiang†, Jia Rao†, Hui Lu*
*Binghamton University, †The University of Texas at Arlington

Abstract

Fast, byte-addressable persistent memory (PM) is becoming a reality in products. However, porting legacy kernel file systems to fully support PM requires substantial effort and encounters the challenge of bridging the gap between block-based access granularity and byte-addressability. Moreover, new PM-specific file systems remain far from production-ready, preventing them from being widely used. In this paper, we propose **P²CACHE**, a novel in-kernel caching mechanism to explore how legacy kernel file systems can effectively evolve in the face of fast, byte-addressable PM. P²CACHE exploits a read/write-distinguishable memory hierarchy upon a tiered memory system involving both PM and DRAM. P²CACHE leverages PM to serve all write requests for instant data durability and strong crash consistency while using DRAM to serve most read I/Os for high I/O performance. Further, P²CACHE employs a simple yet effective synchronization model between PM and DRAM by leveraging device-level parallelism. Our evaluation shows that P²CACHE can significantly increase the performance of legacy kernel file systems – e.g., by 200x for RocksDB on Ext4 – meanwhile equipping them with instant data durability and strong crash consistency, similar to PM-specialized file systems.

1 Introduction

Rapid changes in storage technologies, ranging from rotating hard disk drives (HDD) to NAND-based solid-state drives (SSD), Non-Volatile Memory Express (NVMe) [9], and Storage Class Memory (SCM) [10], play an essential role in driving the evolution of kernel file systems, such as Ext4 [35], Btrfs [37], and XFS [24]. Features have been continuously added to accommodate new and unique characteristics of storage devices. Examples include I/O schedulers designed for different types of storage media [21, 31, 39], concurrency and scalability support for high-speed storage on multi-core systems [17, 20], and the introduction of direct access mode (DAX) for byte-addressable SCM [5, 22].

However, the evolution of kernel file systems hits a plateau in light of emerging *fast, byte-addressable* storage [9, 10]. Kernel file systems are inherently built with the assumption of *slow, block-addressable* storage devices (e.g., HDD/SSD) sitting below. The layered in-kernel storage stack transforms I/O requests from applications to block operations for storage devices. As storage devices become faster, the overhead of the layered storage stack becomes more significant. For example, software contributes 50% of the read latency on a low latency (3 us) NVMe SSD [45]. The software overhead becomes even more dominant as storage devices become faster and closer to the CPU. Intel's Optane Persistent Memory [10] sitting on the memory bus incurs read latency as low as ∼170 ns [44].

To address this pressing challenge, some approaches tended to discard traditional kernel file systems. Indeed, a batch of new file systems [5, 18, 19, 23, 27, 28, 40, 42, 43] has been proposed – i.e., tailored for persistent memory (PM) [10] – and achieved high I/O bandwidth, low I/O latency, and strong crash consistency. Other approaches bypassed the kernel storage stack by exposing PM or low-latency SSDs directly to applications with userspace libraries [2] or file systems [32]. However, such new file systems and storage mechanisms may take a long time to mature and become production-ready – e.g., before having sufficient features and bug fixings.

We, instead, seek to answer the question: Can existing *well-tested, production-ready* kernel file systems effectively evolve to harness performance benefits and new characteristics of emerging storage technologies, achieving the same properties as those device-specialized file systems while requiring *no* application modifications and radical system redesign?

To answer this question, we gather the main insights from PM/SSD-specialized approaches that focus on minimizing system software overhead. First, maximizing the performance advantages of fast storage devices involves avoiding as much work as possible in the critical path. For example, approaches like SplitFS [27] and Strata [28] use a userspace library to handle data (and metadata) operations directly, which are then asynchronously processed by the kernel file system. Second, fast storage devices, along with a lightweight journaling/logging mechanism, can enable strong consistency with little overhead. For instance, NOVA [43] ensures that each file system update is synchronously persisted in an atomic manner.

Third, PM or modern NVMe SSD, with the advent of the high-speed CPU-to-device interconnect technology like Compute Express Link [4], provide a *memory-like, byte-addressable* interface [26], offering new design and optimization opportunities, including efficient handling of small writes for file system updates without write amplification.

These insights lead us to reap fast, byte-addressable storage for legacy kernel file systems with a novel in-kernel caching mechanism, **P²CACHE**. P²CACHE exploits a new *read/write-distinguishable memory hierarchy* within a tiered memory system involving both PM and DRAM. P²CACHE leverages fast PM to serve *all* write requests for instant data durability and strong crash consistency while using DRAM to serve *most* read I/Os for high I/O performance because DRAM's read performance remains significantly higher than PM.

P²CACHE first introduces a *persistent cache* located below the VFS layer. The persistent cache uses PM to quickly and synchronously persist/buffer file system metadata/data updates, guaranteeing instant data durability and strong crash consistency. The buffered operations are then asynchronously applied to underlying kernel file systems via the existing I/O interface, ensuring compatibility with kernel file systems. The persistent cache is built upon a lightweight operation log that captures minimal operations (i.e., file system updates from the VFS) and records them in a write-ahead log. It leverages PM's byte-addressability to efficiently persist metadata/data updates without costly, block-based Copy-on-Write (CoW) (commonly used in PM-based approaches [19,23,43]) – by decoupling the copy operation from the write operation, where a write operation (of any size) is synchronously recorded while the data copy is performed asynchronously.

P²CACHE further advances the *page cache* to serve most reads via faster but volatile DRAM. To allow the two caches – namely, the persistent cache and page cache – to work collaboratively and efficiently, P²CACHE leverages "device-level" parallelism. Specifically, we observe that the I/O latency of writing data to both PM and DRAM (at the same time) is almost the same as that of writing data to PM only, because the latency of the (extra) copy to DRAM is hidden (overlapped) by the *parallel-but-slower* PM write. This leads us to adopt a simple and effective *inclusive cache model*, where multiple copies of the same data are stored across the tiered memory, and the topmost layer (i.e., DRAM) always contains the latest version. The inclusive cache model simplifies the synchronization between the two caches: For writes, P²CACHE updates both caches; for reads, P²CACHE searches from the page cache, persistent cache, and underlying file systems sequentially until it first finds the data.

The benefits of P²CACHE are manifold: (1) P²CACHE requires *no* modifications to user applications, libraries, or kernel file systems while leveraging fast storage technologies to provide high I/O performance, high I/O concurrency, instant data durability, and strong consistency for legacy kernel file systems. Meanwhile, kernel file systems can still operate with their own (slow) storage devices (e.g., HDD/SSD). (2) P²CACHE does not provide complex file system functionalities (e.g., maintaining in-memory/on-disk data structures or disk block management). Instead, it focuses on efficiently persisting and buffering file system updates using PM as a persistent cache. It is extremely lightweight and enables legacy kernel file systems to achieve higher performance than PM-specialized file systems (e.g., NOVA [43]). (3) Unlike existing PM-based approaches that fully bypass DRAM (or page cache), P²CACHE leverages DRAM to maximize I/O performance – i.e., although PM has similar write latency as DRAM, there is a considerable latency gap for reads (e.g., 3x slower [44]). (4) Persistently buffering file system operations (i.e., metadata/data) enables new system optimizations. For example, P²CACHE accelerates the performance of a *cold-start* file system (e.g., after re-mounting or recovering from a system crash) by quickly re-building its in-memory cache (e.g., dentry cache) from the persistent cache.

We have implemented P²CACHE as a Linux kernel module interfacing with the VFS layer. Our evaluation with microbenchmarks and applications shows that the read/write-distinguishable memory hierarchy allows P²CACHE to significantly increase the performance of legacy file systems (e.g., Ext4) by up to 200x for RocksDB [1] while providing instant data durability and strong crash consistency. P²CACHE also achieves higher I/O performance than existing PM-specialized file systems, e.g., by up to 70% for RocksDB to NOVA [43].

2 Motivation

2.1 Fast Storage and Interconnect

Enabled by new storage technologies, such as 3D XPoint [12], NVMe SSDs over the PCIe bus achieve much higher bandwidth (e.g., 8 GB/s under the 70/30 mixed read/write case) and lower latency (e.g., as low as 3μs) [45] than before.

Further, byte-addressable persistent memory (PM) has been commercially available in a DIMM package on the memory bus, e.g., Intel Optane DC persistent memory [10]. PM allows programs to directly access data in non-volatile memory from the CPU using `load` and `store` instructions. PM offers approximately an order of magnitude higher capacity than DRAM (e.g., 8x capacity in Optane DIMMs) and within an order of magnitude performance of DRAM [44] (e.g., as low as 80 ns for write latency and 170 ns for read latency).

