TPFS: A High-Performance Tiered File System for Persistent Memories and Disks

SHENGAN ZHENG, Shanghai Jiao Tong University, China
MORTEZA HOSEINZADEH and STEVEN SWANSON, University of California, San Diego, USA
LINPENG HUANG, Shanghai Jiao Tong University, China

Emerging fast, byte-addressable persistent memory (PM) promises substantial storage performance gains compared with traditional disks. We present TPFS, a tiered file system that combines PM and slow disks to create a storage system with near-PM performance and large capacity. TPFS steers incoming file input/output (I/O) to PM, dynamic random access memory (DRAM), or disk depending on the synchronicity, write size, and read frequency. TPFS profiles the application's access stream online to predict the behavior of file access. In the background, TPFS estimates the "temperature" of file data and migrates the write-cold and read-hot file data from PM to disks. To fully utilize disk bandwidth, TPFS coalesces data blocks into large, sequential writes. Experimental results show that with a small amount of PM and a large solid-state drive (SSD), TPFS achieves up to 7.3× and 7.9× throughput improvement compared with EXT4 and XFS running on an SSD alone, respectively. As the amount of PM grows, TPFS's performance improves until it matches the performance of a PM-only file system.

CCS Concepts: • **Information systems** → **Phase change memory**; **Hierarchical storage management**; • **Software and its engineering** → **File systems management;**

Additional Key Words and Phrases: File system, persistent memory, data migration

ACM Reference format:
Shengan Zheng, Morteza Hoseinzadeh, Steven Swanson, and Linpeng Huang. 2023. TPFS: A High-Performance Tiered File System for Persistent Memories and Disks. *ACM Trans. Storage* 19, 2, Article 20 (March 2023), 28 pages.
https://doi.org/10.1145/3580280

20

1 INTRODUCTION

The recent advent of persistent memory (PM) [5, 22, 30, 32, 42] dramatically improves the performance of storage systems. It offers vastly higher throughput and lower latency compared with traditional block-based storage devices. File system design for persistent memory has been an active research topic for several years. Researchers have proposed several file systems [9, 13, 14, 39, 43]

This work was supported by the Natural Science Foundation of Shanghai (grant no. 22ZR1435400) and the Shanghai Municipal Science and Technology Major Project (grant no. 2021SHZDZX0102). CCF Huawei Populus Grove Fund (grant no. CCF-HuaweiSY202202).
Authors' addresses: S. Zheng and L. Huang, Shanghai Jiao Tong University, 800 Dongchuan Rd., Shanghai, China; emails: {shengan, lphuang}@sjtu.edu.cn; M. Hoseinzadeh and S. Swanson, University of California, San Diego, 9500 Gilman Drive, La Jolla, San Diego, USA, emails: {mhoseinzadeh, swanson}@cs.ucsd.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1553-3077/2023/03-ART20 $15.00
https://doi.org/10.1145/3580280

based on PM. These file systems leverage the direct access (DAX) feature of persistent memory to bypass the page cache layer and provide user applications with direct access to file data.

The high performance of persistent memory comes at a high monetary cost. The average price per byte of persistent memory is higher than that of a solid-state drive (SSD), and SSDs and hard drives scale to much larger capacities than PM. Thus, workloads that are cost-sensitive or require larger capacities than PM can provide would benefit from a storage system that can leverage the strengths of both technologies: PM for speed and disks for capacity.

Tiering is a solution to this dilemma. Tiered file systems manage a hierarchy of heterogeneous storage devices and place data in the storage device that is a good match for the data's performance requirements and the application's future access patterns.

Adopting PM into the storage system poses new challenges to the data placement policy of tiered file systems. Existing tiered storage systems (such as FlashStore [11] and Nitro [31]) are based on disks (SSDs or hard-disk drives [HDDs]) that provide the same block-based interface, and while SSDs are faster than hard disks, both achieve better performance with larger, sequential writes and neither can approach the read or write latency of dynamic random access memory (DRAM). Strata [29] is a cross-media file system that spans PM, SSD, and HDD layers. However, file data can be allocated in the PM layer only, and Strata is inefficient at dealing with operations with poor locality, such as appends.

Emerging persistent memories are byte-addressable and offer read and write latency within a small factor of DRAMs. This makes the decision of where to place data and metadata more complex: The system must decide where to initially place write data (DRAM or PM), and how to divide PM between metadata, newly written data, and data that the application is likely to read.

The first challenge is how to fully exploit the high bandwidth and low latency of PM and DRAM. PM provides a much more efficient way to persist data than disk-based storage systems. File systems can persist synchronous writes simply by writing them to PM, which not only bypasses the page cache layer but also removes the high latency of disk accesses from the critical path. Nevertheless, a DRAM page cache still has higher throughput and lower latency than PM, which makes it competitive to handle read requests and perform asynchronous writes to the disk tiers. These writes can be flushed to disks asynchronously and are guaranteed to be durable after fsync.

The second challenge is how to reconcile PM's random access performance with the sequential accesses that disks and SSDs favor. In a tiered file system with PM and disks, bandwidth and latency are no longer the only differences between different storage tiers. Compared with disks, the gap between sequential and random performance of PM is much smaller [23, 45], which makes it capable of absorbing random writes. At the same time, the file system should leverage PM to maximize the sequentiality of writes and reads to and from disks.

We propose TPFS, a high-performance tiered file system that spans PM and disks. TPFS exploits the advantages of PM through intelligent data placement during foreground file input/output (I/O) and data migration. TPFS includes three placement predictors that analyze the file I/O sequences and predict whether the incoming writes are both large and stable, whether the updates to the file are likely to be synchronous, and whether the file is read frequently. TPFS then steers the incoming I/O requests to the most suitable tier based on the prediction: writes to synchronously updated files go to the PM tier to minimize the synchronization overhead. Small random writes also go to the PM tier to fully avoid random writes to disk. The remaining large sequential writes to asynchronously updated files go to disk. Frequently read files are migrated to disk and cached in DRAM to accelerate reads.

We implement an efficient background migration mechanism in TPFS to make room in PM for incoming file writes and accelerate reads to frequently accessed data. We first profile the temperature of file data (the hotness of data pages) and select the write-coldest and read-hottest file data

blocks to migrate. During migration, TPFS coalesces adjacent data blocks and migrates them in large chunks to disk. TPFS also adjusts the migration policy according to the application access patterns.

The contributions of this article are as follows.

- We describe three predictors to predict the synchronicity, write size, and read frequency of files for different access patterns.
- We describe a migration mechanism that utilizes the characteristics of different storage devices to perform efficient migrations.
- We design an adaptive migration policy that can fit different access patterns of user applications.
- We implement and evaluate TPFS on a real DC persistent memory module (DCPMM) platform to demonstrate the effectiveness of the predictors and the migration mechanism.

We evaluate TPFS using a collection of micro- and macrobenchmarks. We find that TPFS is able to obtain near-PM performance on many workloads even with little PM. With a small amount of PM and a large SSD, TPFS achieves up to 7.3× and 7.9× throughput improvement compared with EXT4 and XFS running on an SSD alone, respectively. As the amount of PM grows, TPFS's performance improves until it nearly matches the performance of a PM-only file system.

The remainder of the article is organized as follows. Section 2 describes a variety of storage technologies and the NOVA file system. Section 3 presents a design overview of the TPFS file system. We discuss the placement policy and the migration mechanism of TPFS in Section 4 and Section 5, respectively. Section 6 evaluates TPFS, and Section 7 shows some related work. We present our conclusions in Section 8.

2 BACKGROUND

TPFS combines emerging PM technologies and conventional block-based storage devices (e.g., SSDs or HDDs). This section provides background on PM and disks as well as the NOVA file system that TPFS is based on.

2.1 Storage Technologies

Emerging pPM provides byte-addressability, persistence, and direct access via the CPU's memory controller. Battery-backed non-volatile dual in-line memory modules (NVDIMMs) [32, 33] have been available for some time. In the event of a power failure or system crash, an onboard controller safely transfers data stored in DRAM to a non-volatile media with the power from rechargeable batteries, thereby preserving data that would otherwise be lost. Battery-free PM technologies include phase change memory (PCM) [30, 34], memristors [42, 46], and spin-torque transfer RAM (STT-RAM) [5, 27]. Intel's Optane DC persistent memory (Optane DCPMM) [22] is the first commercially available persistent memory DIMM. Compute express link (CXL) [10] technology further expands the memory hierarchy to better manage heterogeneous memory devices such as DRAM and PM. All of these technologies offer both longer latency and higher density than DRAM, providing fast, non-volatile, and byte-addressable access to in-memory data. Optane DCPMM has also appeared in Optane SSDs [21], enabling SSDs that are much faster than their flash-based counterparts.

PM, SSD and HDD technologies have their unique latency, bandwidth, capacity, and characteristics. Table 1 shows the performance and cost comparison of different storage devices. Tiered storage systems with SSDs and HDDs (such as [28, 47, 48]) have been proposed to bridge the gap between fast and slow block devices. However, the appearance of fast, byte-addressable PM brings even higher performance and direct access features into tiered storage systems. Therefore,

Table 1. Performance Comparison Among Different Storage Media

Technology	Latency		Sequential Bandwidth		$/GB
	Read	Write	Read	Write	
DRAM	0.1 μs	0.1 μs	25 GB/s	25 GB/s	7.19
PM	0.3 μs	0.1 μs	40 GB/s	15 GB/s	3.37
Optane SSD	10 μs	10 μs	2.5 GB/s	2 GB/s	0.47
NVMe SSD	120 μs	30 μs	2 GB/s	500 MB/s	0.26
SATA SSD	80 μs	85 μs	500 MB/s	500 MB/s	0.20
Hard Disk	10 ms	10 ms	100 MB/s	100 MB/s	0.02

DRAM, PM, and hard disk numbers are estimated based on [2, 23, 26, 49]. SSD numbers are extracted from Intel's website. The monetary costs are extracted from Google Shopping (May 2022).

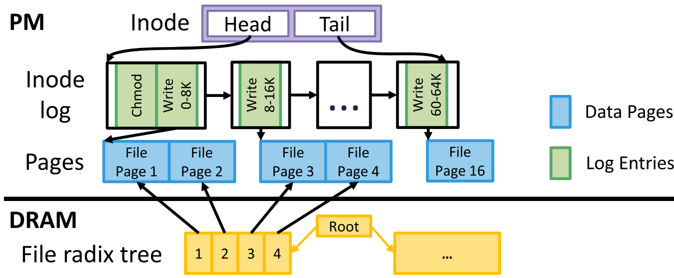


Fig. 1. The file structure of NOVA. A NOVA inode contains pointers to the head and tail of its file log.

it requires a fundamentally different approach from existing tiered storage systems to fully utilize these unique characteristics of PM.

2.2 The NOVA File System

TPFS is implemented based on NOVA [43], a PM-based kernel-space file system designed to maximize performance on hybrid memory systems while providing strong consistency guarantees. Specifically, TPFS inherits the design of log-structured file layout and scalable memory allocators from NOVA. Next, we discuss the file structure and scalability aspects of NOVA's design that are most relevant to TPFS.

NOVA maintains a separate log for each inode to improve multi-core scalability. It also maintains radix trees in DRAM that map file offsets to PM locations. The relationship between the inode, its log, and its data pages is illustrated in Figure 1. For file writes, NOVA creates *write entries* (the log entries for data updates) in the inode log. Each write entry holds a pointer to the newly written data pages as well as its modification time (`mtime`). After NOVA creates a write entry, it updates the tail of the inode log in PM, along with the in-DRAM radix tree. The file log is a linked list of 4 KB log pages. Each log page contains log entries and a pointer that points to the next log page. The inode tail pointer always points to the latest committed log entry. During recovery, NOVA scans the persistent file log from head to tail to rebuild the volatile file radix tree in DRAM to accelerate future accesses.

