Persistent Memory File Systems: A Survey

Wiebe van Breukelen
Vrije Universiteit Amsterdam

Abstract

Persistent Memory (PM) is non-volatile byte-addressable memory that offers read and write latencies in the order of magnitude smaller than flash storage, such as SSDs. This survey discusses how file systems address the most prominent challenges in the implementation of file systems for Persistent Memory. First, we discuss how the properties of Persistent Memory change file system design. Second, we discuss work that aims to optimize small file I/O and the associated metadata resolution. Third, we address how existing Persistent Memory file systems achieve (meta) data persistence and consistency.

Keywords. Persistent Memory, Storage Class Memory (SCM), Byte-addressable Memory, Memory-Aware File Systems, Intel Optane, Direct Access (DAX)

1 Introduction

Over the past several decades, data storage has become an indispensable part of modern society. However, modern storage had its origins in the early twentieth century. Charles Babbage, who is considered by some to be the "father of the computer", introduced a simplistic form of storage in his Analytical Engine: a general-purpose computer that could be programmed by punch cards [19]. With the emergence of faster and more advanced computers in the 1960s, storage demand grew exponentially. As a result, *magnetic storage*, where data is stored on rotating platters, like on a hard disk drive (HDD), quickly gained traction. Until now, this growth has not slowed down.

As storage demands and processing power increased, a new bottleneck emerged. In demanding environments, such as data centers, data access time could not keep up with CPU speed. This speed gap between CPU and storage continues to grow, so faster storage devices are necessary [28, 27].

A Solid-State Drive (SSD), a form of *flash storage*, offers lower read and write latencies than an HDD, especially in a workload that involves a lot of random data accesses [41]. Like HDDs, SSDs exchange data by the smallest unit of access: a block [81]. Exchanging these blocks between the computer (or host) and storage efficiently is an ongoing challenge. Compared to CPUs, storage devices are an order of magnitude slower in terms of latency [33].

Operating systems strive to minimize the impact of high device latency on application speed. For example, the Linux kernel reduces the performance impact as much as possible by maintaining a *page cache*: a chunk of memory where the OS caches chunks of a file for later use. Based on access patterns, disk blocks can be loaded into memory proactively, allowing substantially lower access latencies [14].

Such mitigations are due to the view we had on storage over the past 50 years. We assumed a two-level storage hierarchy: a fast primary memory (e.g., DRAM) and slow secondary memory (e.g., HDD). Both memories have their own unique properties, for example, the access interface, location within the computer architecture, and access latencies. This has a large influence on the overall design of the Operating System. An alternative scheme, the *one-level storage hierarchy*, changes how we view storage as a whole. Instead of a hierarchy in which we combine the strengths of multiple storage devices, we switch to a hierarchy in which we combine storage and memory into a single device. Persistent Memory (PM) enables the use of such hierarchy [2]. It is a form of storage that is very related to DRAM in terms of access latency, the most significant difference being that PM is non-volatile while DRAM is volatile. A well-known example of Persistent Memory is Intel's Optane Memory [36].

To better illustrate the position of PM in the storage hierarchy, consider Figure 1. PM is located between an SSD and a DRAM module in terms of access latencies and is accessed through CPU load and store instructions at cache line granularity; 64 bytes for the `x86-64` architecture [74]. Note that the capacity and cost scale with the access latencies; storage located at the top (e.g., CPU caches) of the pyramid is scarce and costly compared to storage at the bottom of the pyramid, e.g. HDDs. In terms of data bandwidth, DRAM outperforms PM by quite a margin, see Table 1.

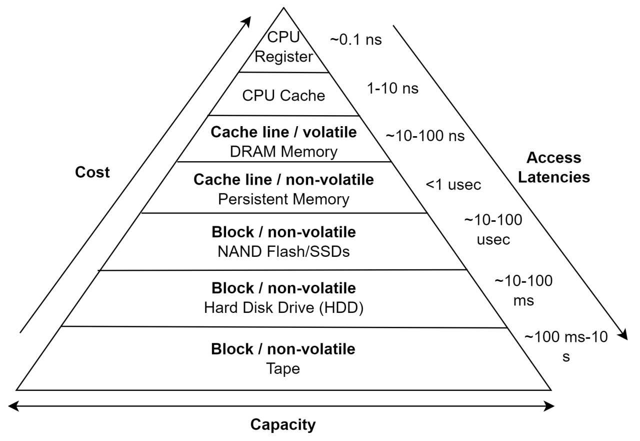
CPU Register
~0.1 ns
CPU Cache
1-10 ns
Cache line / volatile DRAM Memory
~10-100 ns
Cache line / non-volatile Persistent Memory
<1 usec
Block / non-volatile NAND Flash/SSDs
~10-100 usec
Block / non-volatile Hard Disk Drive (HDD)
~10-100 ms
Block / non-volatile Tape
~100 ms-10 s
Cost
Access Latencies
Capacity