Despite Intel discontinuing its Optane product, the storage community is actively embracing high-speed CPU-to-device interconnect technologies [11], such as Compute Express Link (CXL) [4]. CXL provides a more general, unified interface to disaggregate various types of storage devices (e.g., DRAM, PM, and PCIe devices) directly to the CPU. CXL has the potential to offer a memory-like, byte-addressable alternative (i.e., via `load` and `store` instructions) to PCIe storage's block interface with minor modifications [26]. We envision that this trend will continue – storage devices will be increasingly faster with higher bandwidth and lower latency and offer byte-addressability via a memory-like interface. Although our

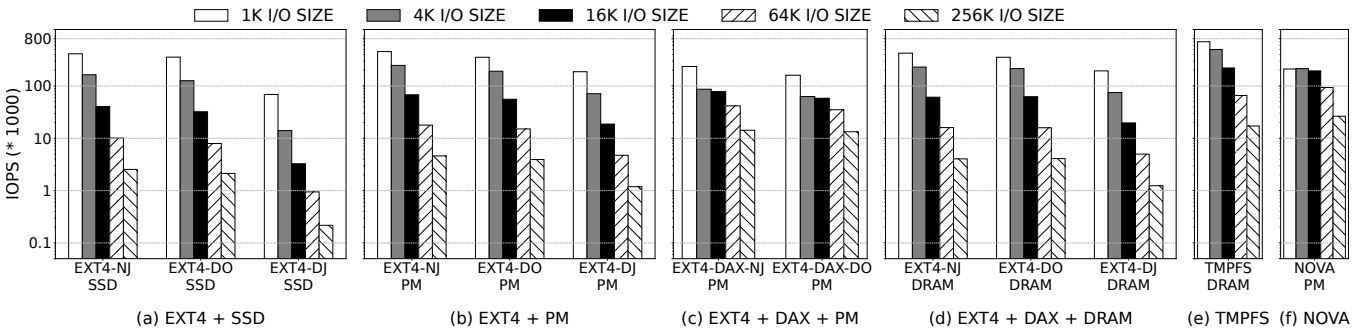


Figure 1: I/O performance comparisons for writes between cases with the combination of (1) distinct file systems: EXT4, EXT4-DAX, tmpfs, and NOVA; (2) journaling mode: no journal (NJ), data order (DO), and data journal (DJ); (3) storage medium: SSD, PM, and DRAM; and (4) various I/O sizes: 1 KB, 4 KB, 16 KB, 64 KB, and 256 KB.

work focuses on DIMM-based PM, it sheds light on bridging the gap between (1) byte-addressability, (2) a user-friendly and backward-compatible programming interface in PM and future CXL memory, and (3) the inherent differences between PM and traditional DRAM-based memory.

2.2 Kernel File Systems

Kernel file systems [7, 24, 29, 35, 37] have undergone continuous development and evolution, with the addition of features, bug fixes, and improvements in performance and reliability. For example, Ext4, the default general-purpose file system in most Linux distributions, was initially released in 2001.

The in-kernel storage stack converts I/O requests from applications into block operations that are persisted to storage devices through multiple software layers – i.e., the virtual file system (VFS), kernel file systems, generic block layers, journaling, and device drivers. While the layered design of the storage stack works well with slow underlying storage devices, such as HDD/SSD, it introduces nontrivial software overhead that becomes more pronounced as storage devices become faster. For example, in the case of low latency (3 us) NVMe SSD, software contributes 50% to read latency [45]. Our study on faster devices confirms this observation. *Observation 1: The in-kernel storage stack becomes the dominant storage bottleneck, rendering traditional kernel file systems unable to fully explore the potential of fast storage devices*. As depicted in Figure 1 (a), (b), and (d), Ext4 demonstrates only a marginal improvement in I/O performance (for writes) with PM (or even DRAM [1]) compared to SSD, despite the considerably faster speeds of PM/DRAM over SSD.

Virtual file system: Not all in-kernel storage layers contribute equally to the storage software overhead. The VFS, sitting atop the storage stack, incurs less than 10% of the overall software overhead [45]. Further, `tmpfs`, a lightweight kernel file system that works atop DRAM, achieves the highest performance (Figure 1(e)), indicating that a thin file system beneath the VFS can still leverage the benefits of fast storage media. On the other hand, the VFS implements key file system abstractions (e.g., `inodes` to represent metadata of on-disk files/directories) and functionalities (e.g., `pathname lookup` to locate a file/directory given a path name and `dcache` to cache such mappings in DRAM for quick lookup), commonly used by underlying file systems. *Observation 2: A storage layer (e.g., a persistent cache or page cache) – sitting below the VFS – can reuse VFS's rich functionality while being slim.*

Page cache: Because traditional disk accesses (e.g., HDD or SSD) are significantly slower than DRAM accesses, the operating system (OS) keeps frequently-accessed disk blocks in a dedicated region of DRAM, namely the *page cache*. The page cache reduces the number of disk accesses and speeds up I/O performance. However, as data updates are first applied to the page cache and later flushed to the storage, data modifications may not immediately reflect in the backing storage in case of sudden system crashes or power losses. This could cause an on-disk file system to enter an *inconsistent* state [34].

Journaling: To provide consistent and recoverable updates for metadata and/or data, kernel file systems often use *journaling* techniques. For example, Linux Ext4 employs the journaling block device version 2 (JBD2) [15], which implements a write-ahead log to record updates to a journal area before applying them to the corresponding file system locations. In the event of system failures, it replays the journal to restore the file system to a consistent state. JBD2 operates with three journaling modes: writeback, ordered, and data modes, providing trade-offs between performance and consistency. In writeback and ordered modes, only the metadata is journaled. However, the ordered mode (i.e., the default option) imposes an ordering requirement that data must be completed before the associated metadata is committed. This ordering constraint ensures stronger file system consistency than writeback, albeit with increased journaling latency [36]. On the other hand, the data mode journals both data and metadata, offering the highest level of consistency at the expense of the highest performance overhead.

Observation 3: Journaling places a serious impediment to the high performance of kernel file systems and can offset the performance benefits provided by the page cache. Figure 1 (a) and (b) show that in the ordered journal mode, the write performance under SSD and PM drops by ∼20% compared

[1] We used DRAM to emulate fast storage devices.

to the "no journal mode". The "data mode" substantially diminishes the write performance to only ∼10% of the no journal mode for SSD and ∼30% for PM. It is because (1) journaling requires writing metadata/data *twice*[2]: one to the log area and one to the file system location; though journaling is performed asynchronously, it can be frequently invoked by a background thread, competing for system resources with user applications. (2) JBD2 operates at the *block level* beneath the file system layer, recording all modified metadata/data blocks in a block unit within the journal area, even if only a small portion of the metadata/data blocks is modified (i.e., the small write case) – causing *write amplification*. (3) JBD2 journals all file system updates with a single kernel thread, limiting its scalability. In addition, the existing journaling scheme cannot ensure instant data durability. Instead, it relies on an explicit synchronous operation from users (e.g., `fsync()` or `fdatasync()`) for instant data durability.

2.3 Related Work

A large body of work has been recently proposed to exploit performance benefits and unique characteristics of fast storage technologies. We categorize them as two groups:

PM-specialized file systems: Many PM-specialized file systems have been studied [5, 18, 19, 23, 27, 28, 40, 42, 43]. They are tailored for the fast, byte-addressable PM to address challenges in write ordering and update atomicity, providing various levels of data/metadata consistency. For instance, BPFS [19] leveraged shadow paging for metadata/data consistency; PMFS [23] used journaling for metadata update atomicity, while performing in-place update for data (no atomicity and consistency); and NOVA [43] adopted a per-inode log-structured file system that offers synchronous persistence. Figure 1 (f) shows that NOVA achieves higher performance than Ext4 with the data journal mode (i.e., EXT4-DJ in Figure 1 (b)), while ensuring strong consistency. User-level PM-specialized file systems have been proposed to mitigate the overhead of system calls: Aerie [40] implemented a POSIX-like file system in userspace; Strata [28] appended updates to a userspace per-process log; and SplitFS [27] used a userspace library to handle data operations while still relying on the kernel-level file system to handle metadata operations.

Observation 4: the process of adapting existing storage systems to PM-specialized file systems remains in its early days. Making any of those projects production-ready needs substantial effort (and years) to achieve a combination of high performance, strong consistency, and comprehensive data protection. In addition, PM-specialized file systems unanimously bypass DRAM. While eliminating DRAM simplifies system design by allowing direct access to PM and avoiding synchronization complexity between PM and DRAM, it results in sub-optimal I/O performance. For example, the read latency of Intel Optane PM is *2x-3x* higher than that of DRAM, while the write bandwidth is only *1/6* [44]. Further compounding this situation, PM-specialized file systems usually trade I/O performance for write atomicity and consistency, for example, via log-structured file system techniques (cumbersome to reads) or shadow paging (causing write amplification) [19,43]. Figure 1 (f) shows that using Copy-on-Write (CoW) in NOVA to ensure data consistency incurs nontrivial overhead and proves detrimental to small writes (e.g., 1 KB).

[2]Although P^{2}CACHE also writes metadata and data twice, it has a very lightweight design with low overhead.

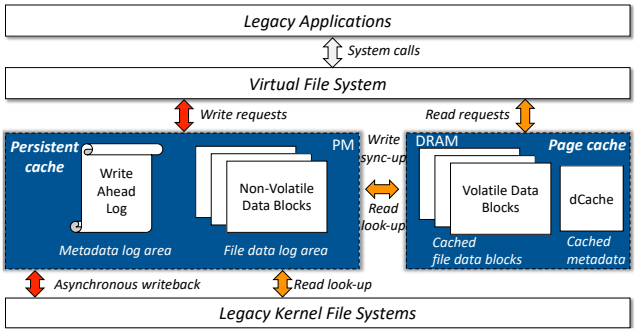


Figure 2: Overview of P^{2}CACHE.

PM-enhanced file systems: A straightforward approach for traditional block-based kernel file systems to adopt PM is to extend them with Direct Access (DAX), such as Ext4-DAX and XFS-DAX [5]. DAX-enabled kernel file systems bypass the page cache and perform reads/writes directly to PM. However, DAX-enabled file systems still operate at the block level and cannot explore PM's byte-addressability and its performance benefits. Figure 1 (c) shows that Ext4-DAX in the data-order mode achieves less than 50% of the performance of NOVA. Further, Ext4-DAX lacks support for the data journal mode and does not provide strong crash consistency.