NOVA uses per-CPU allocators for PM space and per-CPU journals for managing complex metadata updates. Each allocator has its own lock to avoid contention from threads running on different CPUs. This enables parallel block allocation and avoids journal contention. In addition, NOVA uses per-CPU inode tables to ensure good scalability.

3 TPFS DESIGN OVERVIEW

TPFS is a tiered file system that spans across PM and disks (hard or solid-state). We design TPFS to fully utilize the strengths of PM and disks and to offer high file performance for a wide range of access patterns.

Three design principles drive the decisions we made in designing TPFS. First, TPFS should be *cost-effective*. It should use disks to effectively expand the capacity of PM rather than using PM to improve the performance of disks as some previous systems [18, 20] have done. Second, TPFS strives to be *frugal* by placing and moving data to avoid wasting scarce resources (e.g., PM and DRAM capacity, DRAM performance, or disk bandwidth). Third, TPFS should be *predictive* by dynamically learning the access patterns of a given workload and adapting its data placement decisions to match.

These principles influence all aspects of TPFS's design. For instance, being cost-effective means, in the common case, that file writes go to PM. However, TPFS will make an exception if it predicts that steering a particular write in PM would not help application performance (e.g., if the write is large and asynchronous). Alternatively, if the writes are small and synchronous (e.g., to a log file), TPFS will send them to PM initially, detect when the log entries have "cooled," and then aggregate those many small writes into larger, sequential writes to disk. File reads, on the other hand, usually go directly to where the data reside. However, if DRAM bandwidth is underutilized, TPFS will migrate the read-hot file data from PM to disk and cache it in DRAM to accelerate reads. We made the following design decisions in TPFS to achieve our goals.

Send I/O to the most suitable tier. Although PM is the fastest tier in TPFS, file writes and reads should not always go to PM. PM is best suited for small access (since small I/O to disk is slow) and synchronous writes (since PM has higher bandwidth and lower latency). However, for larger asynchronous writes, targeting a disk is faster, since TPFS can buffer the data in DRAM more quickly than it can write to PM, and the write to disk can occur in the background. When it comes to file reads, DRAM offers higher read throughput and scalability than PM. For read-dominated files, storing them in disks and caching them in DRAM provides better performance than reading them directly from PM.

Accurate file temperature profiling. To make migration effective, TPFS picks the target data of migration based on file temperature. TPFS not only needs to migrate the coldest data blocks in the write-cold files to free up space in PM, it also needs to migrate the hottest data blocks in the read-hot files to the DRAM cache to speed up reads. To accurately identify the write-cold/read-hot files, all files in TPFS are sorted by their average modification/read time for write/read migration (Section 5.2). Within each file, TPFS migrates blocks that are older/newer than average, depending on the migration type.

High PM space utilization. TPFS makes full use of PM space to improve performance. TPFS uses PM to absorb synchronous writes and uses disks to expand file system capacity. TPFS uses a dynamic utilization threshold for PM depending on the read-write pattern of applications; thus, it makes the most of PM to handle file writes efficiently.

Exploit DRAM read bandwidth. TPFS optimizes read performance by exploiting both PM and DRAM bandwidth. TPFS monitors the data flow of reads from the two types of memory, migrates the read-hottest data to disk, and caches it in DRAM if DRAM bandwidth is underutilized. These frequently read data pages can achieve higher read throughput on volatile DRAM. Meanwhile, the non-volatile space on PM is cleared up for future writes. By taking advantage of the hybrid memory bandwidth, TPFS achieves high read throughput.

Migrate file data in groups. In order to maximize the write bandwidth of disks, TPFS performs migration to disks as sequentially as possible. The placement policy ensures that most small

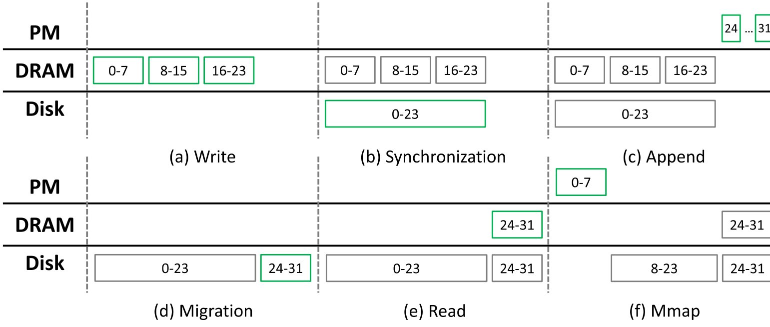


Fig. 2. File operations. TPFS utilizes different storage tiers to handle I/O requests such as write, synchronize, append, migrate, read, and mmap efficiently.

random writes go to PM. However, migrating these small write entries to disks directly will suffer from the poor random access performance of disks. In order to make migration efficient, TPFS coalesces adjacent file data into large chunks for migration to exploit sequential disk bandwidth.

High scalability. TPFS extends NOVA's per-CPU storage space allocators to include all storage tiers. It also uses per-CPU migration and page cache writeback threads to improve the overall scalability of the tiered file system.

Specifically, TPFS uses two mechanisms to implement these design decisions. The first is a placement policy driven by three predictors that measure and analyze past file access behavior to make predictions about future behavior. The second is an efficient migration mechanism that moves data between tiers to optimize PM performance and disk bandwidth. The migration system relies on a simple but effective mechanism to identify write-cold and read-hot data, and moves them accordingly.

We describe TPFS in the context of a simple two-tiered system comprising PM and an SSD, but TPFS can use any block device as the "lower" tier. TPFS can also manage more than one block device tier by migrating data blocks across different tiers. When configured with more than one block device, TPFS divides them into different tiers based on their performance. The faster storage devices reside in higher tiers and vice versa. For the storage devices that have similar or identical performance, they are managed in the same tier, exposing a combined capacity. The file data can be migrated between any two tiers for data caching and eviction during runtime. For block device tiers, TPFS migrates data blocks through DRAM. When a tier is nearly full, TPFS migrates the coldest file data to the lower tier with group migration (described in Section 5.5).

3.1 File Operations

Figure 2 illustrates how TPFS handles operations on files (write, synchronize, append, migrate, read, and mmap) that span multiple tiers.

Write. The application initializes the first 24 blocks of the file with three sequential writes in (a). TPFS first checks the results from the synchronicity predictor and the write size predictor (Section 4) to decide which tier should receive the new data. In the example, the three writes are large and TPFS predicts that the accesses are asynchronous; thus, TPFS steers these writes to disk. It writes the data to the page cache in DRAM and then asynchronously writes them to disk.

Synchronize. The application calls `fsync` in (b). TPFS traverses the write log entries of the file, and writes back the dirty data pages in the DRAM page cache. The write-back threads merge all adjacent dirty data pages to perform large sequential writes to disk. If the file data were in PM, `fsync` would be a no-op.

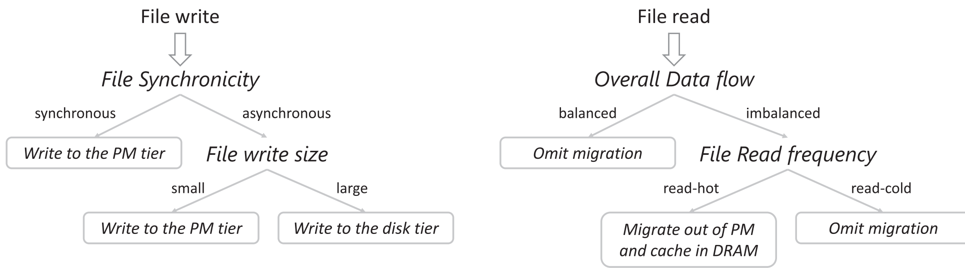


Fig. 3. The placement and migration policy of TPFS. Different scenarios of file writes and reads are classified with decision trees in TPFS.

Append. After the `fsync`, the application performs eight synchronous writes to add eight blocks to the end of the file in (c). The placement predictor recognizes the pattern of small synchronous writes, and TPFS steers these writes to PM.

Migrate. When the file becomes cold in (d), TPFS evicts the first 24 data pages from DRAM and migrates the last eight data blocks from PM to disk using a group migration mechanism (Section 5.5).

Read. The user application reads the last eight data blocks in (e). TPFS fetches them from disk to DRAM page cache. If the frequently read file data is in PM, TPFS migrates them to disk and caches them in DRAM.

Memory map. The user application finally issues a `mmap` request to the head of the file in (f). TPFS uses reverse migration to bring the data into PM and then maps the pages into the application's address space.

4 PLACEMENT POLICY

TPFS adopts different placement policies for file write and read requests from applications. Figure 3 illustrates how TPFS classifies file I/O requests with decision trees. For file writes, TPFS steers synchronous or small writes to PM, but it steers asynchronous, large writes to disk, because writing to the DRAM page cache is quicker than writing to PM, and TPFS can write to disk in the background. It employs two write predictors to distinguish these two types of writes. For file reads, TPFS caches the most frequently read files in DRAM to maximize read throughput. To identify the read-dominated files, TPFS uses a read frequency predictor.

Synchronicity predictor. The synchronicity predictor estimates whether the application is likely to call `fsync` on the file in the near future. The synchronicity predictor counts the number of data blocks written to the file between two calls to `fsync`. If the number is less than a threshold (e.g., 1024 in our experiments), the predictor classifies it as a synchronously updated file. The predictor treats writes to files opened with `O_SYNC` as synchronous as well. The synchronous writes are steered to PM since PM provides in-place persistency rather than going through the DRAM page cache layer.

Write size predictor. The write size predictor not only ensures that a write is large enough to fully utilize disk bandwidth but also that the future writes within the same address range are also likely to be large. The second condition is critical. For example, if the application initializes a file with large I/Os and then performs many small I/Os, these small new write entries will read and invalidate discrete blocks, increasing fragmentation and leading to excessive random disk accesses to service future reads. When the on-disk data blocks are read again, they are loaded into the DRAM page cache. Currently, TPFS utilizes a dedicated page cache by reserving DRAM space. The cached

pages are organized in per-CPU free lists to improve scalability and follow the least recently used (LRU) cache eviction policy.

TPFS's write size predictor keeps a counter in each write entry (the prediction granularity) to indicate whether the write size is both large and stable. When TPFS rewrites an old write entry, it first checks whether the write size is big enough to cover at least half the area taken up by the original log entry. If so, TPFS transfers the counter value of the old write entry to the new one and increases it by one. Otherwise, it resets the counter to zero. If the number is larger than 4 (a tunable parameter), TPFS classifies the write as "large." Writes that are both large and asynchronous go to disk.

Read frequency predictor. The read frequency predictor predicts whether a file will be read frequently. Each file has a counter that keeps track of how many data blocks have been read in a row. The counter is reset when the file is written or appended. If the counter exceeds a certain threshold (e.g., twice the file size in our experiments), the predictor will classify the file as a read-dominated file. In TPFS, only the read-dominated files are subject to DRAM caching. For files with frequent reads and writes, TPFS keeps them in PM. This is because frequent updates lead to frequent DRAM cache invalidation, which, in turn, hurts performance. By caching read-dominated files in DRAM and retaining frequently updated files in PM, TPFS fully utilizes both PM and DRAM read bandwidth to achieve maximum read throughput.

5 MIGRATION MECHANISM

The purpose of migration is to free up space in PM for incoming file writes as well as speed up reads to frequently accessed data. We developed *write migration* and *read migration* approaches to migrate write-cold and read-hot file data to disks. We use a *basic migration* mechanism to migrate data from disk to PM to fully utilize PM space when PM bandwidth is underutilized. We use a *group migration* mechanism to migrate data from PM to disk by coalescing adjacent data blocks to achieve high migration efficiency and free up space for future writes. We implement *reverse migration* to facilitate efficient file data mapping. TPFS can offer near-PM performance for most accesses as long as the migration mechanism is efficient enough.

In this section, we go through the target, timing, and implementation of TPFS's migration mechanism in detail. We first describe how TPFS identifies good targets for migration based on file temperature. Next, we show how TPFS uses two adaptive thresholds to determine whether migration is necessary based on real-time application access pattern analysis. Then, we illustrate how it migrates data efficiently to maximize the bandwidth of the disk with basic migration and group migration mechanisms. Finally, we demonstrate how to migrate file logs in TPFS efficiently.