Figure 1: Overview of storage devices and their position within the *Modern Triangle of Storage Hierarchy*

	DRAM	**Intel Optane PM**	**% slower**
Read Bandwidth	30 GB/s	7 GB/s	76.6%
Write Bandwidth	20 GB/s	2.2 GB/s	89%

Table 1: Intel Optane 256 GB single DIMM bandwidth (4 threads) `ntstore`-based benchmark performed by Yang et. al [80]

In addition to lower latencies, the byte-addressable property of persistent memory introduces even more opportunities. Byte-addressable storage has been around for decades, for instance, in BIOS chips. Persistent memory introduces this concept to storage, allowing applications to define their own persistent data structures in memory. Another advantage is rapid recovery: a rollback can be performed, even if the application is not aware that the memory is non-volatile. Application-critical data structures are transparently made persistent, so no expensive backup/snapshot run is required [83]. The emergence of PM brings exciting new possibilities but also new challenges:

(1) Traditionally, file blocks are accessed through a system call. Within the kernel, this request is passed to the Virtual File System (VFS), which performs a metadata lookup, resolves the physical location of the file on the disk or page cache, and finally copies the data into a user-allocated buffer.

In the case of slow block devices, the overhead of trapping into the kernel is negligible, as the latency of the device is much higher than the time of the kernel trap. However, this does not apply to PM, as it can perform storage operations in which the latencies are magnitudes smaller than a kernel trap. Consequently, the overhead shifts from the device to the host I/O stack [43].

(2) A closely related challenge is *indexing overhead*. For example, suppose that an application wants to change the permissions for the file `/foo/bar/a.txt`, located on a mounted *ext*4 file system. It performs a `chmod()` call. This call is then forwarded to the kernel's VFS, which in turn performs a so-called *file path walk* by walking the directory tree to locate *file metadata* [9]. Then, a metadata operation is performed; in this case, a file permission change.

The major overhead in this example boils down to this path walk. The total time spent is linear to the number of subdirectories in the file path, in the aforementioned example, two. In the case of slow block storage devices, I/O latency dominates, so this delay is acceptable. On the contrary, PM latencies are an order of magnitude smaller and, therefore, such software overhead is unacceptable [9, 38].

(3) The third challenge is related to the persistence of the data. Modern operating systems use hardware and software caches for performance benefits. Recall that in PM, data is accessed using CPU load and store instructions, like DRAM. In the event of a system crash, this may cause data loss, as the data in the caches may not yet be flushed to disk [4]. A closely related issue is write ordering. If write ordering is enforced, a rollback is trivial, as storage transactions are temporally ordered. However, the Memory Controller, also known as the Memory Management Unit (MMU), can bypass this constraint, as it can violate the order of writing in favor of performance [77].

This survey discusses the changes proposed to solve these challenges. First, we cover the design choices and data structures of existing PM file systems at a high level, i.e., the differences in various file system designs, their position within the OS ecosystem, and their respective strengths and weaknesses.

Second, we discuss work that aims to optimize small file I/O and the associated metadata resolution. Finally, we discuss how PM file systems achieve data crash consistency.

2 Study Design

This section discusses the setup of the literature study. First, we define the survey goal, resulting in one research question and three sub-questions. Subsequently, the scope of the research is defined and, finally, the methodology used for the selection of related literature.

2.1 Research Goal

As mentioned in section 1, the goal of this survey is to investigate the changes required to deal with the three challenges in persistent memory, namely: OS overhead, indexing overhead, and data persistence. Therefore, the main research question is: *Which file system design changes are needed to cope with the challenges that arise when using Persistent Memory?*

In order to answer this question, we have defined three sub-questions. Each sub-question covers one of the three challenges relevant when designing persistent memory file systems:

RQ1. How have the properties/features of Persistent Memory led to changes in file system design?