Mostly related to P^{2}CACHE, efforts that explored the non-volatile nature of PM in building a cache layer have been conducted [30, 38]. They united either the journaling and caching [30] or the storage and caching [38] functionalities into a single cache layer. However, they are limited due to (1) like PM-specialized file systems, they eliminate DRAM despite its significant performance benefits for reads; (2) they only focus on caching data, while metadata operations remain on the slow path. Note that metadata can take up more than 50% of file system operations [3]; (3) they remain working at the block level (e.g., block-based CoW for data journaling or read-modify-write for partial writes), failing to leverage the byte-addressability of PM. *Observation 5: It is vital to have an OS caching approach that recognizes both data and metadata and distinct characteristics between PM and DRAM*.

3 P^{2}CACHE

We introduce P^{2}CACHE, a novel in-kernel caching mechanism. The goal of P^{2}CACHE is to enable the key properties of PM-specialized file systems for legacy kernel file systems, including instant data durability, strong consistency, high performance, and high concurrency while requiring *no* modifications to user applications, libraries, and kernel file systems. As illustrated in Figure 2, the key idea behind P^{2}CACHE is to

exploit a read/write-distinguishable memory hierarchy within the tiered PM/DRAM system, where P^{2}CACHE leverages fast PM to serve all write requests for instant data durability and strong crash consistency, while relying on DRAM to handle most read I/Os for high performance. P^{2}CACHE comprises two key kernel components: a persistent cache (Section 3.2) and a page cache (Section 3.3). Based on the observations in Section 2, P^{2}CACHE adopts the following key design choices.

3.1 Design Overview

A read/write-distinguishable memory hierarchy: We share the same observation as [41] that *the (modern) storage hierarchy is not a hierarchy* given the advent of fast, byte-addressable storage like PM. Unlike a traditional hierarchy where all I/O requests are first handled by the upper performance layer(s) and then consumed by the lower capacity layer(s), P^{2}CACHE distinguishes read and write operations in the PM/DRAM memory hierarchy with the goal to allow PM to handle all write requests while DRAM to serve most read I/Os because (1) P^{2}CACHE must persist each update in PM for instant data durability and strong crash consistency; and (2) DRAM's read performance is significantly higher than PM. To achieve this, P^{2}CACHE employs the following read/write strategies: (1) All writes are directed to PM, with a copy of the data modification also made in DRAM. It ensures that both PM and DRAM have the same data version. (2) Reads are first served from DRAM. If not found, P^{2}CACHE searches PM and underlying file systems. While P^{2}CACHE writes data to both PM and DRAM, it leverages *device-level parallelism* with *little* performance degradation – the latency of each data copy to DRAM is hidden by the parallel (slower) PM write.

A lightweight operation log: To harness the high performance of PM, P^{2}CACHE minimizes the operations in the critical path that involves PM. As shown in Figure 2, P^{2}CACHE's persistent cache resides at an early I/O layer, just below the VFS to leverage its general file system abstractions and functionalities while being slim. The role of the persistent cache is to capture all file system updates from the VFS (e.g., data overwrites/appends and metadata updates) and quickly, atomically persist them in an operation log stored in PM.

A PM-optimized logging mechanism: P^{2}CACHE's operation log in PM consists of two log areas: one for metadata updates and one for file data updates. First, P^{2}CACHE uses fixed-sized log entries (e.g., 64 bytes) to record metadata updates in the *metadata log area*. For data updates, P^{2}CACHE uses different strategies: (1) For *unaligned* overwrites (i.e., covering one or spanning across two *partial* data blocks), P^{2}CACHE directly appends the data to the end of its metadata update log entry for fast persistence. (2) For *aligned* overwrites (i.e., covering one or multiple contiguous full data blocks), P^{2}CACHE allocates free data blocks in the *file data log area* to store the data. (3) For *data appends*, P^{2}CACHE simply stores the appended data at the end of its data blocks in the file data log area (allocating additional blocks, if necessary). This approach ensures data consistency by never overwriting any old

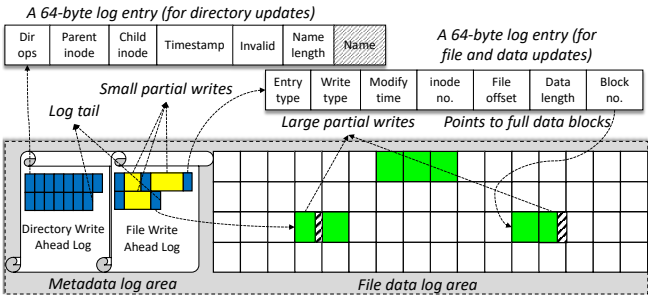


Figure 3: The layout of one CPU core's PM space.

data before the commit stage and avoiding the costly CoW operations commonly employed in other PM-based solutions.

Fast reads via in-DRAM indexes: While P^{2}CACHE's persistent cache benefits writes, it challenges reads. For example, a read operation may involve data scattered across various locations, and P^{2}CACHE may even create "holes" in the data blocks (in PM and/or DRAM) due to partial overwrites that are not aligned with the block size. To ensure fast reads, P^{2}CACHE leverages *in-DRAM* indexes to facilitate data search, including indexes for (1) log entries in PM's metadata log; (2) data blocks in PM's file data log area; (3) data blocks in the page cache; and (4) partial-write slots in the page cache. These indexes enable the rapid assembly of read content, even when it is distributed across multiple storage media. Importantly, these indexes reside exclusively in DRAM and can be quickly reconstructed from the persistent cache, such as during a system reboot or recovery.

3.2 Write-centric Persistent Cache

To achieve high performance, it is crucial to defer "writes" to slow storage for as long as possible. This is precisely why most kernel file systems employ DRAM-based caches, such as dcache for metadata and the page cache for file data, to expedite I/O operations. As depicted in Figure 2, P^{2}CACHE introduces a new OS component, a PM-based *persistent cache*, positioned beneath the VFS and above any legacy kernel file systems. It interfaces with the VFS to efficiently persist updates to file system data and metadata, ensuring crash consistency and quickly responding to user applications. Meanwhile, P^{2}CACHE relies on mature, well-tested underlying kernel file systems for data organization and management – i.e., the persisted metadata/data update operations in the persistent cache are eventually flushed back to underlying file systems.

3.2.1 Layout of PM

Operation log: P^{2}CACHE's persistent cache captures and records file system operations related to writes (i.e., metadata/data updates) in a PM-backed write-ahead operation log (WAL). As illustrated in Figure 3, a WAL is implemented as a circular buffer consisting of fixed-size log entries. As there are two types of operations, directory operations and file operations, P^{2}CACHE builds two WALs: the directory write-ahead-log (dWAL) and the file write-ahead-log (fWAL). To improve the concurrency of P^{2}CACHE, each CPU core has its own WALs to log updates on that core independently.

	VFS Interface
Directory	create(), link(), unlink(), symlink(), mkdir(), rmdir(), mknod(), update_time(), setattr(), rename()
File	write(), write_iter(), fsync(), flush() open() (access time might need to updated)

Table 1: Interface exposed by VFS for metadata/data updates.

Algorithm 1 Atomically persisting file system updates

```
1:  function PERSISTENCE(operation)
2:      log_entry := create_a_log_entry(operation);
3:      if is_directory_op then
4:          write_to_dWAL(log_entry);
5:      else
6:          write_to_fWAL(log_entry);
7:          data_persistence(log_entry);   //Section 3.2.3
8:      end if
9:      update_in_DRAM_indexes();      //Section 3.3.2
10:     sfence();
11:     update_log_tail();
12:     sfence();
13: end function
```

PM space management: P^{2}CACHE partitions the PM space into n groups, where n is the number of CPU cores. Each group is further divided into two main areas: the *metadata log area* for storing dWALs and fWALs and the *file data log area* for storing data updates, as depicted in Figure 3. In systems with multiple PM DIMMs, P^{2}CACHE employs the *interleaved* mode, which distributes contiguous data blocks across the DIMMs in an interleaved manner. The management of PM space allows P^{2}CACHE to achieve high concurrency: (1) P^{2}CACHE-related tasks can be independently handled on different cores; (2) Sequential reads/writes can be concurrently served by multiple PM DIMMs. While file and directory operations can be processed by different cores and stored across multiple WALs, these operations are inherently ordered by their time of occurrence. Each operation log entry in the WAL contains a timestamp, as depicted in Figure 3. This guarantees that logged operations are later consumed by the underlying file system in the exact order in which they were issued by user applications. P^{2}CACHE relies on the VFS to prevent conflicts arising from concurrent updates to the same directory or file (via the per-inode read-write lock). For example, while one thread is writing data to a file, all other threads attempting to read from or write to the same file must wait. As a result, writes to the same file are recorded sequentially across the WALs, and their orders are determined by their timestamps.

3.2.2 Durability and Crash Consistency

Instant metadata/data durability: Using the PM-backed operation log, P^{2}CACHE first ensures instant *data/metadata durability*. Any metadata/data updates are captured by the persistent cache and *synchronously* persisted in the WALs. P^{2}CACHE captures these updates via the VFS-exposed interface as listed in Table 1. For each update, one or more log entries are synchronously created to store such an update operation in either the dWAL or the fWAL for metadata durability. If the operation involves the file data update (e.g., `write()`), new data should also be synchronously stored for data durability (more details in Section 3.2.3). Note that achieving instant metadata/data durability in traditional kernel file systems requires user applications to explicitly invoke synchronous operations, such as `fsync()` or `fdatasync()`. For example, most database systems use `fsync()` to ensure immediate data durability. In contrast, with P^{2}CACHE's instant data durability, (1) those `fsync()` issued by legacy applications can be immediately returned; (2) P^{2}CACHE-aware applications can eliminate `fsync()`; the return of a file operation indicates that both metadata and data have been persisted.