5.1 Migration Thresholds

In TPFS, migration is triggered by two occasions: (1) excessive file writes resulting in high PM utilization and (2) skewed reads from PM/DRAM leaving DRAM/PM bandwidth underutilized. TPFS adopts two adaptive thresholds, a *utilization threshold*, and a *dataflow threshold*, to monitor the state of each storage tier and adapt to the access patterns of different applications dynamically.

Utilization threshold. Most existing tiered storage systems (such as [4, 29]) use a fixed utilization threshold to decide when to migrate data to lower tiers. However, a high utilization threshold is not suitable for write-dominated workloads since the little amount of free space in PM will be quickly consumed by intensive file writes. In this case, the file writes have to either stall until migration threads have freed up enough space in PM or write to disk, leading to high write latency. On the other hand, a low utilization threshold is not desirable for read-dominated workloads. In this case, reads have to load more blocks from disks instead of PM, underutilizing PM bandwidth.

Therefore, we limit the migration threshold to a range: 90% (maximum threshold) and 50% (minimum threshold) to avoid write blocking and bandwidth underutilization, respectively. We also design a dynamic PM utilization threshold heuristically to adjust the threshold within the threshold range based on the overall read-write data ratio of the file system:

$$threshold = 70\% + 5\% * \log_2 \frac{\#bytes\ of\ reads\ from\ PM}{\#bytes\ of\ writes\ to\ PM}. \quad (1)$$

When PM usage is above the threshold, TPFS initiates file migration from PM to disks. The threshold is increased when there are more reads than writes, enabling PM to absorb more file data cache to improve file read performance. Conversely, the threshold drops when the amount of file writes exceeds reads, allowing for a more aggressive migration strategy to migrate file data more efficiently, preventing write stalling due to insufficient PM. We leave the optimizations for a more accurate utilization threshold algorithm as future work.

Dataflow threshold. The *dataflow threshold* is used to balance the reads from PM and DRAM. We record the amount of data read from PM and DRAM during every second for the last 4 seconds and calculate the dataflow ratio:

$$ratio = \frac{\#bytes\ of\ reads\ from\ PM}{\#bytes\ of\ reads\ from\ DRAM} \quad (2)$$

If *ratio* is greater than 2 in every second of the 4 consecutive seconds, then TPFS deems DRAM read bandwidth underutilized. In this case, TPFS migrates the read-dominated file data with group migration (Section 5.5) from PM to disks and caches them in DRAM to accelerate file reads. On the contrary, if *ratio* is less than 0.5, TPFS deems PM read bandwidth underutilized. In this situation, TPFS migrates file data from disks to PM using reverse migration (Section 5.4).

5.2 File Temperature Profiler

The key to making migration in TPFS effective is identifying the write-coldest and read-hottest file data to migrate in the background. TPFS profiles the temperature of each file by maintaining *temperature lists*, which are the per-CPU lists of files on each storage tier. Each temperature list consists of two sub-lists: the *write-cold list* and the *read-hot list*. The write-cold lists and the read-hot lists are sorted by the average modification time (`amtime`) and average read time (`artime`) computed across all of the blocks in the file, respectively. The per-CPU temperature lists correspond to per-CPU migration threads, which migrate files from one tier to another. TPFS updates the corresponding temperature list whenever it accesses a file. To identify the write-coldest blocks or the read-hottest blocks within a file, TPFS tracks the `mtime` and `rtime` for each block in the file.

To migrate data, TPFS pops the coldest files from a write-cold list or the hottest files from a read-hot list. For the write-cold files during write migration, if the `mtime` of the popped file is not recent (more than 30 seconds ago), then TPFS treats the whole file as write-cold and migrates all of it. Otherwise, the modification time of the file's block will vary, and TPFS migrates the write entries with `mtime` earlier than the `amtime` of the file. As a result, the write-cold portion of the file is migrated to a lower tier, and the write-hot portion of the file stays in the original tier. Similarly, the read-hot portions of the read-hot files are migrated to disks during read migration. After that, the frequently read file data will be cached in DRAM and follow its cache eviction policy.

5.3 Basic Migration

The goal of basic migration is to migrate the write-coldest or read-hottest data in TPFS to disk. When the utilization threshold or the dataflow threshold is reached, a per-CPU migration thread migrates the write-coldest or read-hottest file data to disk. The migration procedure repeats until

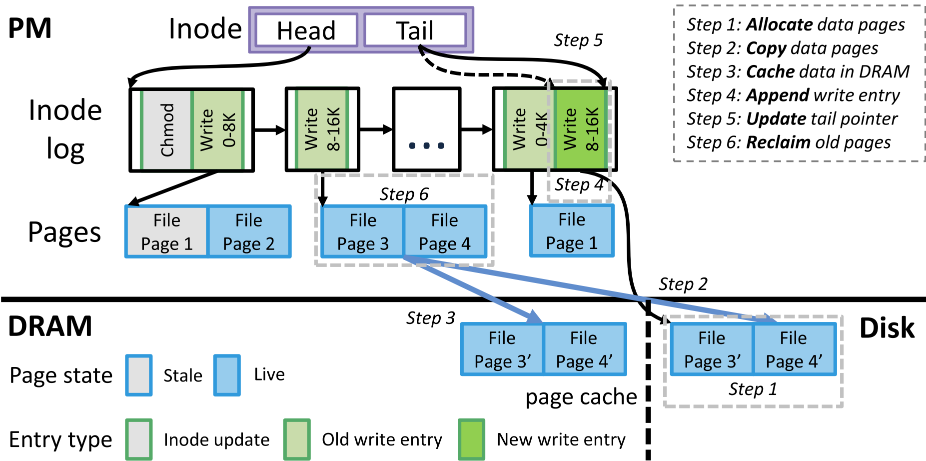


Fig. 4. TPFS's file structure and basic migration. TPFS migrates file data between tiers using its basic migration and group migration mechanisms. The blue arrows indicate data movement, whereas the black arrows indicate pointers.

the usage of the upper tier is below the utilization threshold and the amount of data read from PM is smaller than that from DRAM.

The granularity of migration is a write entry. During migration, we traverse the in-DRAM radix tree to locate every valid write entry in the file and migrate the write entries with `mtime`/`rtime` earlier/later than the `amtime`/`artime` of the file.

Figure 4 illustrates the basic procedures of how TPFS migrates a write entry from PM to disk. The first step is to allocate continuous space on disk to hold the migrated data.

TPFS copies the data from PM to disk and caches it in DRAM. Then, it appends a new write entry to the inode log with the new location of the migrated data blocks. After that, it updates the log tail in PM and the radix tree in DRAM. Finally, TPFS frees the old blocks of PM.

The consistency of file migration is ensured by the atomic update to the log tail pointer of the inode (Step 5). If a crash happens before step 5, the log tail pointer has not been updated yet, indicating that the migration has not been committed. The file data still reside in PM and the persistent data pointers in the inode log are still intact. TPFS can still retrieve the consistent file data from the original logs of the file. If a crash occurs after step 5, the log tail pointer has been updated. The latest file data is organized with the new consistent layout, in which file data has been successfully migrated. Note that although the consistency of file migration is always guaranteed, interrupted migration may result in garbage file data pages. These data pages will be reclaimed by TPFS during garbage collection, when TPFS scans the file logs from the log head pointer to the log tail pointer to reclaim stale data pages and stale log entries.

To improve scalability, TPFS uses locks in the granularity of a write entry instead of an entire file. TPFS locks write entries during migration but other parts of the file remain available for reading. Migration does not block file writes. If any foreground file I/O request tries to acquire the inode lock (e.g., file reads and writes), it will be added to the wait-list of the file's reader-writer lock. The migration thread checks the wait-list periodically. If a foreground I/O request is detected, the migration thread will stop migrating the current file and release the lock.

If a write entry is migrated to a disk during read migration, TPFS will also make a copy of the pages in the DRAM page cache to accelerate future reads. The cache is made by performing `memcpy` from PM, which is used to reduce disk reads in the near future. For write migration, TPFS

will make the copy only when the DRAM page cache usage is low (i.e., below 50%). Writes will benefit from this as well, since unaligned writes have to read the partial blocks from their neighbor write entries to fill the data blocks.

5.4 Reverse Migration

TPFS implements reverse migration, which migrates file data from disks to PM, using basic migration. The procedure of reverse migration is mostly the same as with basic migration (see Section 5.3) except that file data are copied from disk to PM in this scenario. Write entries are migrated successively without grouping since PM can handle sequential and random writes efficiently.

File `mmap` (memory mapping, a function that maps files into memory) uses reverse migration to enable direct access to persistent data. Reverse migration also improves the performance of read-dominated workloads when the dataflow from PM is much lower than that from DRAM. If TPFS can migrate data only from a faster tier to a slower one, then the precious available space of PM will stay idle when running read-dominated workloads. Meanwhile, the data on disks contend for limited DRAM. Reverse migration makes full use of PM in such a scenario.

5.5 Group Migration

The group migration mechanism avoids fine-grain migration to improve efficiency and maximize sequential bandwidth to disks. TPFS tends to fill PM with small writes due to its data placement policy. Migrating them from PM to disk with basic migration is inefficient because it will incur the high random access latency of disks.

The group migration mechanism coalesces small write entries in PM into large sequential ones to disk. There are four advantages: (1) It merges small random writes into large sequential writes, which improves migration efficiency. (2) If the migrated data are read again, loading continuous blocks is much faster than loading scattered blocks around the disk. (3) By merging write entries, the log itself becomes smaller, reducing metadata access overheads. (4) It moderates disk fragmentation caused by log-structured writes by mimicking garbage collection. Fragmented file data blocks are grouped by group migration to form large sequential data segments, whereas obsolete file data are reclaimed by TPFS.

As illustrated in Figure 5, the steps of the group migration mechanism are similar to migrating a write entry. In step 1, we allocate large chunks of data blocks in the lower tier. In step 2, we copy multiple pages to the lower tier of TPFS with a single sequential write and cache the corresponding data pages in the DRAM page cache. After that, we append the log entry, and update the inode log tail, which commits the group migration. The stale pages and logs are freed afterward.

The *group migration granularity* (the granularity of group migration) has a significant impact on migration efficiency. On the one hand, large group migration granularity provides better performance for write-skewed workloads. Since disks and SSDs prefer sequential accesses, TPFS can achieve higher migration throughput with large granularity. As a result, TPFS absorbs more writes in PM, which improves write performance. On the other hand, small group migration granularity reduces the latency of small reads. Since group migration granularity is also the granularity of DRAM caching, small granularity mitigates the read-amplification induced by large granularity caching from disks. Small caching granularity also makes full use of the precious DRAM space. Ideally, the group migration granularity should be set close to the future I/O size, so that applications can fetch file data with one sequential read from the disk. In addition, it should not exceed the CPU cache size in order to maximize the performance of loading the write entries from disks. In this article, we choose 16 MB as the default group migration granularity in TPFS and discuss the effects of different granularity choices in Section 6.6.

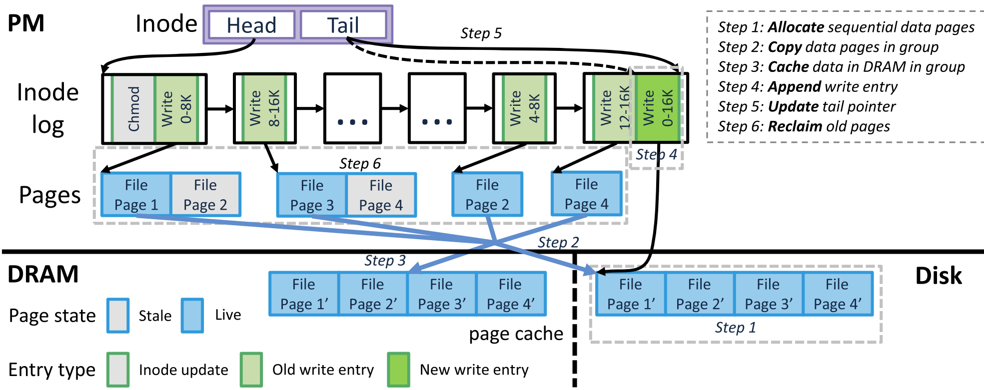


Fig. 5. TPFS's group migration. TPFS migrates file data between tiers using its basic migration and group migration mechanisms. The blue arrows indicate data movement, whereas the black arrows indicate pointers.