RQ2. Which optimizations help to decrease file indexing overhead in small I/O workloads?

RQ3. How can Persistent Memory file systems guarantee data crash consistency?

2.2 Scope

In this study, we focus on the challenges in implementing file systems for Persistent Memory. Papers in related fields, such as Persistent Key-Value (KV) stores/databases, are *only* included if they explicitly aim to exploit PM advantages for better FS performance. Note that this assessment is based on the impression of the paper's abstract and conclusion, unless stated otherwise.

We can illustrate this with an example. RocksDB, a persistent KV store, uses the advantages of NOR or NAND Flash SSDs to store associative arrays efficiently. It achieves significant space efficiency and better write throughput while achieving acceptable read performance [22, 24].

Recent work has shown that RocksDB can be adapted to work with PM, although this implementation is very experimental and poorly documented [61]. In such cases, we opt to exclude RocksDB from the literature study.

In conclusion, we limit this survey to the available literature that *specifically* aims to address the challenges, opportunities, or future work when implementing PM-based file systems.

2.3 Methodology

In this survey, we use the *Snowball* sampling [75] methodology to find relevant papers. In this search methodology, one starts by reading the so-called *seed papers*. These papers are then used to find other relevant articles. The seed papers used for this study are listed in Table 2. In order for a paper to be selected, it must adhere to the following inclusion criteria, and none of the exclusion criteria:

- I.1 - The work proposes a new file system, or is a continuation/improvement of an existing file system;
- I.2 - The work especially targets Persistent Memory or related terms: Storage Class Memory (SCM), Phase Change Memory (PCM), or Non-Volatile Main Memory (NVM/NVMM). One or more of these terms must be present in the paper's abstract;
- I.3 - The work is released after 2005. We selected this cut-off date because this year Intel announced PCM modules that could replace traditional flash cards [66];.
- E.1 - The work proposes a new data storage paradigm built on top of an existing file system, for example, a key-value store or hash table. Papers that propose using such paradigms to improve file systems are exempted from this criteria;
- E.2 - The work is classified as a literature study;

Note that in rare cases, we might violate constraints, e.g., when discussing relevant background theory.

In addition to Snowball Sampling, we also performed a manual search. The most relevant keywords are listed in Table 3. We prefer work that is well-cited (i.e, > 20 citations), builds on well-established papers, and is published in the last five years; 2017 - 2022. The names of the conferences considered are: *USENIX (FAST)*, *EuroSys*, *ODSI*, *ASPLOS*, *ACM* (*SIGARCH*, *SIGOPS*, *SPAA*), *IEEE*, *SOSP*, *VLDB*, *SC*, and *ISCA*. In addition, we use *ACM*, *Arxiv*, and *Google Scholar* search engines to find additional work and background theory.

Title	Year
BPFS [18]	2009
SCMFS [77]	2013
Aerie [71]	2014
HiNFS [57]	2016
Strata [39]	2017
NOVA [79]	2017
SplitFS [37]	2019
HashFS [54]	2021
Kuco [15]	2021
FlatFS [9]	2022

Table 2: Seed Papers, shortened titles

Keyword	Accepted	Rejected
Persistent Memory File System	12	6
NVM(M) File System	1	35
Intel Optane File System	3	6
Direct Access/DAX File System	2	2
Persistent Memory Kernel Bypass	0	8

Table 3: Exploratory Keywords and Paper inclusion/exclusion count

3 Background

Before going into the changes required to build fast PM file systems, we provide the relevant background theory. First, subsection 3.1 discusses the relevant hardware concepts depicted in Figure 2. Subsequently, subsection 3.2 shifts the focus to the software components within the kernel, using Figure 4 as a guideline. Finally, subsection 3.3 discusses well-established data structures used in modern file systems.

3.1 Persistent Memory Hardware Architecture

As mentioned in the introduction, persistent memories are byte-addressable. Therefore, a CPU can issue read/write requests using similar microprocessor memory instructions issued when accessing DRAM. Figure 2 displays the position of PM within the computer hardware architecture. When the CPU issues a read request, it requests memory at a particular memory address in DRAM/PM. Now, we have two options: the data is in the cache and can be immediately transferred to the CPU (*cache hit*, faster), or the data should be fetched from DRAM/PM using the *memory controller* (*cache miss*, slower).