Strong crash consistency: P^{2}CACHE further provides *strong crash consistency* similar to PM-specialized file systems [43]. P^{2}CACHE achieves this by ensuring that each file operation is *atomic* – i.e., updates made by the operation are committed in an all-or-none fashion. As described in Algorithm 1, for a *metadata* update, P^{2}CACHE appends the operation to the end of the dWAL/fWAL by creating a log entry (size of 64 bytes). Then, P^{2}CACHE atomically updates the log tail to commit the metadata update. For a *data* update, P^{2}CACHE first appends the operation in the fWAL. Depending on the type of the writes (partial or full-block), P^{2}CACHE stores the file data either to the fWAL by creating log entries or in the file data log area by allocating free data blocks (Section 3.2.3). Finally, the log tail will be updated to commit the data update. Note that as Optane PM only guarantees atomicity for an 8-byte update, ensuring the atomicity of updates larger than 8 bytes (e.g., metadata/data updates) requires P^{2}CACHE to atomically move the log tail to the end of the dWAL/fWAL, thus committing the update. To ensure correct write ordering and prevent the tail update from occurring before the metadata/data update, P^{2}CACHE uses two `sfence` instructions: one after WALs or file data are written, and one after the log tail is updated.

Compared to in-kernel file system journaling (e.g., JBD2), which offers "relaxed" consistency, P^{2}CACHE's strong consistency provides the following benefits: (1) Each file operation mostly involves updating a small log entry (e.g., 64 bytes), which is much lightweight, whereas JBD2 needs a complex transaction operation involving multiple (4 KB) blocks, such as a journal header, multiple descriptor blocks, and a journal commit block. (2) Since the logs in P^{2}CACHE are committed to PM, they are persistent. As PM has a much larger capacity (than DRAM), persisted data can stay for quite a long time. Hence, P^{2}CACHE can defer "writes" to the underlying slow file systems and storage devices as long as possible. (3) Once operations are asynchronously consumed by the underlying file system, many optimizations can be employed, such as coalescing repeated writes and removing obsolete data, similar to [28]. (4) As shown in Section 4, P^{2}CACHE (though with strong consistency) achieves much higher performance than legacy kernel file systems (with relaxed consistency) because P^{2}CACHE significantly mitigates software overhead.

While P^{2}CACHE can implement "relaxed" consistency by placing WALs in DRAM first and asynchronously flushing

them back to PM, its current focus lies in providing strong (synchronous) consistency to kernel file systems. Further, P^{2}CACHE does not support atomic `mmap`, as "mmapped" I/Os bypass the VFS and access "mmapped" files directly via `load`/`store` instructions. Currently, P^{2}CACHE relies on user applications to achieve instant data durability and strong consistency for `mmap`, while a modified (or new) `mmap` interface for update atomicity is our ongoing investigation.

3.2.3 Fine-grained, Highly-efficient Data Logging

Compared to metadata, persisting data operations can incur much higher overhead. To mitigate such overhead, P^{2}CACHE invents a fine-grained, highly-efficient data logging mechanism that leverages PM's byte-addressability.

To ensure consistency, one should never overwrite old data before committing its new update. Otherwise, if a crash happens in the middle of an overwrite, it may corrupt the old data, causing inconsistency. To consistently persist a file data update, an intuitive approach is to allocate *free* data blocks in PM for storing the new file data, record the addresses of these blocks, and finally commit the data update. Up to this point, the data blocks that store old data can be released. Reclaiming these blocks can be done asynchronously (Section 3.4).

Note that if the updated file data *aligns* perfectly with one or multiple block boundaries (e.g., 4 KB), no data copying is required. In this case, the new file data blocks simply replace the old data blocks once the update is committed. If the updated file data does not align with the data blocks, *partial updates* are involved. Existing approaches [19, 23, 43] use a CoW strategy to copy the old data to a new data block and then apply the partial updates. Unfortunately, this approach leads to write amplification and long write latency.

P^{2}CACHE addresses this issue by decoupling (and delaying) "copy" from "write" in a CoW operation, named *decoupled CoW*. Decoupled CoW distinguishes writes of different sizes. As depicted in Figure 3, there are two types of partial writes: (1) For a *small* partial write (< 2KB, assuming a block size of 4 KB), P^{2}CACHE first appends the write operation log entry to the end of the fWAL and then directly appends the data content after the log entry. Finally, P^{2}CACHE atomically updates the log tail to the end of the data content to commit the update. (2) For a *large* partial write ($\geq$ 2KB and < 4KB), P^{2}CACHE instead allocates a free block to store the content of the partial write. Similar to NOVA [43], P^{2}CACHE employs a red-black tree for tracking and allocating free blocks.

In both the aforementioned cases, P^{2}CACHE does *not* copy the old data in the write path, neither does it in the read path – P^{2}CACHE devises an approach to efficiently assemble distinct partial updates during reads (Section 3.3.2). P^{2}CACHE performs data copying to convert a partially updated block to a full block at any later time. For instance, when reclaiming space in PM (Section 3.4), P^{2}CACHE copies small partial writes from the fWAL to their data blocks in the file data log area, or fills the missing portion in the data block of the large partial write with old data. If such data blocks do not exist in PM, a read-modify-write (RMW) operation is invoked to copy the required data from the underlying file system.

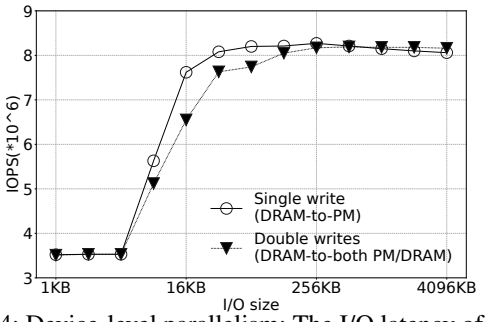


Figure 4: Device-level parallelism: The I/O latency of writing data to both PM and DRAM (i.e., one sfence after two writes: one to PM, and one to DRAM) is almost the same as that of writing data to PM only (i.e., one sfence after each PM write).

The decoupled CoW approach allows P^{2}CACHE to quickly persist partial updates by leveraging PM's byte-addressability. Real-world I/O traces show that a significant number of partial updates, ranging from 30% to 90% [33], commonly exist. In addition, by distinguishing writes by their sizes, P^{2}CACHE ensures that it requires copying *at most* half a block of data (e.g., 2KB with a block size of 4KB): P^{2}CACHE either copies the data of small partial writes (< 2KB) to their data blocks or the unmodified portion of old data (< 2KB) to the data blocks of large partial writes (> 2KB). Section 4 demonstrates that P^{2}CACHE, with the fine-grained data logging mechanism, achieves much higher performance for small writes than NOVA [43], a leading PM-specialized file system.

3.3 Read-centric Page Cache

P^{2}CACHE advances the page cache to handle most read I/Os in the tiered PM/DRAM hierarchy. P^{2}CACHE's page cache is a separate implementation other than the native page cache. It does not impact the behaviors of non-P^{2}CACHE supported kernel file systems, which still access the native page cache. To allow the persistent cache and P^{2}CACHE's read-centric page cache to work efficiently without comprising strong consistency, P^{2}CACHE employs a simple-and-effective *inclusive cache model* to exploit *device-level* parallelism.

3.3.1 Inclusive Cache Model

Similar to traditional caching mechanisms, P^{2}CACHE strives to maximize the hit ratio of the DRAM-based page cache. P^{2}CACHE employs an *inclusive cache model* where multiple copies of the same data can be stored across the tiered memory, and the topmost layer (i.e., DRAM) always contains the latest data version. It works as follows: (1) Given a *write* access, it will be persisted by the persistent cache (Section 3.2.2); meanwhile, a data copy will be made to the page cache before committing the update in PM. In this way, P^{2}CACHE allows the page cache to always have the latest version of all cached data. (2) Given a read access, P^{2}CACHE searches from the page cache (in DRAM), then the persistent cache (in PM), and finally the underlying file system until

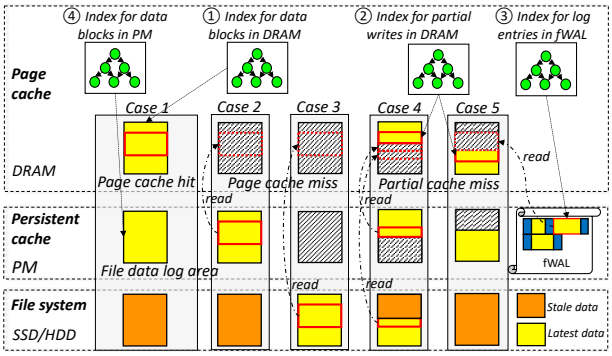


Figure 5: Fast read assembling via in-DRAM indexes.

it finds the data. For a page cache miss (e.g., the first time read or due to page eviction), the required data must be transferred either from the persistent cache or the underlying file system to the page cache, but data are never moved from the underlying file system to the persistent cache. (3) P^{2}CACHE uses LRU for cached page replacement, though other policies also apply. Evicted (dirtied) pages are dropped because any modifications have been recorded in the persistent cache.

Although P^{2}CACHE involves "double writes" for each data update, these two writes can be performed *in parallel* as they target separate devices: one for PM and one for DRAM. P^{2}CACHE can benefit from *device-level parallelism*. Figure 4 shows that the bandwidth (and latency) under "double writes" (i.e., writing to DRAM and PM at the same time) is very close to the "single-write" case (i.e., writing to PM only), indicating that the slower PM hides the latency of the extra data copy to DRAM as long as these two writes overlap each other. This way, P^{2}CACHE trades DRAM's bandwidth for simple synchronization between the two caches leading to a simplified read path. Note that DRAM has a much higher write bandwidth than PM (e.g., 6x [44]). Similar to the native page cache, P^{2}CACHE's page cache only uses "idle" DRAM and can grow/shrink. As all data on DRAM are in sync with PM, when the system memory pressure is high, the DRAM used by P^{2}CACHE can be reclaimed for other applications.