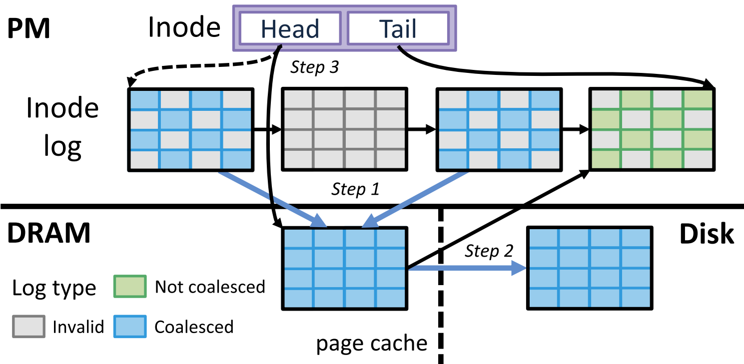


Fig. 6. Log migration in TPFS. TPFS compacts logs as it migrates them from PM to disk.

5.6 File Log Migration

TPFS migrates file logs in addition to data when PM utilization is too high, freeing up space for hot data and metadata. Initially, file logs are allocated and managed in PM since PM is capable of handling small file logs efficiently. However, when PM usage is extremely high, TPFS periodically scans the write-cold lists and initiates log migration on write-cold files. Figure 6 illustrates how log migration is performed. TPFS copies live log entries from PM into the page cache. The log entries are compacted into new log pages during copying. Then, it writes the new log pages back to the disk and updates the inode metadata cache in DRAM to point to the new log. After that, TPFS atomically replaces the old log with the new one and reclaims the old log.

In TPFS, the priority of log block migration is lower than data block migration because of two observations. First, the portion of metadata in TPFS is very small. Each write entry is only 64 B, which contains at least one 4-KB data page. A write entry can contain a massive amount of pages after the log coalescence by group migration. Therefore, migrating file data can free storage space much faster than migrating file metadata. Second, the metadata are usually hotter than data. For instance, reading a data page of a file requires both accessing the corresponding write entry and updating the read time in the inode block, which would be extremely inefficient if they have to be fetched from the disk.

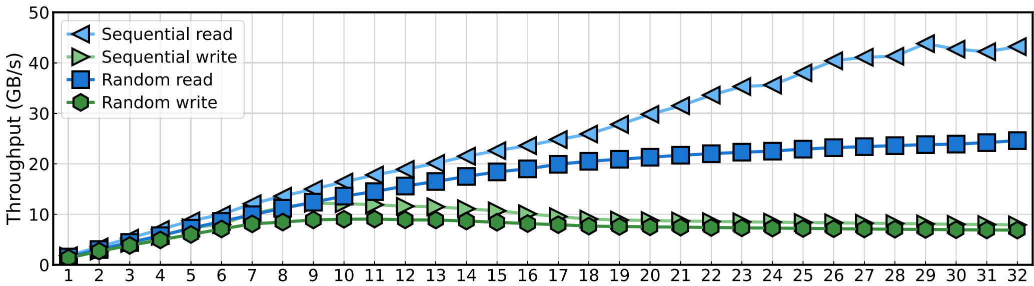


Fig. 7. The sequential and random access bandwidth of the PM on our platform. Optane DCPMM shows high scalability for reads but limited scalability for writes.

6 EVALUATION

In this section, we evaluate the performance of TPFS by comparing it with existing PM-based file systems as well as general-purpose file systems. Our evaluation answers the following questions:

- How do different TPFS configurations affect its I/O performance? (Section 6.2)
- How does TPFS perform against other file systems on multi-threaded workloads? (Section 6.3)
- How does TPFS perform over key-value operations? (Section 6.4)
- How well does TPFS perform when dealing with write-ahead logging? (Section 6.5)
- How do different TPFS parameter choices affect its performance? (Section 6.6)

6.1 Experimental Setup

We implement TPFS on Linux 4.13. We used NOVA as the code base to build TPFS and added around 8.9k lines of code. To evaluate the performance of TPFS, we run microbenchmarks and macrobenchmarks on a dual-socket Intel Xeon server. The server is equipped with two Intel Xeon Gold 6240 CPUs (running at 2.6 GHz with 36 physical cores), 384 GB DDR4 DRAM.

The server has 12 Optane DC Persistent Memory 100 DIMMs (128 GB per module, 1.5 TB in total). To demonstrate the throughput and scalability of the PM on our platform, we perform a raw bandwidth test with Fio [3] to run 4 KB I/O in direct access mode. As shown in Figure 7, the sequential read and write bandwidths are 43.8 GB/s and 12.2 GB/s, respectively. The random write throughput is saturated at around 10 threads.

The server also has an Optane SSD and a SATA SSD, which are used to represent fast and slow SSDs, respectively. The hardware specifications of the SSDs are given in Table 2. The Optane SSD has about an order of magnitude higher bandwidth than the SATA SSD. During the experiments, all of the applications are pinned to run on the processors, DRAM, and PM of NUMA node 0. Both Optane and SATA SSDs are also attached to the same socket (NUMA node 0).

We compare TPFS with a variety of file systems. For PM-based file systems, we compare TPFS with NOVA [43], Strata [29] (PM only), and the DAX-enabled file systems on Linux: EXT4-DAX and XFS-DAX. For general-purpose file systems, we compare TPFS with EXT4 in the data journaling mode (-DJ) and XFS in the metadata logging mode (-ML). Both EXT4-DJ and XFS-ML provide data atomicity, like TPFS. For EXT4-DJ, the journals are stored in a 2-GB journaling block device (JBD) on PM. For XFS-ML, the metadata logging device is 2 GB of PM. We also compare TPFS with EXT4 and XFS with DM-WriteCache [16, 35]. DM-WriteCache is a writeback cache layer that improves performance by caching writes from the page cache to the storage media. We use PM as the hosting device for the writeback cache and set its size to 2 GB for a fair comparison with other file systems. We limit the capacity of the DRAM page cache to 16 GB.

Table 2. SSD Specifications

	Optane SSD	SATA SSD
Product name	Intel Optane DC P4800 SSD	Intel DC S4600 SSD
Interface	NVMe	SATA
Capacity	275 GB	240 GB
Sequential read bandwidth	2400 MB/s	500 MB/s
Sequential write bandwidth	2000 MB/s	260 MB/s

The specifications of the Optane SSD and SATA SSD in our platform. Bandwidth numbers are extracted from Intel's website.

Table 3. Zipf Parameters

Locality	90/10	80/20	70/30	60/40	50/50
Parameter θ	1.04	0.88	0.71	0.44	0

We vary the Zipf parameter, θ, to control the locality of the access stream.

For tiered file systems, we do the comparison among TPFS with different configurations only. To the best of our knowledge, Strata is the only currently available tiered file system that spans across PM and disks. However, the publicly available version of Strata supports only a few applications and has trouble running multi-threaded applications. When running workloads with dataset size larger than the PM size, Strata failed to perform digestion from PM to SSD on our platform. As a result, we run Strata in the PM-only mode.

We vary the PM capacity available to TPFS to demonstrate how performance changes with different storage configurations. The variation starts with TPFS-2 (i.e., TPFS with 2 GB of PM). In this case, the majority of the data is stored on disk, forcing TPFS to frequently migrate data to accommodate incoming writes. TPFS-2 is also an interesting comparison point for EXT4-DJ and XFS-ML since those configurations take different approaches to use a small amount of PM to improve file system performance. The variation ends with TPFS-128/256 (i.e., TPFS with 128/256 GB of PM). The group migration granularity of TPFS is set to 16 MB. We run each workload five times and report the average throughput as well as the 95% confidence intervals (the error bars in the figures) across these runs.

6.2 Microbenchmarks

We demonstrate the relationship between access locality and the read/write throughput of TPFS with Fio [3]. Fio can issue random read/write requests according to Zipfian distribution. We vary the Zipf parameter θ to adjust the locality of random accesses. We present the results with localities ranging from 90/10 (90% of accesses go to 10% of data) to 50/50 (50% of accesses go to 50% of data, i.e., uniform distribution) according to Table 3. The files are initialized with 2 MB writes and the total dataset is 32 GB. We use 32 threads for the experiments, each thread performing 4 KB random I/Os to a private file. For the random write workloads, all writes are synchronous.

Figures 8 and 9 show the results for TPFS, EXT4-DJ, XFS-ML, EXT4-DM, and XFS-DM on the Optane SSD and SATA SSD, as well as NOVA, Strata, EXT4-DAX, and XFS-DAX on PM. The gaps between the throughputs from the Optane SSD and SATA SSD in both graphs are large because the Optane SSD's read/write bandwidth is much higher than the SATA SSD's. The throughputs of TPFS-128 and TPFS-256 are both close to NOVA for the 50/50 locality, while the throughput of TPFS-2 is higher than that of EXT4-DJ and XFS-ML.

As illustrated in Figure 8, the random read performance of TPFS improves with increased locality since higher locality brings a higher probability that the target data blocks are located in memory.

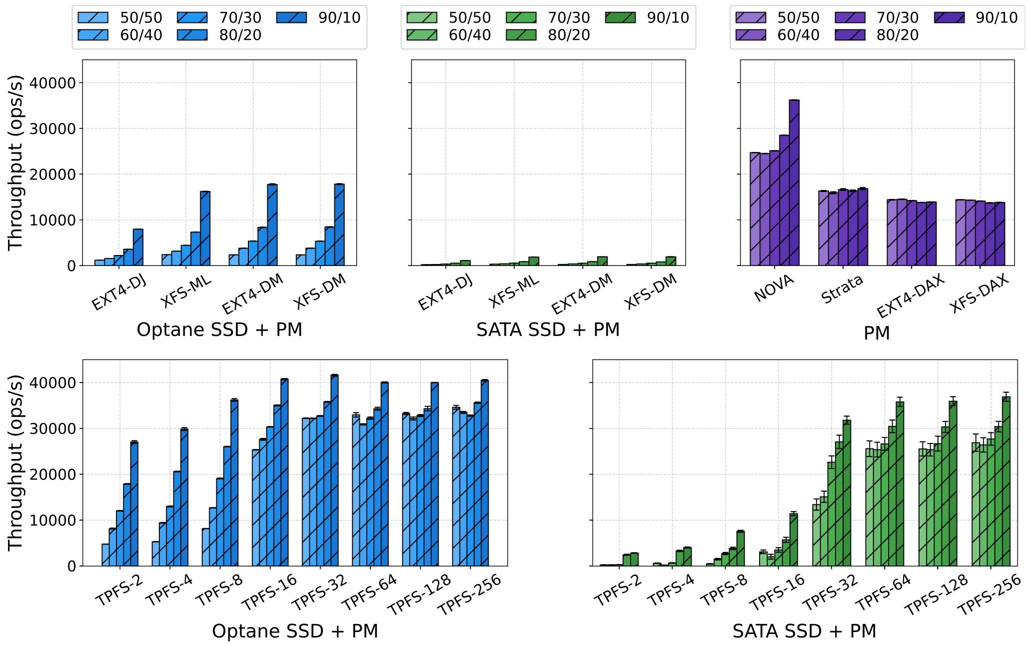


Fig. 8. Fio read performance. TPFS achieves high read throughput for high-locality read workloads due to its effective read migration mechanism and efficient DRAM caching policy.