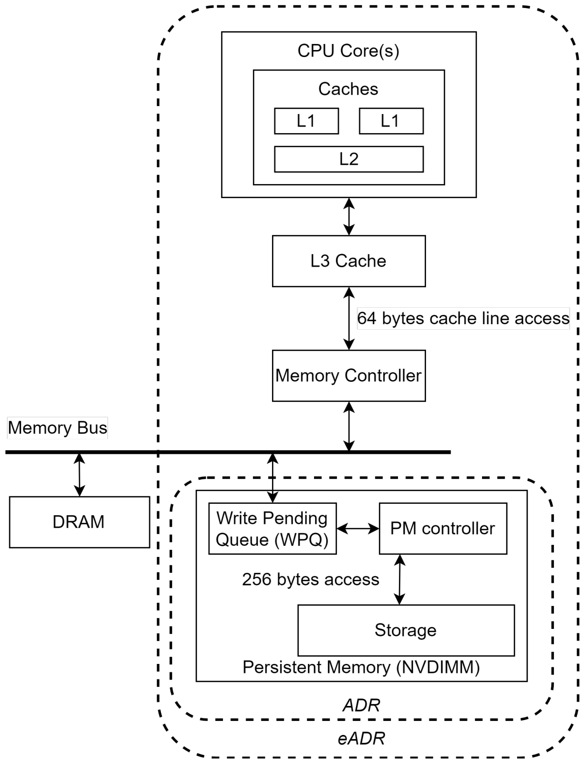

Figure 2: Position of Persistent Memory within the CPU hardware architecture

CPU cache: Modern CPUs use a hierarchical caching architecture consisting of 'levels' to reduce the access latency to main memory [67]. For example, for the `x86-64` architecture, the first cache level (L1) offers lower latencies than a level 2 (L2) or level 3 (L3) cache. On the contrary, an L3 cache offers greater capacity than an L2 cache. For instance, CPUs based on Intel's *Ice Lake* architecture have 48 kB of L1 cache, 512 kB of L2 cache, and 6 MB L3 cache. In terms of latency, an L1 cache line fetch takes approximately 5 CPU cycles, while an L3 cache fetch costs $\sim$ 41 cycles [1] [11].

[1]In case of a cache hit. If requested data do not reside in the cache (i.e., a *cache miss*), the access latency increases as the data must be fetched from main memory [67].

In addition to latency and capacity, three properties are especially important when considering PM [25, 77, 4, 44]. First of all, data resident in the cache may be shared between cores, allowing better multiprocessing performance. Second, in contrast to PM, the cache may not be byte-addressable. In the case of the `x86-64` architecture, the cache stores data in fixed fragments of 64 bytes, also known as a *cache line*. As this is the smallest unit of access, the data is fetched or written in 64 byte chunks.

Third, in shared memory systems, where memory is shared between multiple cores, caches need to confirm to a *cache coherence protocol*. This protocol ensures that each core always has access to the most up-to-date version of a memory location contained in the cache [8]. However, adhering to such a protocol at the price of performance, as it requires additional CPU cycles to perform the necessary consistency checks. To amortize this impact, CPUs may reorder writes [50]. Although this might be beneficial for DRAM performance, it has a big implication for PM which we demonstrate by the example provided in Figure 3.

In this example, the CPU memory controller is instructed to perform four DRAM/PM writes in total. For instance, at $t = 0$ a file system commits the value '56' to address `0x20010000` in PM and the value '201' to DRAM. Note that these requests are not processed immediately; instead, they are buffered at the CPU memory controller. At $t = 2$, the memory controller decides to delay the writes at $t = 0$ in favor of two other writes posted at $t = 1$. Just after processing these two writes, the system crashes. Consequently, both the PM and DRAM writes at $t = 0$ are lost forever. Due to the non-volatile property of PM, the file system ends up in an inconsistent state. The file system (falsely) assumes that file changes are committed to PM in chronological order [44], therefore, it assumes that the value 56 is present at the location `0x20010000`. Instead, this location now contains the value '19' due to the aforementioned write reordering.

Summarizing, write reordering may be beneficial for DRAM reads/writes; however, it poses a threat to PM transactions [77, 71, 43, 82]. First of all, the order data is being written to PM may differ from the user's intentions. Second, in case of a system crash, the data could still be located in the cache and not yet flushed to PM [4]. For now, it is important to understand the implications of CPU caches for PM. The available solutions to fix this issue will be discussed in section 6.