3.3.2 Fast Reads

While P^{2}CACHE's data logging (Section 3.2.3) greatly sharpens the write path, it brings new challenges to the read path due to: (1) While an operation log is efficient for updates, but not anymore for searching data (in the event of a page cache miss); (2) Decoupled CoW could leave holes in a page cache's data block – i.e., regions that are neither written by applications nor fetched from PM or underlying file systems.

In-DRAM indexes: To address these challenges, P^{2}CACHE uses in-DRAM indexes. P^{2}CACHE leverages Linux kernel's `XArray` – a memory-efficient, parallelizable treed data structure that performs lookups without locking – to create four per-`inode`, in-DRAM indexes (Figure 5) to track ① data blocks in the page cache; ② partial-write slots in the page cache; ③ log entries in the fWAL; and ④ data blocks in PM's file data log area. In consequence, in the *write* path, before committing a data update, P^{2}CACHE needs to insert the mapping information – between the updated data range and the log entry in the fWAL – in index ③. If a data block in the file data log area or the page cache has been allocated, index ④ or ① should be updated. If such an update involves a partial write, the offset and length of the partial write should be stored in index ②.

Assembling data for reads: With these in-DRAM indexes, the data content of a read request, specified by `offset` and `length`, can be quickly assembled as follows:

First, P^{2}CACHE uses the `offset` to query its per-inode index ① to check whether the data has been *fully* cached in the data block(s) of the page cache. If so, P^{2}CACHE returns the data of the requested length to user applications directly (e.g., case 1 in Figure 5). Otherwise, P^{2}CACHE uses the 2-tuple key `{offset, length}` to query index ② to check whether one or more partial slots in the range of the requested data exist in the page cache (e.g., case 4 & 5). If the aggregated partial slots do not cover the whole requested data, P^{2}CACHE moves to the persistent cache for the missing slots (e.g., case 4 & 5).

P^{2}CACHE uses the same 2-tuple key `{offset, length}` to query index ③ to get all log entries belonging to the queried data range, some of which may contain small partial writes (e.g., case 5). Further, by providing the `offset`, P^{2}CACHE retrieves the data block(s) stored in the file data log area via index ④ (e.g., case 2 & 4). Then, P^{2}CACHE copies the needed (missing) data slots – from the combined small partial writes (from the fWAL) and large partial writes (from PM's data blocks) – to the page cache (e.g., case 2, 4 & 5). If, unfortunately, there are still uncovered "holes" (e.g., case 3 & 4), P^{2}CACHE contacts the underlying file system for reading the needed data blocks to the page cache, taking longer time.

3.4 System Recovery and Digest

Rebuilding cache: P^{2}CACHE updates the log tail to commit operation-related records (Algorithm 1), indicating that all records preceding the log tail are considered valid. In case of a system crash or system remount, P^{2}CACHE discards any uncommitted records in the operation log (WALs). During system recovery/remount, *to facilitate fast reads*, P^{2}CACHE needs to scan the logs and build (1) two in-DRAM indexes, i.e., index ③ and index ④ (Section 3.3.2) and (2) a hash table (i.e., `dCache`). The process of scanning and building is considerably quick because logs are typically small (no data scanning is needed) and stored in fast PM. Table 2 shows that given a practical setup – for instance, with 10 thousand opened directories/files and 1 million log entries (typically multiple updates target one directory/file) – P^{2}CACHE uses a single core to recover index ③ within 33 ms while less than 17 ms to rebuild `dCache`. Moreover, the recovery of in-DRAM indexes can be made in parallel due to the design of per-core WALs – the time to rebuild ③ drops to 14 ms with 8 cores.

Digesting cache: P^{2}CACHE applies cached operations in PM to underlying file systems *asynchronously* via the existing I/O interface, namely the digest process [28]. The large capacity

# of log entries/files	10^4	10^5	10^6	10^7
Rebuild dCache (ms)	16.52	95.57	904.84	8009.02
Rebuild index ③ (ms)	0.89	4.44	32.92	320.50

Table 2: Time to rebuild dCache and in-DRAM indexes.

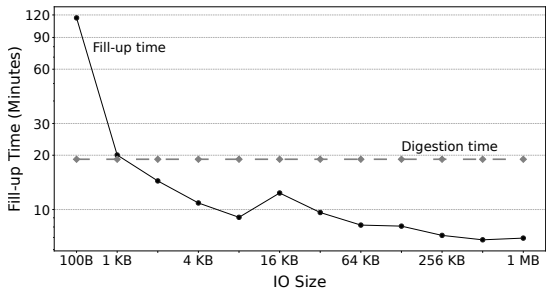


Figure 6: P^{2}CACHE filling-up time: P^{2}CACHE operates on a 512-GB PM, while the kernel file system (Ext4) runs upon one 2-TB SSD. A (sequential) write-intensive workload from Filebench [6] with 8 threads continuously issues I/O requests of various sizes at *maximum* speed. The PM's speed determines the cache fill-up time (the solid line), while the SSD's speed determines the digestion time (the dashed line). The digest process can merge adjacent write requests in the persistent cache and create large sequential I/Os writing to SSD.

of PM provides more flexibility to P^{2}CACHE by preventing the contention between the digest process and the normal working process. P^{2}CACHE digests operations during system idleness upon two conditions: the usage of PM space is high (e.g., more than 80%) or the number of log entries is large (e.g., more than 10 million). During the digest process, P^{2}CACHE relies on the return of `fsync` (after each operation) to ensure that metadata/data has been persisted and committed. A system crash may also occur during digest; P^{2}CACHE simply re-applies uncommitted operations. Similar to [28], P^{2}CACHE applies optimizations to coalesce multiple updates to the same file/directory during the digest process.

Filling cache: Given that (1) PM is generally faster than the storage devices on which the underlying kernel file systems operate (e.g., HDD/SSD) and (2) the speed of the digest process is mainly limited by the kernel file system's storage devices, the persistent cache of P^{2}CACHE can reach its full capacity. As a concrete example, Figure 6 shows the time required to fill up the persistent cache. When the persistent cache is full, P^{2}CACHE's current strategy is to throttle foreground I/O threads till the digest process reclaims enough space from PM (e.g., 20%). We note that as long as the *average* I/O rate is lower than the speed of the kernel file system's storage devices – or the fill-up time (the solid line in Figure 6) exceeds the digestion time (the dashed line) – the digest process can write back data timely. In practice, the occurrence of P^{2}CACHE reaching PM's full capacity is expected to be less frequent because (1) PM could be large (further CXL can aggregate even larger PM), and (2) I/O requests in a production environment arrive at moderate rates [38], typically lower than the speed of the kernel file system's storage devices.

4 Evaluation

We have implemented P^{2}CACHE as a Linux kernel module with ∼2000 lines of kernel code. P^{2}CACHE is available at https://github.com/YesZhen/P2CACHE. As an independent kernel module, P^{2}CACHE can be easily (un-)loaded without modifying other kernel components. P^{2}CACHE has passed all the test cases (∼7,000) of Linux official POSIX file system test suite [13], demonstrating P^{2}CACHE's POSIX compliance.

We have evaluated the effectiveness of P^{2}CACHE. Results from microbenchmarks demonstrate that (1) P^{2}CACHE accelerates metadata operations by *∼200x* against kernel file systems (e.g., Ext4 with `fsync`) and *3.5x* against PM-specialized file systems (e.g., NOVA). (2) P^{2}CACHE yields much higher write performance, particularly for *small, partial* writes – e.g., by *6.8x* than NOVA and *1,000x* than Ext4 (with `fdatasync`) for 1 KB writes. (3) P^{2}CACHE can leverage the DRAM-based page cache to achieve higher read performance – by *1.5x* than NOVA. The performance benefits brought by P^{2}CACHE further contribute to the improved application-level performance – e.g., by *72%* to NOVA for RocksDB's `insert` operations.

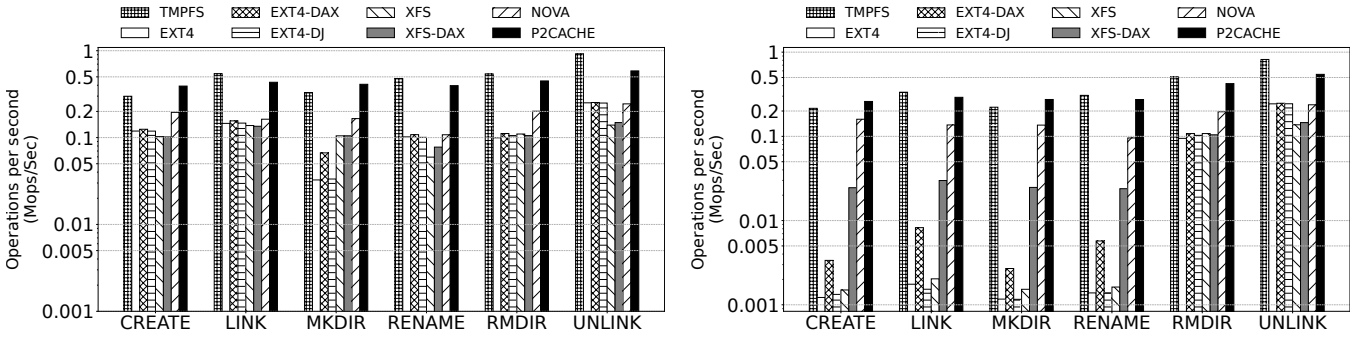
Experimental setup: The experiments were conducted on an ASUS RS700-E10-RS12U server equipped, with two 12-core Intel Xeon Gold 5317 processors (3.0 GHz and 18M Cache) with 2 NUMA nodes each with 256 GB DRAM. Hyperthreading was disabled while turbo boost was enabled. We installed four 128 GB (totaling 512 GB) Intel Optane 200 series persistent memory for each NUMA node and one 2-TB Samsung PM883 SSD. Since our focus is not on the NUMA effect, all experiments were conducted on one NUMA node.