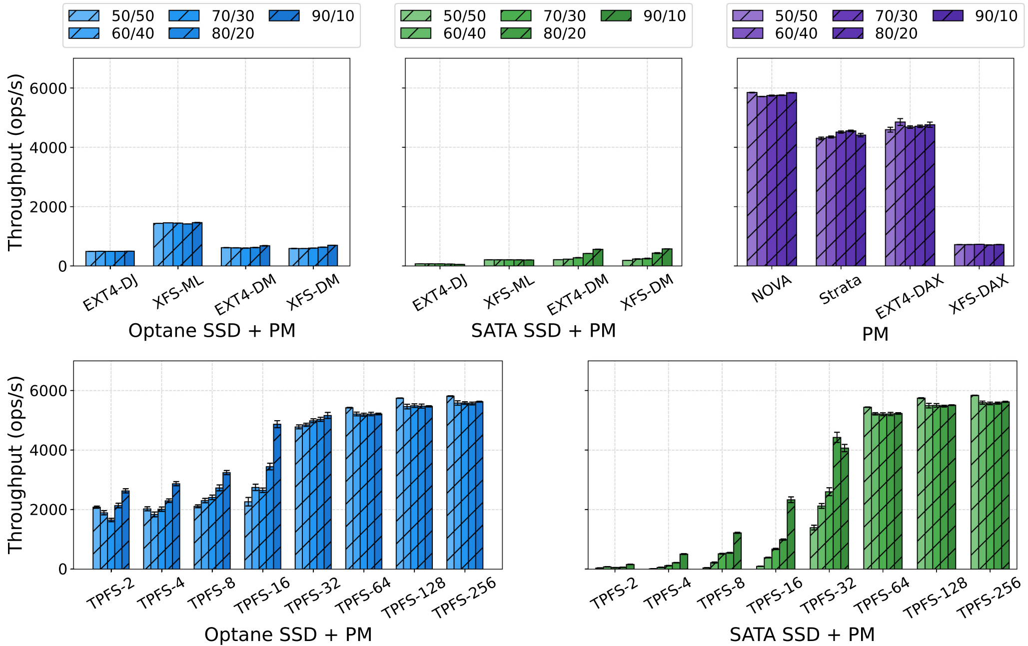


Fig. 9. Fio write performance. TPFS steers most of the synchronous writes to PM and migrates them to disk in the background to achieve high write throughput.

The read performance of TPFS-256 with the Optane SSD and SATA SSD outperforms NOVA by 29% and 6% on average, respectively. This is due to the fact that TPFS makes full use of DRAM's high read bandwidth by identifying read-hot files, migrating them to disk, and caching them in DRAM. For TPFS(Optane), TPFS-16 and TPFS-32 achieve even higher throughput than TPFS-256. This is because part of the file data has already been migrated to the disk tier during the warmup phase due to the limited PM capacity. DRAM caching effectively improves overall read performance. The majority of read overhead comes from fetching cold data blocks from disk to DRAM page cache. In contrast, existing PM-aware file systems use PM only to store the read-frequent data, leaving DRAM bandwidth underutilized. As shown in Figure 8, there is a dramatic performance increase in 90/10 of all the file systems due to CPU caching and the high locality of the workload.

For file writes, as shown in Figure 9, the difference between the random write performance of TPFS with different amounts of locality is small. Since all of the writes are synchronous 4-KB aligned writes, TPFS steers these writes to PM. If PM is full, TPFS writes the new data blocks to the DRAM page cache and then flushes them to the SSD synchronously. Since the access pattern is random, the migration threads cannot easily merge the discrete data blocks to perform group migration in large sequential writes to the SSD. As a result, the migration efficiency is limited by the random write bandwidth of SSDs, which leads to accumulated cold data blocks in PM. As we can see in Figure 9, the performance of TPFS-2 has almost degraded to the throughput of the Optane SSD and SATA SSD. Increasing PM size, increasing locality, or reducing work set size can all help alleviate this problem. DM-writecache helps EXT4 and XFS to achieve higher write throughput on the SATA SSD, especially for highly skewed accesses. For both DM-writecache and TPFS, absorbing data writes in PM effectively alleviates the long write latency caused by disk I/O. When PM can hold all of the written data without the need to migrate to the SSD, the performance gap between TPFS-128 and NOVA is within 1%. TPFS-128 outperforms Strata, EXT4-DAX, and XFS-DAX by 30%, 21%, and 7.9×, respectively.

To evaluate the sensitivity of the ratio of available DRAM to PM of TPFS, we run the Fio read and write workloads again on TPFS with a fix-sized (128 GB) PM and a variable size DRAM page cache. We vary the page cache size from 64 GB to 2 GB to show the performance of TPFS with respect to different DRAM/PM capacity ratios. As shown in Figure 10, for file reads, TPFS achieves high performance when there is ample DRAM and PM space. Large DRAM capacity enables TPFS to leverage the high read bandwidth of DRAM to achieve a high read throughput that exceeds PM bandwidth. The read throughput drops when DRAM cache size is below 16 GB. Since the workload size is 32 GB, TPFS is unable to maintain the balance between the reads from DRAM and PM when the DRAM size is too small. Therefore, TPFS makes full use of the small DRAM page cache to improve the read throughput. As the DRAM size shrinks, the throughput of random (50/50) reads becomes close to the PM read bandwidth (Figure 7). For file writes, TPFS manages to maintain high throughput regardless of DRAM page cache sizes. TPFS handles the synchronous small writes by steering them directly to PM and migrating them to SSDs in the background. DRAM page cache is not involved in the process of small synchronous writes. Hence, write performance remains high and matches the performance of a PM-only file system.

As illustrated in Figures 8, 9, and 10, when having a large PM, TPFS maintains high throughput regardless of different lower tier storage devices. For file reads, as long as file data are either cached in DRAM or stored in PM, TPFS does not have to perform disk I/O to fetch data from slow block devices, achieving memory-level performance. As illustrated in Figure 10, additional DRAM space improves read performance, which makes TPFS reaches even higher read throughput than PM-only file systems through read migration. For file writes, a large PM space enables TPFS to absorb foreground write requests. With the help of group migration, TPFS batches the small random writes in PM, and asynchronously migrates them to SSDs, hiding disk I/O from foreground

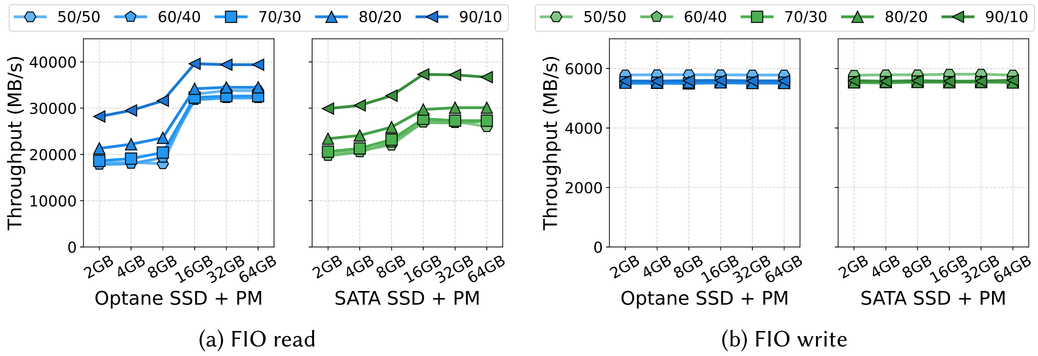


Fig. 10. Sensitivity analysis of TPFS. TPFS achieves high I/O throughput with large DRAM and PM capacity.

Table 4. Filebench Workload Characteristics

Workload	Average file size	Number of files	I/O size (R/W)	R/W ratio
Fileserver	2 MB	30 K ~ 50 K	16 KB/16 KB	1:2
Webserver	2 MB	30 K ~ 50 K	1 MB/16 KB	10:1
Varmail	2 MB	30 K ~ 50 K	1 MB/16 KB	1:1

These Filebench workloads have different read/write ratios and access patterns.

applications. The random-access workloads (such as random read/write workloads of Fio) tend to require larger PM to obtain memory-level throughput, since it is difficult for TPFS to group data pages and migrate them to the SSD sequentially. However, the sequential-access workloads (such as the WAL-based insertion of RocksDB in Section 6.5) need only a small amount of PM to absorb the foreground writes and achieve memory-level performance.

6.3 Macrobenchmarks

We select three Filebench [36] workloads to evaluate the overall performance of TPFS: fileserver, webserver, and varmail. We vary the number of threads from 1 to 32 to demonstrate the scalability of TPFS as well as other file systems. The dataset size of each workload represents 25% utilization of the total storage capacity, ranging from 60 GB (25% of SATA SSD's capacity) to 100 GB (25% of PM and the Optane SSD's capacity). Table 4 summarizes the characteristics of these workloads.

The Filebench throughputs on our five comparison file systems and several TPFS configurations are shown in Figures 11, 12, and 13. In general, we observe that the throughput of TPFS-128 is close to NOVA. The performance gap between TPFS-128 and NOVA is within 5% due to the negligible bookkeeping overhead of TPFS. TPFS gradually bridges the gap between disk-based file systems and PM-based file systems by increasing the PM size. TPFS also shows good scalability by extending the original NOVA design with a scalable migration and page cache mechanism. As shown in the figures, TPFS reaches maximum throughput when using 8 to 16 threads. This is due to the fact that the I/O performance of Optane PM does not scale well with thread count [23, 37, 45].

The fileserver workload emulates the I/O activity of a simple file server, which consists of creates, deletes, appends, reads, and writes. In the fileserver workload, TPFS-2 has similar throughput to EXT4-DJ and XFS-ML. The throughput from the Optane SSD is 1.7× higher than that of the SATA SSD on average due to the higher write bandwidth of the Optane SSD. The performance improves dramatically when the PM size is larger than 64 GB since most of the data resides in PM. TPFS-128

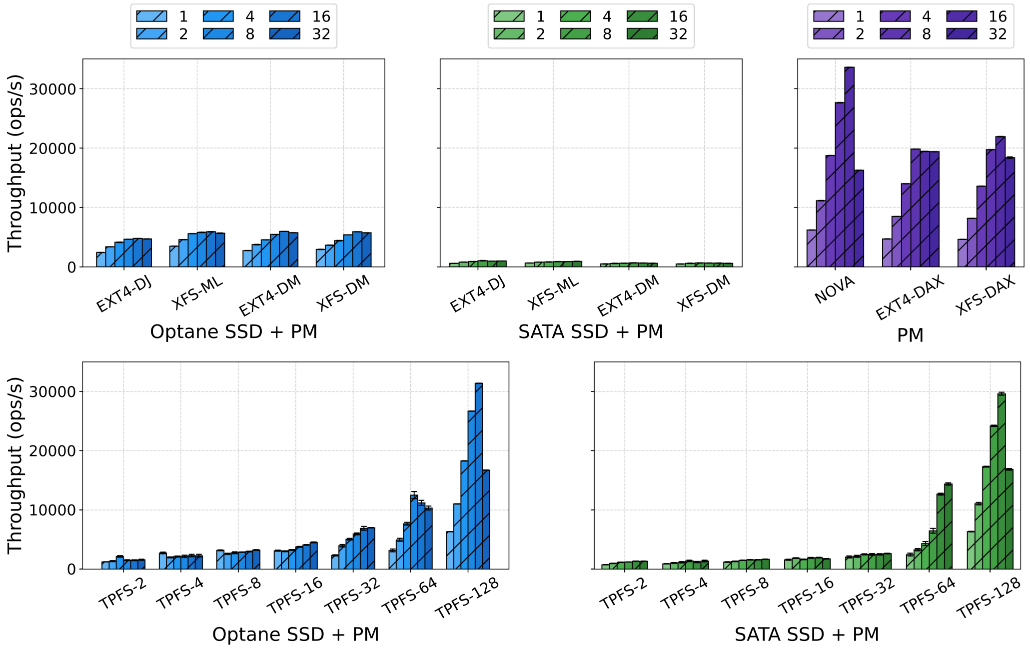


Fig. 11. Fileserver performance. TPFS bridges the performance gap between PM and SSDs.