Memory controller: In the case of a *cache miss*, the data must first be obtained from DRAM; therefore, the request is sent to *memory controller*, also known as a Memory Management Unit (MMU). An MMU serves two purposes: translation and isolation [77]. Translation comes down to converting virtual addresses into physical (device) addresses. An MMU provides memory isolation for security and efficient memory fragmentation between applications and the kernel. The internals of an MMU are quite complex; therefore, we refer

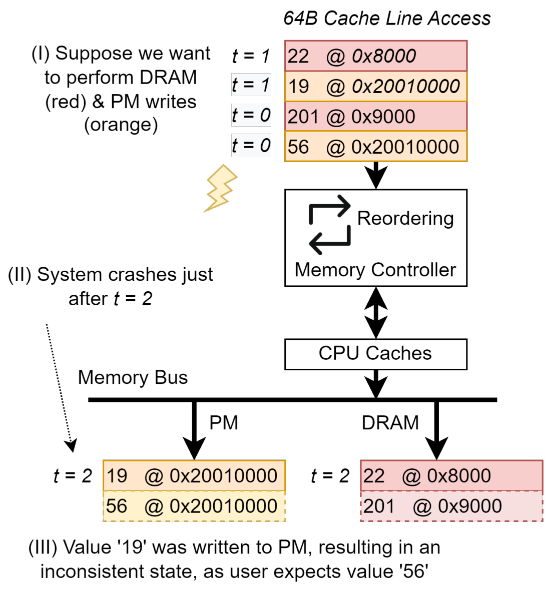

Figure 3: Implications of Write Reordering for Persistent Memory

to the Linux kernel documentation [31], which we quickly summarize.

An MMU uses a *page table* to map virtual addresses to physical addresses. The *x86-64* architecture uses a hierarchical structure in which virtual addresses have a length of 48 bits. The 16 MSB bits cannot be used. This tree structure consists of four levels [2], where each level can accommodate for 512 mappings. At level 3, each mapping covers one 4 kB page, so in total 2 MiB of physical memory as $4\text{kiB} * 1024 = 2\text{MiB}$, that is, the last 12 bits of a virtual address. At Level 2, each mapping covers one 2 MB huge page, i.e., the last 19 bits of the virtual addresses [34]. A key observation here is that when the level decreases, the associated mappings become coarse-grained, therefore, mapping larger regions of memory.

Persistence Domain: Another important concept is the Persistence Domain (PD). As mentioned before, PM is non-volatile memory. Therefore, if a crash occurs, persistence must be preserved, unlike DRAM, where data loss is inevitable. A persistent domain is a region within a computer system where data is guaranteed to be persistent, even in the event of system failure [55]. Figure 2 depicts two persistence domains, namely, Asynchronous DRAM Refresh (ADR) and enhanced ADR (eADR). The first guarantees data persistence within the PM device, while the second extends this guarantee to the CPU cache [62].

We now switch our focus to the internals of the PM device depicted at the bottom right of Figure 2. Device commands end up in a buffer queue, the Writing Pending Queue (WPQ). This queue is part of both the ADR and eADR domains. In case of a crash or power failure, this buffer is emptied and written to storage using the remaining electrical charge in a (super) capacitor.

After passing the WPQ, the writes end up at the PM controller. This controller includes an Address Translation Table (AIT), which maps physical addresses to device addresses and performs wear leveling. Wear leveling prolongs the lifetime of PM by spreading the Program and Erase (P/E) cycles over the memory cells. This is important because flash cells can only endure a finite number of cycles [36].

Eventually, the data is written to PM in fixed-sized chunks, in the case of Intel Optane 256 bytes. As mentioned above, the CPU accesses DRAM via 64 *bytes* load and store instructions. Consequently, writing less than 256 bytes will result in *write amplification*: the difference between the actual amount written and the amount of data intended to be written [59].

We can conclude that PM devices have interesting hardware properties, especially issues related to write reordering and the persistence domain. Both properties are fundamental in evaluating the design of PM file systems.

3.2 Storage Software Stack

In order to utilize storage devices efficiently, software is essential. This subsection discusses software aspects relevant when working with storage devices. More specifically, we discuss the two most prominent forms of accessing storage from user space: conventional system calls and Direct Address (DAX) via the MMU. These techniques can be considered ‘building blocks’ for more advanced PM file system designs which we discuss later.