We evaluated P^{2}CACHE on the Linux kernel 5.4, comparing it with (1) two kernel file systems, Ext4 and XFS, both operating on the SSD with the default metadata journaling mode and the data journaling mode for Ext4 (i.e., Ext4-DJ); (2) two PM-enhanced file systems, Ext4-DAX [5] and XFS-DAX [25], operating on PM; and (3) one PM-specialized file system, NOVA [43] (in strict mode) also operating on PM. We tested P^{2}CACHE atop Ext4 as the underlying kernel file system, though P^{2}CACHE can run atop any kernel file system. We evaluated P^{2}CACHE with both microbenchmarks and real-world applications. We have developed our own microbenchmarks to delicately generate desired I/O requests and patterns to test various design aspects of P^{2}CACHE. We selected three representative real-world applications for testing: Filebench [6], RocksDB [1], and MinIO [14].

4.1 Microbenchmarks

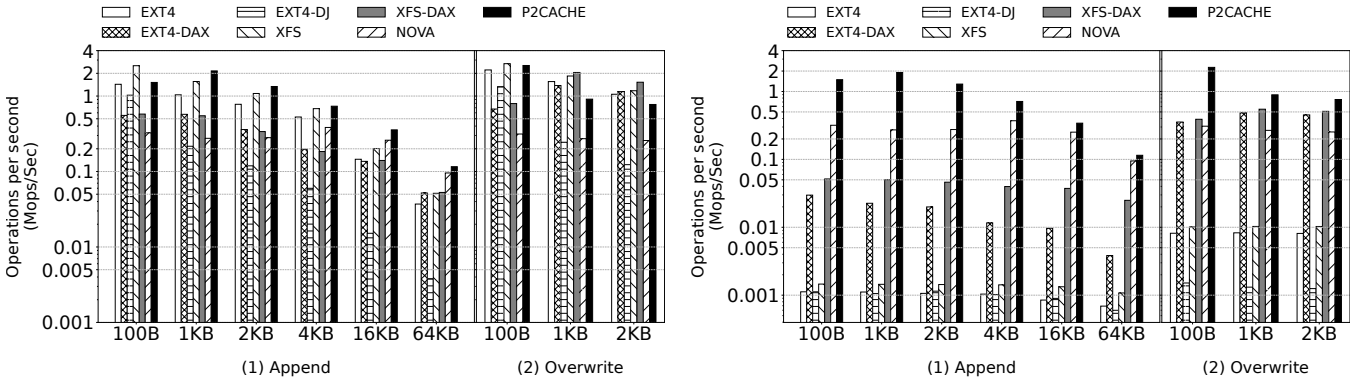
Metadata operations: We first show how P^{2}CACHE benefits metadata operations. We chose the six most complex ones in Figure 1, i.e., `create`, `link`, `mkdir`, `rename`, `rmdir`, and `unlink`. For each type of metadata, our micro-benchmark kept issuing the operations sequentially – i.e., the subsequent one was issued upon the completion of the previous one.

Figure 7a shows that P^{2}CACHE significantly accelerates the speed of all six metadata operations compared to all other cases, except for `tmpfs`. For example, for the most complex



(a) Without `fsync`. (b) With `fsync`.

Figure 7: P^{2}CACHE significantly accelerates metadata operations as against other cases except for `TMPFS`.



(a) Without `fdatasync`. (b) With `fdatasync`.

Figure 8: P^{2}CACHE accelerates data operations, especially for small, partial writes, as against other cases.

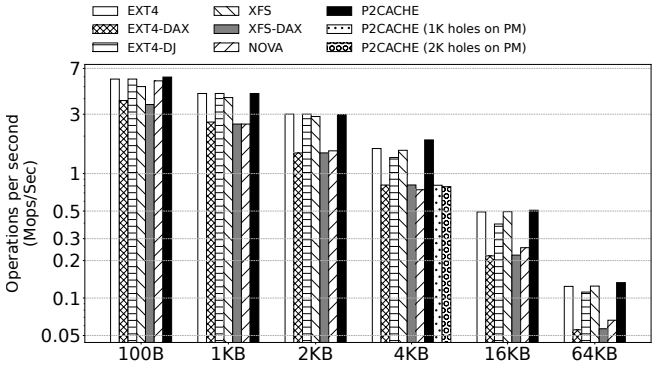


Figure 9: Comparisons of performance for reads.

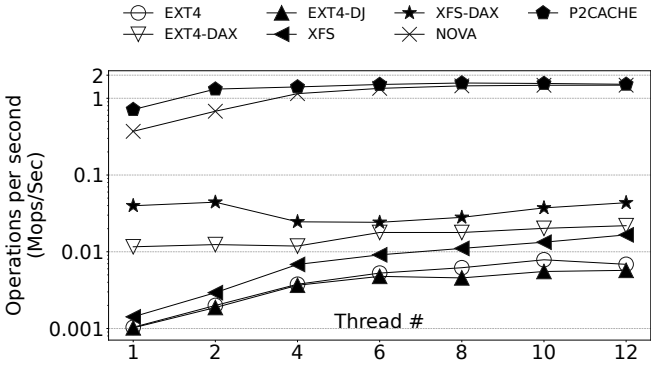


Figure 10: Scalability test with 4 KB append operations.

operation `rename` (i.e., involving multiple inodes), P^{2}CACHE achieves *∼4x* the performance of NOVA and Ext4 (in terms of operations/second). It is because P^{2}CACHE keeps the critical path (involving PM) extremely short by simply storing a log entry that represents the `rename` operation in the dWAL. In contrast, NOVA requires the creation of multiple logs and updates to multiple log entries, while Ext4 involves more operations (i.e., first `unlink` and then `link`). As another example, P^{2}CACHE enhances the performance of `mkdir` by a factor of *12x* and *6x* compared to Ext4 and Ext4-DAX. Figure 7a also shows that the performance gap between P^{2}CACHE and `tmpfs` is narrow – `tmpfs` is an extremely simple kernel file system that works upon DRAM. P^{2}CACHE is mostly within *80%* the performance of `tmpfs`; P^{2}CACHE even outperforms `tmpfs` for `create` (1.3x) and `mkdir` (1.25x). The main reason lies in that, instead of using the default heavyweight "inode_init_always" function, P^{2}CACHE implements its optimized one by initializing (fewer) needed fields. Again, as a caching mechanism, P^{2}CACHE can be lighter than full-fledged file systems, even including `tmpfs`.

Figure 7b shows a *strong consistency* scenario by issuing `fsync` after each metadata operation. Note that P^{2}CACHE and NOVA provide strong consistency in nature; they return `fsync` without any action. With strong consistency, ex-

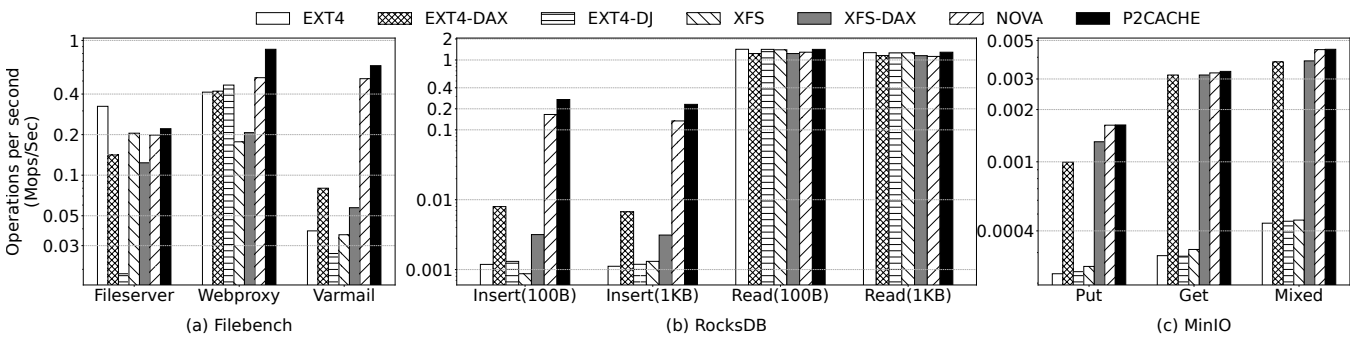


Figure 11: Performance comparisons of using real-world applications (a) Filebench, (b) RocksDB, and (c) MinIO.

cept for P^{2}CACHE, NOVA, and `tmpfs`, the performance of all other approaches drops significantly, where P^{2}CACHE outperforms them by up to *200x*. Noticeably, the performance of P^{2}CACHE and NOVA drops with `fsync` (Figure 7b) compared to the case without `fsync` (Figure 7a). It is because the `fsync` system call (though a no-op) incurs higher software overhead. As all the approaches rely on VFS's `dCache` for caching metadata, they achieve the same performance for read-related metadata operations (not listed).