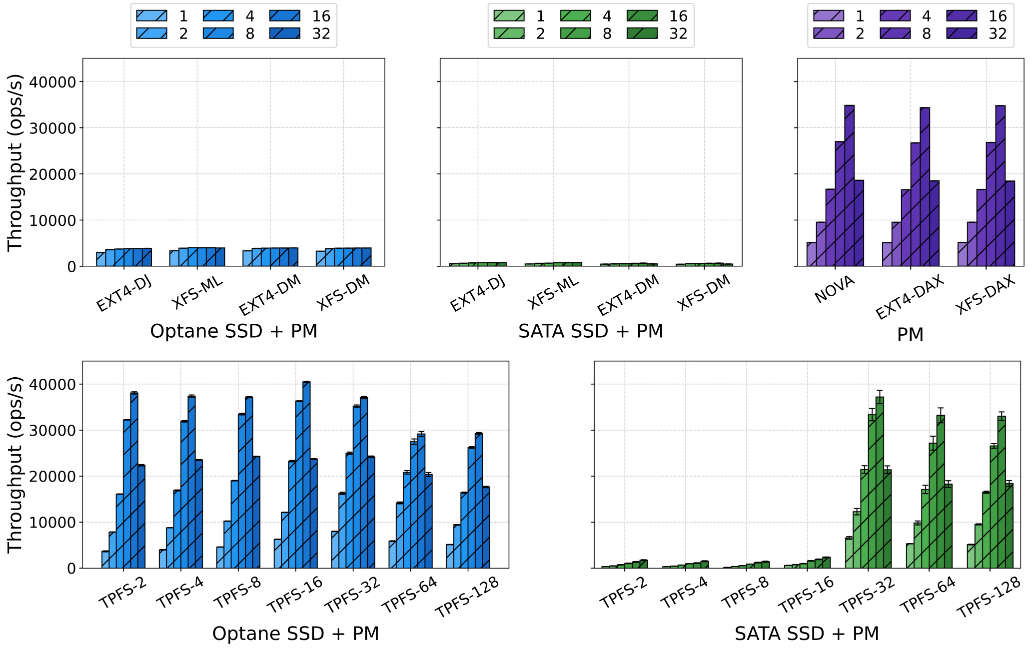


Fig. 12. Webserver performance. TPFS caches the file reads in PM and absorbs the file appends in PM.

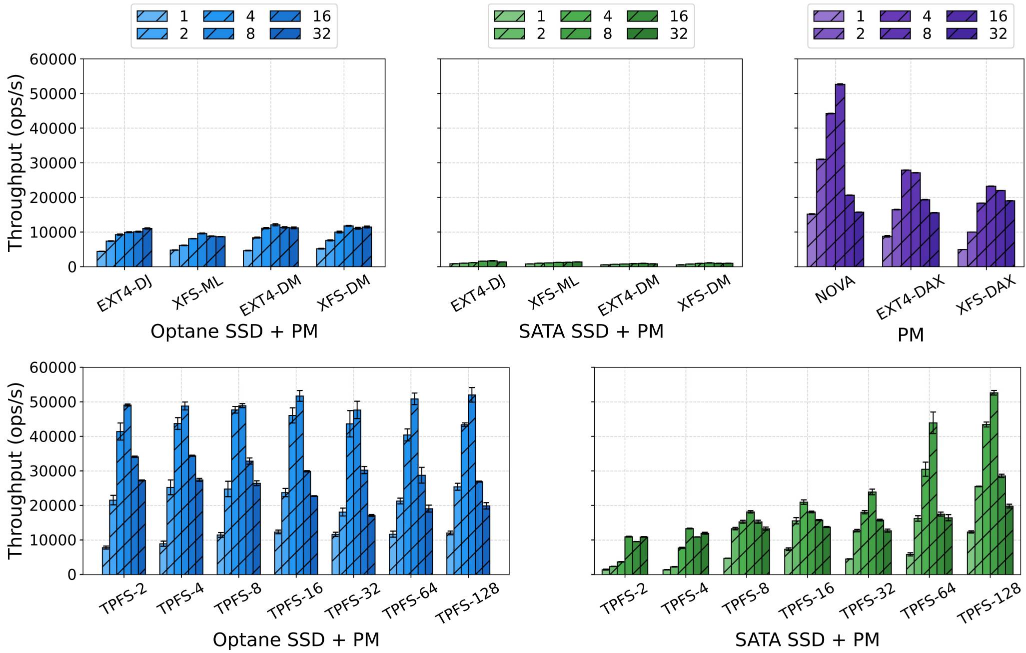
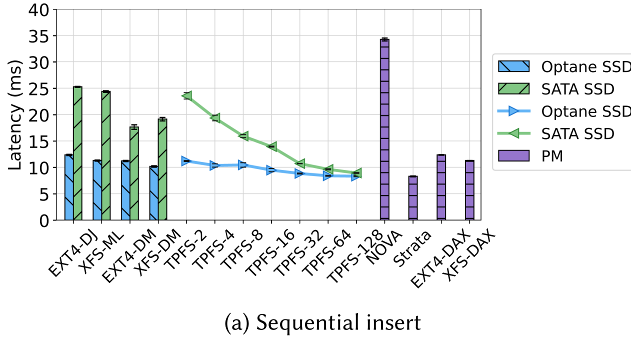


Fig. 13. Varmail performance. TPFS absorbs the synchronous writes in PM and migrates them to the SSD in the background.

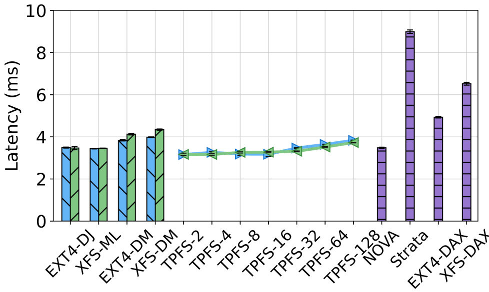
outperforms EXT4-DAX and XFS-DAX by 1.3× and 1.2×, respectively. The throughputs of the file systems are maximized at 16 threads due to the limited scalability of the PM (demonstrated in Figure 7).

Webserver is a read-dominated workload, which involves appends and whole-file reads. Therefore, all of the PM-aware file systems achieve high throughputs with 16 threads due to the high scalability of PM reads. On average, TPFS(Optane) achieves 13%, 15%, and 14% higher throughput than NOVA, EXT4-DAX, and XFS-DAX, respectively. This is because TPFS caches the read-dominated file data in DRAM to achieve high read performance by fully utilizing the DRAM read bandwidth. In the meantime, the file appends are absorbed in PM with low latency. The performance of TPFS(SATA) suffers when the PM capacity is less than 16 GB due to the high access latency of SATA SSD.

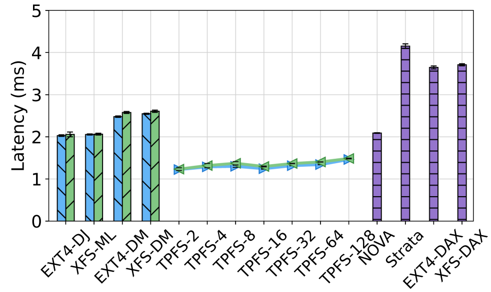
Varmail emulates an email server with frequent synchronous writes. The maximum throughput appeared when running with 8 threads due to PM's limited write scalability. On average, TPFS-2 outperforms EXT4-DJ, XFS-ML, EXT4-DM, and XFS-DM by 3.9×, 4.5×, 5.3×, and 4.8×, respectively. Varmail executes an `fsync` after every two appends. TPFS analyzes these synchronous appends and steers them to PM, eliminating the cost of most of the `fsync` operations. For TPFS(Optane), the write bandwidth of the Optane SSD is capable of handling all background migrations. Therefore, the performance of TPFS-2 is similar to TPFS-128 and NOVA, since all of the synchronous writes are absorbed in PM and migrated to the Optane SSD in the background. The migration of TPFS(SATA), on the other hand, suffers from the low write bandwidth of the SATA SSD. As a result, the performance drops when PM is clogged with foreground writes, exposing the SATA SSD's long write latency to the critical path. Nevertheless, the efficient migration mechanism still enables TPFS(SATA) to outperform EXT4 and XFS by a large margin.



(a) Sequential insert



(b) Random read



(c) Skewed read

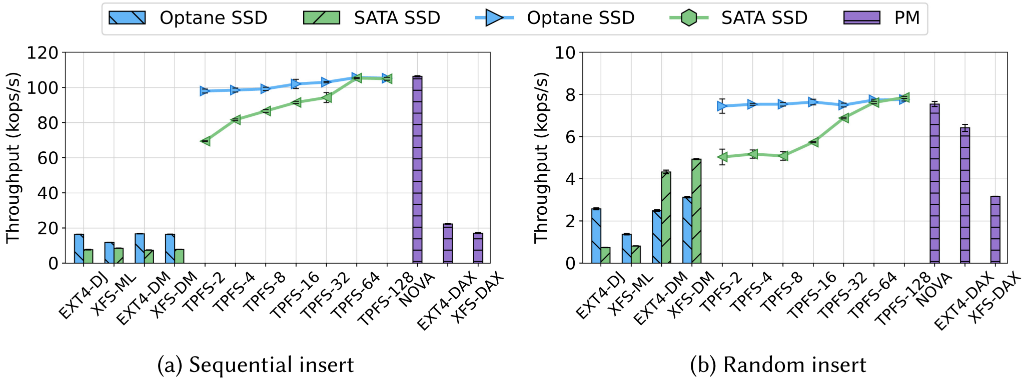
Fig. 14. Average LevelDB benchmark latency. TPFS-2 achieves the lowest latency for random and skewed key-value reads.

6.4 LevelDB

We analyze the access latency of key-value operations with LevelDB [19]. LevelDB is a fast key-value storage library that provides an ordered mapping from string keys to string values. We select three LevelDB workloads from db_bench: sequential insert (FillSeq), random read (ReadRandom), and skewed read (ReadHot) to compare the average latency of typical key-value operations. For each workload, the database size is set to 16 GB. We run each workload in single-threaded mode.

Figure 14(a) measures the average latency of LevelDB's sequential insert operations. Strata achieves the lowest average latency for sequential inserts. Strata's asynchronous log digestion mechanism enables it to merge consecutive insertion logs in the background, reducing the insertion latency. Both TPFS(Optane) and TPFS(SATA) achieve relatively low latency thanks to the group migration strategy, which coalesces adjacent small write requests into large sequential writes to disk. With the increasing PM size, the write latency of TPFS gradually declines, improving the overall throughput. The large sequential asynchronous writes are profiled and steered to the page cache in DRAM to improve write efficiency. Among these file systems, EXT4-DM and XFS-DM use PM to cache key-value inserts effectively to sustain higher throughput. NOVA has the highest insertion latency, which is 3.1× and 1.5× higher than TPFS(Optane)-2 and TPFS(SATA)-2, respectively. This is mainly because NOVA adopts a fixed data block size of 4 KB, which causes write amplification when dealing with small key-value insertions.

Figures 14(b) and 14(c) show the average latency of random and skewed key-value reads of LevelDB. TPFS(Optane)-2 achieves the lowest read latency in both workloads. When dealing with file reads, the read-hot files in TPFS are identified through profilers. These files are then migrated to SSDs and cached in DRAM, which facilitates fast random reads. The read latency further decreases



(a) Sequential insert (b) Random insert

Fig. 15. RocksDB insert performance. TPFS shows good performance for inserting file data with write-ahead logging due to the clear distinction between hot and cold files and its migration mechanism.

for the skewed read workload. This is because the higher read locality of the workload further increases the profilers' accuracy, resulting in decreased read latency due to DRAM caching. On average, TPFS-2 outperforms EXT4-DJ, XFS-ML, EXT4-DM, and XFS-DM by 1.6×, 1.7×, 2×, and 2.1× on skewed read workload, respectively. Disk-based file systems (EXT4-DJ and XFS-ML) achieve similar performance since most of the key-value reads are served directly from the DRAM cache. NOVA also achieves relatively low latency due to the high read performance of PM. Strata, however, performs the worst among all of the file systems for key-value reads. It has 3.4× higher read latency than TPFS-2 on average. Strata utilizes a small DRAM read-ahead buffer to accelerate file reads, which leads to high read latency when the dataset size exceeds the buffer capacity.