Data operations: Next, we demonstrate how P^{2}CACHE benefits data operations. Figure 8a shows that in most cases P^{2}CACHE significantly outperforms other PM-based approaches in write performance – under both `append` (i.e., sequentially writing to the end of a file) and `overwrite` (i.e., sequentially overwriting existing content) – due to its lightweight design for the write path. Particularly, P^{2}CACHE achieves high performance for partial writes (e.g., for 100 bytes, 1 KB, and 2 KB) due to its *fined-grained, highly-efficient* data logging mechanism (Section 3.2.3). For example, the performance of `append` (for 1 KB) under P^{2}CACHE is 7.8x, 3.7x, and 3.9x as high as NOVA, Ext4-DAX, and XFS-DAX. Note that, only P^{2}CACHE and NOVA provide strong consistency, while Ext4-DAX and XFS-DAX only provide metadata consistency. Even with strong consistency, in many cases (except for the 100B case for `append` and 1KB & 2KB cases for `overwrite`), P^{2}CACHE achieves higher performance than kernel file systems (Ext4 and XFS), where the `append` and `overwrite` operations are directly applied to DRAM-based page cache. It indicates that P^{2}CACHE greatly reduces software overhead. The reason that the 100 B `append` case does not perform as well as other partial-write cases (e.g., 1 KB and 2 KB) lies in that the I/O size of 100 bytes is not aligned with PM's physical media access granularity, i.e., 256 bytes, causing write amplification and inefficiency [44].

Similarly, Figure 8b shows a strong consistency scenario by issuing `fdatasync` after each `append` or `overwrite` operation. The performance gap between P^{2}CACHE and others (except NOVA) widens significantly – e.g., P^{2}CACHE outperforms Ext4/XFS by more than *1,000x* and Ext4-DAX/XFS-DAX by more than *10x* for small writes. The poor performance of the kernel file systems is due to (1) slow SSD and

Metadata/data ops	Initial state	Steady state	Steady/initial
CREATE	259,697	300,778	+15.8%
LINK	290,852	276,509	-4.9%
MKDIR	274,165	308,090	+12.4%
RENAME	273,929	250,650	-8.5%
RMDIR	422,855	376,619	-10.9%
UNLINK	544,964	513,720	-5.7%
Append (100 B)	1,417,632	1,428,365	+0.8%
Append (64 KB)	114,357	128,625	+12.5%
Overwrite (100 B)	2,266,327	2,297,884	+1.4%
Overwrite (64 KB)	118,416	132,006	+11.5%

Table 3: Performance comparisons between the initial and steady states: metadata (Mops); data (IOPS).

(2) high overhead of file system journaling (Section 2.2).

For P^{2}CACHE's read performance, we tested Case 1 and 5, as listed in Figure 5. For Case 1, where all data was cached in the page cache, we measured the performance of sequential reads with various I/O sizes ranging from 100 B to 64 KB. For Case 5, where partial data was stored in DRAM and partial data was in PM, we randomly created numerous 1 KB or 2 KB "holes" in the data blocks of the page cache (averaging one hole per 4 KB) and measured the performance of sequential reads with the I/O size of 4 KB.

Figure 9 shows that, for case 1, P^{2}CACHE achieves the same (or slightly better) performance as (than) those which can leverage the DRAM-based page cache, e.g., Ext4, Ext4-DJ, and XFS. However, the difference is that P^{2}CACHE also provides strong consistency, while others do not. In contrast, P^{2}CACHE outperforms other PM-related approaches that bypass the page cache, e.g., by 1.5x compared to NOVA (for 4 KB reads). For case 5, P^{2}CACHE quickly assembled each 4 KB read from both DRAM and PM (Section 3.3.2). However, the read performance was limited by the slower device – e.g., PM. For example, P^{2}CACHE achieves the same read performance as NOVA, Ext-DAX, and XFS-DAX – the PM's speed limited the read performance.

Steady-state performance: We studied the steady-state performance by first simulating a steady-state scenario of P^{2}CACHE through running a mix of metadata/data operations until the persistent cache gradually reaches its 60% capacity with around 1 million files. We then measured the performance by running the above six metadata operations and append/overwrite data operations with strong consistency

(`fsync` or `fdatasync` after each operation) and compared the steady-state performance with that of the "initial state", where the persistent cache is empty. Table 3 demonstrates that the performance variation mostly falls within 15%.

Concurrency and scalability: We measured the scalability of P^{2}CACHE by running an increasing number of concurrent threads accessing different files on separate cores (up to 12 as one NUMA node has 12 cores). The results in Figure 10 show that both P^{2}CACHE and NOVA scale well as the number of threads increases until they reach the peak PM performance (for 4 KB `appends`). P^{2}CACHE achieves peak performance much faster than NOVA due to, again, its lightweight design.

Consistency checks: We developed a consistency checker to empirically generate test cases to examine whether the strong consistency property provided by P^{2}CACHE holds. The checker added "crash points" along P^{2}CACHE's persistence path (Algorithm 1) with three cases (1) inserting the crash point before the first `sfence`; (2) between the two `sfences`; and (3) after the second `sfence`. For Case 1, the operation should not be atomically persisted as the log tail is not updated; for Case 3, the operation should be persisted as the log tail is updated; for Case 2, the operation may or may not be persisted. The checker examines that if the operation is persisted, the final state should match the expected state (a priori knowledge), while if the operation is not persisted, nothing should be recorded (i.e., none or nothing). For all the microbenchmark tests, we also used the checker to perform consistency checks without observing any violations. We note that such consistency checks are incomplete and unable to explore all possible test cases. We leave the investigation of a more comprehensive method in future work.

4.2 Real-world Applications

Filebench: To evaluate the performance of P^{2}CACHE with real-world applications, we first selected three Filebench workloads [6]: (1) a write-intensive workload, *fileserver* (1:2 read/write ratio); (2) a read-intensive workload, *webproxy* (5:1 read/write ratio); and (3) a read/write balanced workload, *varmail* (1:1 read/write ratio). For all cases, the average read size was 1 MB, and the average write size was 16 KB. We added another type of write with the I/O size of 1 MB for fileserver. We fixed the thread number to 8 for all cases.

Figure 11 (a) shows that P^{2}CACHE consistently outperforms other PM-based approaches (e.g., NOVA, Ext4-DAX, and XFS-DAX), especially for read-intensive test cases, e.g., webproxy and varmail, due to P^{2}CACHE's read/write distinguishable memory hierarchy which leverages both PM and DRAM. For example, P^{2}CACHE outperforms NOVA by 60% for webproxy and 20% for varmail. P^{2}CACHE achieves lower performance than Ext4 for fileserver with intensive write operations, as Ext4 leverages faster DRAM-based page cache while P^{2}CACHE persists data in slower PM for writes. However, Ext4 does not provide strong consistency. P^{2}CACHE outperforms Ext4-DJ (with data journaling enabled) by *10x*.

RocksDB: We then used *db_bench* [1] – RocksDB's official benchmark tool – to evaluate P^{2}CACHE for RocksDB (a key-value store). RocksDB's architecture is highly concurrent for reads but not for writes [8]. Therefore, for writes, we focused on a single-threaded *synchronous* case by randomly inserting 10 million records to RocksDB; for reads, we focused on a multi-threaded random case with 8 threads to randomly read 10 million key-value records from RocksDB. We prepared a dataset with 10 million records. We fixed the key size to 20 bytes and evaluated two value sizes – 100 B and 1KB – a common case in RocksDB.

Figure 11 (b) shows that P^{2}CACHE outperforms all other approaches for small writes (i.e., `insert`) – e.g., by ∼72% to NOVA, ∼33x to Ext4-DAX, and ∼200x to Ext4. Note that, the extremely poor performance of Ext4 (though using the native page cache) is due to (1) synchronous insert operations, which persist data on slow SSD; and (2) read-modify-write caused by unaligned writes (e.g., 100 B or 1 KB are not aligned with 4 KB block size). As we purposely conducted the tests of reads after `insert` to have all records stored in the page cache, all the approaches (e.g., P^{2}CACHE, Ext4, Ext4-DJ, and XFS) that can leverage the page cache achieve the same performance – higher than PM-based approaches that bypass DRAM.

MinIO: Last, we evaluated P^{2}CACHE using an object storage, MinIO [14]. Compared to the above applications (e.g., RocksDB and Filebench), MinIO's software I/O path is much longer with extra data management (e.g., data checksum and placement). We used MinIO's official benchmark tool *warp* [16] with three workloads: `put`, `get`, and `mixed` (45% get, 30% stat, 15% put, and 10% deletion operations). We used log2 to distribute object sizes – i.e., objects are distributed in equal numbers for each doubling of the size.

Figure 11(c) shows that P^{2}CACHE and NOVA achieve the same performance under all cases and outperform other approaches by a range between 5% (over Ext4-DAX/XFS-DAX for `get`) and 10x (over Ext4/Ext4-DJ for `get`). We observed that software overhead from MinIO became dominant; neither P^{2}CACHE nor NOVA can exploit the full capacity of PM.

5 Conclusions

We have presented P^{2}CACHE, an in-kernel caching mechanism, which harnesses performance benefits and unique characteristics of fast, byte-addressable PM for legacy kernel file systems. P^{2}CACHE works upon a read/write-distinguishable memory hierarchy that leverages PM to persist writes and DRAM to handle reads, thus equipping kernel file systems with the key properties similar to PM-specialized file systems, including instant data durability, strong consistency, high concurrency, and high performance. Our evaluation with both microbenchmarks and applications shows that P^{2}CACHE significantly increases the performance of legacy kernel file systems, and even higher than PM-specialized file systems.

6 Acknowledgments

We thank our shepherd Mike Mesnier and the anonymous reviewers for their helpful feedback. This work was supported by NSF under Awards CCF-1845706 and CNS-2237966.

References

[1] Benchmarking rocksdb. https://github.com/EighteenZi/rocksdb_wiki/blob/master/Benchmarking-tools.md.

[2] Build ultra high-performance storage applications with the storage performance development kit. https://spdk.io/.

[3] Ceph fs dynamic metadata management. https://docs.ceph.com/en/latest/cephfs/dynamic-metadata-management/.

[4] Compute express link™: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/download-the-specification.

[5] Direct access for files. "https://www.kernel.org/doc/html/latest/filesystems/dax.html".

[6] Filebench. https://github.com/filebench/filebench.

[7] How windows ntfs finally made it into linux. https://www.theregister.com/2021/10/13/how_ntfs_finally_made_it/.

[8] Improving rocksdb's write scalability counting things at smyte. https://www.heavybit.com/library/article/improving-rocksdbs-write-scalability-counting-things-at-smyte.

[9] Intel optane dc ssd series. https://www.intel.com/content/www/us/en/products/details/memory-storage/data-center-ssds/optane-dc-ssd-series.html.

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

[11] Intel optane is winding down. what's that mean for you your customers? https://www.techproviderzone.com/cloud-and-data-centers/intel-optane-is-winding-down-what-s-that-mean-for-you-your-customers.

[12] Intel® optane™ ssd p5800x series. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html.

[13] Linux posix file system test suite. https://lwn.net/Articles/276617/.

[14] Multi-cloud object storage. https://min.io/.

[15] *The Linux Journalling API*. https://www.kernel.org/doc/html/latest/filesystems/journalling.html.

[16] Warp. https://github.com/minio/warp.

[17] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference* (New York, NY, USA, 2013), SYSTOR '13, Association for Computing Machinery.

[18] CHEN, Y., SHU, J., OU, J., AND LU, Y. Hinfs: A persistent memory file system with both buffering and direct-access. *ACM Trans. Storage 14*, 1 (apr 2018).

[19] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, Association for Computing Machinery, p. 133–146.

[20] CORBET, J. The multiqueue block layer. LWN.net.

[21] CORBET, J. Two new block i/o schedulers for 4.12. LWN.net.

[22] CORBET, J. The future of dax. LWN.net.

[23] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. EuroSys '14, Association for Computing Machinery.

[24] HELLWIG, C. Xfs for linux. In *Proceedings of Linux 2003 Conference and Tutorials, Edinburgh, Scotland* (2003).

[25] INTERNATIONAL., S. G. Xfs: A high-performance journaling filesystem. In *http://oss.sgi.com/projects/xfs*.

[26] JUNG, M. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). HotStorage '22, Association for Computing Machinery, p. 45–51.

[27] KADEKODI, R., LEE, S. K., KASHYAP, S., KIM, T., KOLLI, A., AND CHIDAMBARAM, V. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 494–508.

[28] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 460–477.

[29] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 273–286.

[30] LEE, E., BAHN, H., AND NOH, S. H. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (USA, 2013), FAST'13, USENIX Association, p. 73–80.

[31] LI, S. block: An iops based ioscheduler. LWN.net.

[32] LIU, J., REBELLO, A., DAI, Y., YE, C., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 819–835.

[33] LU, H., SALTAFORMAGGIO, B., XU, C., BELLUR, U., AND XU, D. Bass: Improving i/o performance for cloud block storage via byte-addressable storage stack. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (2016), pp. 169–181.

[34] MA, A., DRAGGA, C., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND MCKUSICK, M. K. Ffsck: The fast file-system checker. *ACM Trans. Storage 10*, 1 (jan 2014).

[35] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., AND VIVIER, L. The new ext 4 filesystem : current status and future plans.

[36] PARK, D., AND SHIN, D. iJournaling: Fine-Grained journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, July 2017), USENIX Association, pp. 787–798.

[37] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (2013), 1–32.

[38] SONG, H., KIM, S., KIM, J. H., PARK, E. J., AND NOH, S. H. First responder: Persistent memory simultaneously as high performance buffer cache and storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 839–853.

[39] VALENTE, P., AND ANDREOLINI, M. Improving application responsiveness with the bfq disk i/o scheduler. In *Proceedings of the 5th Annual International Systems and Storage Conference* (New York, NY, USA, 2012), SYSTOR '12, Association for Computing Machinery.

[40] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, Association for Computing Machinery.

[41] WU, K., GUO, Z., HU, G., TU, K., ALAGAPPAN, R., SEN, R., PARK, K., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (2021), pp. 307–323.

[42] WU, X., QIU, S., AND NARASIMHA REDDY, A. L. Scmfs: A file system for storage class memory and its extensions. *ACM Trans. Storage 9*, 3 (aug 2013).

[43] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 323–338.

[44] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 169–182.

[45] ZHONG, Y., LI, H., WU, Y. J., ZARKADAS, I., TAO, J., MESTERHAZY, E., MAKRIS, M., YANG, J., TAI, A., STUTSMAN, R., AND CIDON, A. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 375–393.

A Artifact Appendix

Abstract

The artifact contains the source code of P^{2}CACHE required to reproduce the results and figures presented in the paper. The code is designed to work upon Intel Optane PMem 200 series. To facilitate the reproduction of the results, we provide a collection of scripts for compiling and installing P^{2}CACHE, executing the experiments, collecting logs, and creating graphs. More details are available in the "README.md" file.

A.1 Description & Requirements

A.1.1 How to access

- **Link**: https://github.com/YesZhen/P2CACHE.git
- **Artifact license**: GNU GPL V3.0
- **Artifact version**: v0.0

A.1.2 Dependencies

For information on the hardware/software requirements needed to run P^{2}CACHE, please refer to "README.md".

A.1.3 Benchmarks

The experiments are carried out using several third-party benchmarking tools and applications, including FxMark, Filebench, db_bench/RocksDB, and warp/MinIO.

A.2 Testbed Setup

For instructions on how to set up and configure the test machine, please refer to the "README.md" file.

A.3 Evaluation

A.3.1 Major Claims

We summarize the major claims (Cx) in the paper as follows.

- (**C1**): P^{2}CACHE accelerates metadata operations, e.g., by *∼200x* against kernel file systems (e.g., Ext4) and *∼3.5x* against PM-specialized file systems (e.g., NOVA).
- (**C2**): P^{2}CACHE achieves much higher write performance especially for *small, partial* writes, e.g., by *6.8x* than NOVA and *1,000x* than Ext4 (with `fdatasync`) for 1 KB writes.
- (**C3**):P^{2}CACHE can leverage DRAM-based page cache to achieve high read performance, e.g., by *1.5x* than NOVA.
- (**C4**): The performance benefits brought by P^{2}CACHE further contribute to improved application-level performance – e.g., by *72%* to NOVA for RocksDB's `insert`.

A.3.2 Experiments

To reproduce the results presented in this paper, please refer to the "README.md" file and follow the instructions provided in the "Reproduce results from the paper" section.

Experiment (E1): Metadata Operations (without `fsync`)
Expected outcome. E1 produces the results as shown in Figure 7a, which illustrates that P^{2}CACHE significantly accelerates the speed of all six metadata operations (i.e., `create`, `link`, `mkdir`, `rename`, `rmdir`, and `unlink`) when compared to all other approaches (i.e., `Ext4`, `Ext4-DJ`, `XFS`, `Ext4-DAX`, `XFS-DAX`, and `NOVA`), except for `tmpfs`.

Experiment (E2): Metadata Operations (with `fsync`)
Expected outcome. E2 creates the results of Figure 7b. It demonstrates a strong consistency case by issuing `fsync` after each metadata operation. Except for P^{2}CACHE, `NOVA`, and `tmpfs`, the performance of all other approaches (i.e., `Ext4`, `Ext4-DJ`, `XFS`, `Ext4-DAX`, `XFS-DAX`) drops significantly.

Experiment (E3): Write Operations (no `fdatasync`)
Expected outcome. E3 generates the results as depicted in Figure 8a. It shows that in most cases (across various I/O sizes), P^{2}CACHE outperforms other PM-based approaches (i.e., `NOVA`, `Ext4-DAX`, and `XFS-DAX`) for two write operations: `append` and `overwrite`.

Experiment (E4): Write Operations (with `fdatasync`)
Expected outcome. E4 produces the results as depicted in Figure 8b. It demonstrates a strong consistency scenario by issuing `fdatasync` after each `append` or `overwrite` operation. P^{2}CACHE outperforms all other approaches in terms of higher `append` and `overwrite` performance.

Experiment (E5): Read Operations
Expected outcome. E5 creates the results as demonstrated in Figure 9. It shows that P^{2}CACHE achieves the same (or slightly better) performance as (than) those which can leverage the DRAM-based page cache (i.e., `Ext4`, `Ext4-DJ`, and

XFS). Further, P^{2}CACHE outperforms all other PM-based approaches (i.e., NOVA, Ext4-DAX, XFS-DAX) in terms of higher read performance.

Experiment (E6): Scalability
Expected outcome. E6 generates the results in Figure 10, showing that both P^{2}CACHE and NOVA scale well as the number of threads increases until reaching the peak performance, whereas P^{2}CACHE achieves the peak faster than NOVA.

Experiment (E7): Application: Filebench
Expected outcome. E7 produces the results as shown in Figure 11(a). It runs three workloads with Filebench, which are fileserver, webproxy, and varmail. The results show that P^{2}CACHE consistently outperforms other PM-based approaches (i.e., NOVA, Ext4-DAX, and XFS-DAX) in terms of higher application-level performance (operations/second).

Experiment (E8): Application: RocksDB
Expected outcome. E8 creates the results in Figure 11(b). It shows that for the insert operations (with sizes of 100 B and 1 KB), P^{2}CACHE outperforms all other approaches. All the approaches (i.e., P^{2}CACHE, Ext4, Ext4-DJ, and XFS) that can leverage page cache achieve similar read performance – slightly higher than other PM-based approaches that bypass DRAM (i.e., NOVA, Ext4-DAX, and XFS-DAX).

Experiment (E9): Application: MinIO
Expected outcome. E9 produces the results in Figure 11(c). The results show that P^{2}CACHE and NOVA achieve the same performance under all test cases and outperform other approaches (e.g., Ext4, Ext4-DJ, Ext4-DAX, XFS, and XFS-DAX).