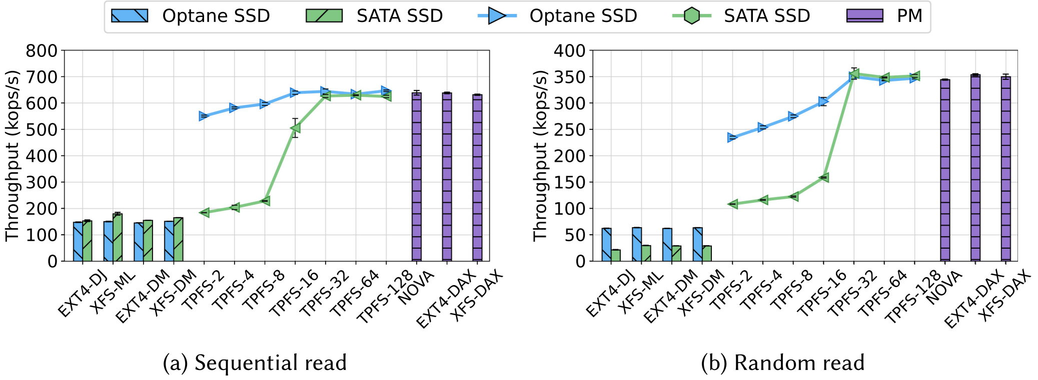
6.5 RocksDB

We use RocksDB [17], a persistent key-value store based on log-structured merge trees (LSM-trees), to demonstrate the high performance and scalability of key-value operations with write-ahead logging (WAL) on TPFS. RocksDB is a fork of LevelDB optimized to exploit CPU cores. Every update to RocksDB is written to two locations: an in-memory data structure called memtable and a write-ahead log (WAL) in the file system. When the size of the memtable reaches a threshold, RocksDB writes it back to the SSD and discards the log.

We select four RocksDB workloads from db_bench: sequential insert (FillSeq), random insert (FillUniqueRandom), sequential read (ReadSeq), and random read (ReadRandom) to evaluate the key-value throughput and migration efficiency of TPFS. We set the writes to synchronous mode for a fair comparison. We run each workload with 8 threads. The database size is set to 32 GB.

Figure 15 measures the throughput of RocksDB's insert operations. When running on PM and the Optane SSD, TPFS is able to stably maintain near-PM performance even when there are only 2 GB of PM for both sequential and random inserts. In the sequential insert workload, TPFS-2 delivers 7.4×, 8.2×, 7.5×, and 7.3× throughput of EXT4-DJ, XFS-ML, EXT4-DM, and XFS-DM on average, respectively. The performance of TPFS-128 is comparable to NOVA's and is 4.7× and 6.1× faster than EXT4-DAX and XFS-DAX, respectively. In the random insert workload, TPFS with 2 GB of PM outperforms EXT4-DJ, XFS-ML, EXT4-DM, and XFS-DM by 4.8×, 5.8×, 2.1×, and 1.7× on average, respectively. DM-writecache significantly improves the random write throughputs of EXT4 and XFS.

The WAL-based insertion of RocksDB is a good fit for TPFS, allowing it to fully unleash its performance advantage. The reason is three-fold. First, since the workload updates WAL files much



(a) Sequential read (b) Random read

Fig. 16. RocksDB read performance. TPFS-2 outperforms disk-based file systems due to the efficient use of both PM and DRAM read bandwidth.

more frequently than the database files, the migration profiler can differentiate them easily. The frequently updated WAL files remain in PM, whereas the rarely updated database files are subject to migration. Second, the database files are usually larger than the group migration granularity. As a result, group migration can make full use of the high sequential bandwidth of SSDs. Moreover, since RocksDB mostly updates the journal files instead of the large database files, the migration threads can merge the data blocks from the database files and perform sequential writes to SSDs without interruption. The high migration efficiency helps clean up PM space more quickly, allowing PM to absorb more synchronous writes, which, in turn, boosts the performance. Third, the WAL files are updated frequently with synchronous and small updates. The synchronicity predictor can accurately identify the synchronous write pattern from the access stream of the WAL files, and the write size predictor can easily discover that the updates to these files are too small to be steered to the SSD. Therefore, TPFS steers the updates to PM so that it can eliminate the double copy overhead caused by synchronous writes to SSDs. Since the entire WAL files are hot, TPFS is able to maintain high performance as long as the size of PM is larger than the total size of the WAL files, which is only 128 MB in our experiments.

For RocksDB's read operations, as shown in Figure 16, TPFS delivers high performance through efficient data migration and DRAM caching. The read throughputs of TPFS-128, NOVA, EXT4-DAX, and XFS-DAX are close (within 3%) since all of the reads are served from either PM or DRAM. However, when the PM size is less than 16 GB, a portion of key-value reads are inevitably served from SSDs due to limited memory capacity. In this case, the high read bandwidth of the Optane SSD helps to maintain a high throughput despite that some SSD reads are involved on the critical path. TPFS(Optane)-2 achieves 85.1% and 67.6% of the throughput of TPFS(Optane)-128 for sequential and random reads, respectively. It also outperforms EXT4-DJ and XFS-ML by 3.8× and 3.7× on average, respectively. Meanwhile, the poor performance of the SATA SSD becomes a bottleneck of the file system, which makes TPFS(SATA)-2 reach only 29.5% and 30.8% of the throughput of TPFS(SATA)-128 for sequential and random reads, respectively. The key to maintaining TPFS's high performance is to avoid direct reads with small I/O sizes to a high latency SSD.

6.6 Parameter Tuning

We illustrate the impact of the parameter choices on performance by measuring the throughputs of workloads from Fio, Filebench, and RocksDB with a range of thresholds. We run the workloads with TPFS-2 on PM and the Optane SSD.

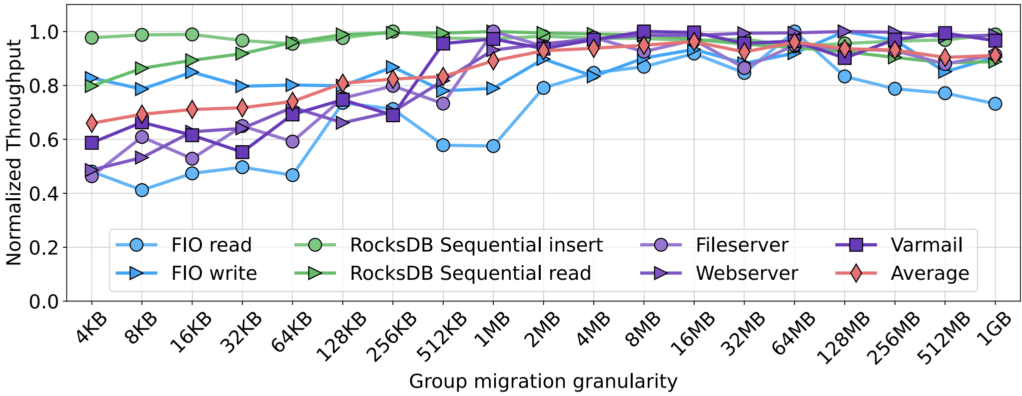


Fig. 17. Performance impact of group migration granularity. We use Fio (blue), RocksDB (green), and Filebench (purple) as our benchmarks. The average throughput (red) peaks when the group migration granularity is set to 16 MB.

Group migration granularity. To demonstrate the performance impact of different group migration granularities, we vary the group migration granularity from 4 KB to 1 GB. The normalized throughputs relative to the maximum performance are shown in Figure 17. In general, larger group migration granularity provides better performance for the majority of the workloads. The average throughputs peak when the group migration granularity is set to 16 MB. This is because the CPU cache size of our experimental platform is 25 MB. Note that the group migration granularity is also the granularity of loading file data from disk to DRAM since TPFS fetches file data in the granularity of write entry to ensure the atomicity and consistency of file reads and writes. On the one hand, if the group migration granularity is too small, TPFS has to issue multiple requests to load the on-disk file data into DRAM, which degrades performance. On the other hand, if the group migration granularity is too large, then a small read request will fetch redundant data blocks from the disk, which will waste I/O bandwidth and pollute the CPU cache.

The throughputs of Filebench workloads are saturated when the group migration granularity reaches 2 MB because the average file size of the workloads is 2 MB. During the migration of the Filebench workloads, the data blocks of a file are coalesced into one write entry, which suits the access pattern of whole-file reads.

When the group migration granularity is 4 KB, the sequential read throughput of RocksDB is approximately only 80% of its maximum performance. This is because large DRAM cache granularity improves sequential read performance due to its high spatial locality. For small group migration granularity, TPFS has to frequently load file data from the disk to serve read requests, leading to high access latency.

Synchronous write size threshold. We vary the synchronous write size threshold from 4 KB to 1 GB. The performance results are insensitive to the synchronous write size threshold throughout the experiments. The standard deviation is less than 2% of the average throughput. We further examine the accuracy of the synchronicity predictor given different synchronous write size thresholds. The accuracy is measured by verifying whether TPFS calls an `fsync` within the synchronous write size threshold according to the prediction result. The predictor accurately predicts the presence or absence of an `fsync` in the near future 99% of the time. The lowest accuracy (98.2%) occurs when the synchronous write size is set between the average file size and the append size of Varmail. In this case, the first `fsync` contains the writes from file initialization and the first append, while the subsequent `fsync`s contain only one append. In general, the synchronous write

size threshold should be set slightly larger than the average I/O size of the synchronous write operations from the workloads. In this case, the synchronicity predictor can not only identify synchronously updated files easily but also effectively distinguish asynchronous, large writes from the rest of the access stream.

Read/write counter thresholds. We vary the read frequency threshold and the sequential write counter threshold of TPFS from 1 to 16. We find that different read/write counter thresholds have little impact on overall performance since the characteristics of our workloads are stable. Users should balance the trade-off between accuracy and prediction overhead when running workloads with unstable access patterns, based on workload profiling. A higher threshold increases the accuracy of the read/write predictor, which can effectively avoid jitter in fluctuating user workloads. However, it also introduces additional prediction overhead for TPFS to deliver reliable prediction results. A lower threshold would benefit the workloads with infrequent pattern changes.

7 RELATED WORK

The introduction of multiple storage technologies provides an opportunity to have a large uniform storage space over a set of different media with varied characteristics. Applications may leverage the diversity of storage choices either directly (e.g., the persistent read cache of RocksDB) or by using PM-based file systems (e.g., NOVA, EXT4-DAX, or XFS-DAX). In this section, we place TPFS's approach to this problem in context relative to other work in this area.

PM-Based File Systems. BPFS [9] is a storage class memory (SCM) file system, which is based on shadow-paging. It proposes short-circuit shadow paging to curtail the overheads of shadow-paging in regular cases. However, some I/O operations that involve a large portion of the file system tree (such as moving directories) still impose significant overheads. Like BPFS, TPFS also exploits fine-grained copy-on-write in all I/O operations.

SCMFS [41] offers simplicity and performance gain by employing the virtual address space to enable continuous file addressing. SCMFS keeps the mapping information of the whole available space in a page table that may be scaled to several gigabytes for large PM. This may result in a significant increase in the number of TLB misses. Although TPFS similarly maps all available storage devices into a unified virtual address space, it also performs migration from PM to block devices and group page allocation, which reduces TLB misses.

PMFS [14] is another PM-based file system that provides atomicity in metadata updates through journaling, but large-size write operations are not atomic because it relies on small size in-place atomic updates. Unlike PMFS, TPFS's update mechanism is always through journaling with fine-grained copy-on-writes.

SoupFS [13] is a simplified soft update implementation of a PM-based file system. The authors adjust the block-oriented directory organization to use hash tables to leverage the byte-addressability of PM. It also gains performance by taking out most of the synchronous flushes from the critical path. TPFS also exploits asynchronous flushes to clear the critical path for higher write throughput.

NOVA [43, 44] is a log-structured file system for PM that provides atomicity through journaling for both data and metadata. It keeps metadata and data information in PM and the indexes reside in DRAM. Each inode has its own log in the form of blocks; thus, the data and metadata updates are committed simply by updating log pointers.

Recent studies have been focused on offloading time-consuming tasks from kernel to user space. ZoFS [12] is a PM-based file system that separates PM protection from management by utilizing coffer, an abstraction to store files with the same permission. User-space libraries can manage files with full control within each coffer. Compared with TPFS, which offers strict data protection through kernels, ZoFS uses kernels to handle cross-coffer protection. ZoFS achieves high performance and flexibility through direct PM management in user space.

SplitFS [25] is another file system that splits the responsibilities between a user-space library and a kernel-space counterpart. In SplitFS, all data operations are handled in user space while all metadata operations are handled in kernel space. SplitFS also introduces a relink primitive to optimize file appends and atomic data operations as well as an optimized logging protocol to provide atomicity and synchronicity to file operations. For file writes, SplitFS has to first append data in a staging file, and then relink it to the actual file by trapping into the kernel, which hurts its I/O performance.

KucoFS [7] provides high scalability by kernel space–user space collaboration. In KucoFS, the time-consuming tasks, such as pathname resolution and concurrent I/O coordination, are offloaded to the user-space library to minimize kernel involvement. File writes are also conducted in a three-phase manner to guarantee consistency while avoiding kernel processing bottlenecks.

Linux has also added support for PM to existing file systems, such as EXT4-DAX [38] and XFS-DAX [8], to allow direct access to PM, bypassing DRAM page cache to improve performance. DM-WriteCache [16, 35] is a writeback caching implementation in Linux that uses PM or the SSD to speed up all writes to the underlying block devices. It only caches writes and relies on a kernel's volatile page cache for read caching.

Tiering Systems. Hierarchical Storage Management (HSM) systems date back decades to when disks and tapes were the only common massive storage technologies. There have been several commercial HSM solutions for block-based storage media such as disk drives. IBM Tivoli Storage Manager [24] is one of the well-established HSM systems that transparently migrates rarely used or sufficiently aged files to low-cost media. EMC DiskXtender is another HSM system with the ability to automatically migrate inactive data from a high-cost tier to a low-cost tier. AutoTiering [47] is another example of a block-based storage management system. It uses a sampling mechanism to estimate the input/output operations per second (IOPS) of running a virtual machine on other tiers. It calculates their performance scores based on the IOPS measurement and the migration costs, and sorts all possible movements accordingly. Once it reaches a threshold, it initiates a live migration.

Since the invention of NVDIMMs, many fine-grained tiering solutions have been introduced. Agarwal and Wenisch propose Thermostat [1], a methodology for managing huge pages in two-tiered memory that transparently migrates cold pages to PM as the slow memory and hot pages to DRAM as the fast memory. The downside of this approach is the performance degradation for those applications with uniform temperature across a large portion of the main memory. Conversely, TPFS's migration granularity is variable; thus, it does not hurt performance due to fixed-size migration as in Thermostat. Instead of huge pages, it coalesces adjacent dirty pages into larger chunks for migration to block devices.

X-Mem [15] is a set of software techniques that relies on an offline profiling mechanism. The X-Mem profiler keeps track of all memory accesses and traces them to find out the best storage match for every data structure. X-Mem requires users to make several modifications to the source code. Additionally, unlike TPFS, the offline profiling run should be launched for each application before the production run.

Strata [29] is a multi-tiered user-space file system that exploits PM as the high-performance tier and SSD/HDD as the lower tier. It utilizes the byte-addressability of PM to coalesce logs and migrate them to lower tiers through log digestion to minimize write amplification. File data can be allocated in PM in Strata only and can be migrated only from a faster tier to a slower one. The profiling granularity of Strata is a page, which increases the bookkeeping overhead and wastes the locality information of file accesses.

SpanDB [6] is an LSM-tree–based KV store that adapts RocksDB to combine the strengths of fast and slow disks. It enables users to host the majority of their data on a cheaper and larger disk

while relocating the WAL and the top levels of the LSM-tree to a smaller and faster disk. Similarly, TPFS uses PM to host file logs by default and stores cold data pages in block devices. However, TPFS employs three placement predictors to analyze the file access patterns accurately in order to adapt to various workloads. To better utilize the fast disk, SpanDB enables fast, parallel accesses via Storage Performance Development Kit (SPDK) to bypass kernel I/O stack and minimize the overhead of WAL writes.

Non-Hierarchical Caching (NHC) [40] improves performance as compared with traditional caching by redirecting excess load to slower devices in the storage hierarchy. Improving cache hit rate is not beneficial when the faster device is already delivering its maximum performance. By offloading requests to the slower devices and dynamically adjusting allocation and access decisions, NHC delivers high performance from both fast and slow storage devices. TPFS leverages a similar insight to enable reads and writes to the block device tier to achieve maximum overall performance. In contrast to the caching approach of NHC, TPFS adopts a tiering approach that migrates file data among tiers, which fully utilizes the durability and performance advantages of PM.

8 CONCLUSION

We have implemented and described TPFS, a tiered file system that spans PM and disks. We manage data placement by accurate and lightweight predictors to steer incoming file reads and writes to the most suitable tier as well as an efficient migration mechanism that utilizes the different characteristics of storage devices to achieve high migration efficiency. TPFS bridges the gap between disk-based storage and PM-based storage, providing high performance and large capacity to applications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback and insightful suggestions.

REFERENCES

[1] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, Xi'an, 631–644.

[2] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* ACM, Melbourne, 707–722.

[3] Jens Axboe. 2012. Fio: Flexible i/o tester. Retrieved January 25, 2023 from http://freecode.com/projects/fio.

[4] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent N. Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. 2017. Curator: Self-managing storage for enterprise clusters. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, USENIX Association, Boston MA, 51–66.

[5] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, and S. Wang. 2010. Advances and future prospects of spin-transfer torque random access memory. *IEEE Transactions on Magnetics* 46, 6 (2010), 1873–1878.

[6] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, USENIX Association, virtual event, 17–32.

[7] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable persistent memory file system with kernel-userspace collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, USENIX Association, virtual event, 81–95.

[8] Dave Chinner. 2015. xfs: DAX support. Retrieved January 25, 2023 from https://lwn.net/Articles/635514/.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles.* ACM, Association for Computing Machinery, Big Sky, MT, 133–146.

[10] CXL Consortium. 2022. Compute Express Link^{TM}: The Breakthrough CPU-to-Device Interconnect. Retrieved January 25, 2023 from https://www.computeexpresslink.org.

[11] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1414–1425.

[12] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, Association for Computing Machinery, Huntsville Ontario, 478–493.

[13] Mingkai Dong and Haibo Chen. 2017. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA*. 719–731.

[14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems*. ACM, Amsterdam, 1–15.

[15] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the 11th European Conference on Computer Systems*. ACM, London, 1–16.

[16] Rémi Dulong, Rafael Pires, Andreia Correia, Valerio Schiavoni, Pedro Ramalhete, Pascal Felber, and Gaël Thomas. 2021. NVCache: A plug-and-play NVMM-based I/O booster for legacy systems. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'21)*. IEEE, Taipei, 186–198.

[17] Facebook. 2012. Rocksdb. (2012). Retrieved January 25, 2023 from http://rocksdb.org.

[18] Ru Fang, Hui-I. Hsiao, Bin He, C. Mohan, and Yun Wang. 2011. High performance database logging using storage class memory. *IEEE 27th International Conference on Data Engineering*, IEEE Computer Society, Hannover, 1221–1231.

[19] Google. 2011. LevelDB. Retrieved February 3, 2023 from https://github.com/google/leveldb.

[20] Dave Hitz, James Lau, and Michael A. Malcolm. 1994. File system design for an NFS file server appliance. In *USENIX Winter*, Vol. 94.

[21] Intel. 2018. Intel Optane Technology. Retrieved January 25, 2023 from https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[22] Intel. 2020. Intel optane DC persistent memory. Retrieved January 25, 2023 from https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[23] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, et al. 2019. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[24] Michael Kaczmarski, Tricia Jiang, and David A. Pease. 2003. Beyond backup toward storage management. *IBM Systems Journal* 42, 2 (2003), 322–337.

[25] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, Association for Computing Machinery, Huntsville, 494–508.

[26] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. 2018. Designing a true direct-access file system with DevFS. In *16th USENIX Conference on File and Storage Technologies*, USENIX Association, Oakland, CA, 241.

[27] Takayuki Kawahara. 2010. Scalable spin-transfer torque ram technology for normally-off computing. *IEEE Design & Test of Computers* 28 (2010), 52–63.

[28] K. R. Krish, Ali Anwar, and Ali R. Butt. 2014. hats: A heterogeneity-aware tiered storage for Hadoop. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'14)*. IEEE, Chicago, IL, 502–511.

[29] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, Association for Computing Machinery, New York, NY, 460–477.

[30] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, Association for Computing Machinery, Austin, TX, 2–13.

[31] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. 2014. Nitro: A capacity-optimized SSD cache for primary storage. In *USENIX Annual Technical Conference*, USENIX Association, Philadelphia, PA, 501–512.

[32] Micron. 2017. Battery-backed NVDIMMs. (2017). Retrieved January 25, 2023 from https://www.micron.com/products/dram-modules/nvdimm/.

[33] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, London, 401–410.

[34] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, Association for Computing, Austin, TX, 24–33.

[35] Rajesh Tadakamadla, Mikulas Patocka, Toshi Kani, and Scott J. Norton. 2019. Accelerating database workloads with DM-writecache and persistent memory. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, Association for Computing Machinery, Mumbai, 255–263.

[36] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking. *USENIX; Login* 41, 1 (2016), 6–12.

[37] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, Athens, 496–508.

[38] Matthew Wilcox. 2014. Add support for NV-DIMMs to ext4. Retrieved January 25, 2023 from https://lwn.net/Articles/613384/.

[39] M. Wilcox. 2017. Add support for NV-DIMMs to ext4. Retrieved February 3, 2023 from https://lwn.net/Articles/613384/.

[40] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. 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)*, USENIX Association, virtual event, 307–323.

[41] Xiaojian Wu and A. L. Reddy. 2011. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Association for Computing Machinery, Denver, CO, 1–11.

[42] Cong Xu, Xiangyu Dong, Norman P. Jouppi, and Yuan Xie. 2011. Design implications of memristor-based RRAM cross-point structures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE'11)*. IEEE, Grenoble, 1–6.

[43] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceeding of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, Santa Clara, CA, 323–338.

[44] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. Nova-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, Association for Computing Machinery, Shanghai, 478–496.

[45] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, USENIX Association, Santa Clara, CA, 169–182.

[46] J. Joshua Yang, Dmitri B. Strukov, and Duncan R. Stewart. 2013. Memristive devices for computing. *Nature Nanotechnology* 8, 1 (2013), 13.

[47] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. 2017. AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter. In *IEEE 36th International Performance Computing and Communications Conference (IPCCC'17)*. IEEE, San Diego, CA, 1–8.

[48] Gong Zhang, Lawrence Chiu, and Ling Liu. 2010. Adaptive data migration in multi-tiered storage based cloud environment. In *IEEE 3rd International Conference on Cloud Computing (CLOUD'10)*. IEEE, Miami, FL, 148–155.

[49] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, Istanbul, 3–18.

Received 7 October 2021; revised 21 June 2022; accepted 15 December 2022