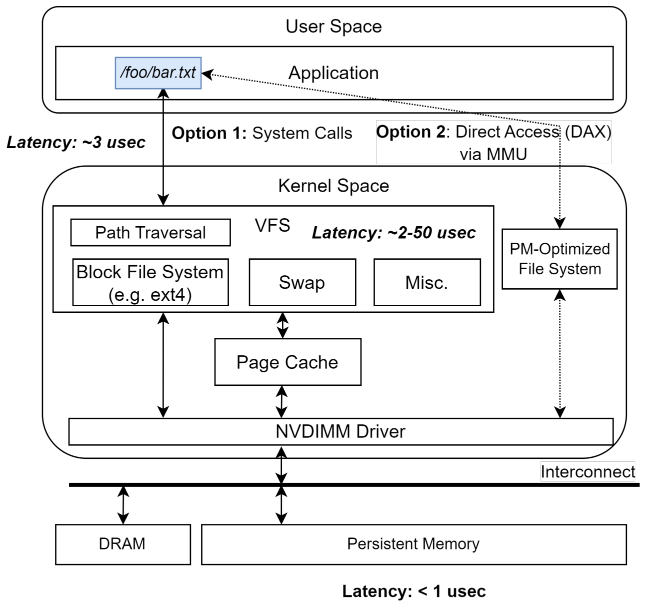

Figure 4: Software Architecture Persistent Memory

[2]Since the Intel Ice Lake microarchitecture (released in 2017) the CPU supports five-level page tables, extending the size of a virtual address to 56 bits, resulting in 128 petabyte addressable space [35].

Figure 4 depicts the position of PM in the software stack. We briefly elaborate on the most important components.

In conventional two-level file systems, applications in *user space* request FS-related services through *system calls*. For example, the `open()` system call can be used to open a file handle, `write()` to write to a file, and `close()` to close a file handle. In kernel space, these FS-related system calls are directed to the Virtual File System (VFS). In order to support multiple file systems, the VFS implements a generic software interface that is independent of the actual file system, e.g.: `ext4`, swap space, or a PM-optimized file system. Internally, the VFS performs a *path traversal*, which recursively traverses the file path, for example, "`/foo/bar.txt`", and informs the corresponding file system of an incoming request. Now, the file system looks up the corresponding *file metadata* and performs the requested action [9].

As mentioned before, the Linux Kernel uses a page cache to decrease the performance impact of slow device latencies. As a result, file data may (partly) reside on the storage device or in the *page cache*:

(1) The file is (partially) located in the page cache. In this case, the requested data can be returned immediately at a very low cost.

(2) The requested file blocks are not in the cache. A request is sent to the corresponding device driver, which in turn sends the actual command to the storage device.

The use of a page cache in the case of PM is controversial, as the induced latency of the VFS is higher than the access latency of PM [43, 77, 18, 15], as seen in Figure 4.

An alternative technique to access file data is Direct Access (DAX). Using DAX, pages are directly mapped in user space using the MMU, avoiding the performance-degrading VFS and page cache [29]. Although this concept is very appealing, it has one (potential) disadvantage: kernel and file system guarantees are lost as one can access storage without kernel interference.

In short, there are two prominent methods to access storage: System Calls and DAX. DAX avoids the use of a page cache.

3.3 Traditional File System Structures

This subsection covers the most prominent data structures found in file systems: metadata structures, B+ trees, LSM trees, and extent/radix trees.

Metadata structures File systems rely on *metadata* to gain insight into files and directories stored within the file system. It is stored on the storage device alongside the actual data blocks. The most important use of metadata is to enable *file mapping*: the operation of mapping a logical file offset to a physical location on the underlying storage device [54]. Generally, file systems map files at block granularity, in most cases 4 *kB*, to constrain the amount of metadata required.

File systems can define any metadata structure to better accommodate their requirements. For example, the *ext2* file system uses *inodes*. An inode is a per-file structure [3] that contains elementary data fields such as file creation modification date, file size, and permissions. Inodes are stored in a *inode table* on disk. To accommodate file mapping for a wide variety of file sizes, *ext2* uses a scalable structure, more specifically an *extent tree*, which we will discuss later. Nodes within this tree represent logical to physical block mappings.

B+ tree A B tree is a self-balancing tree, which means that the node keys are sorted in ascending order, enabling fast sequential performance [16]. Compared to a B tree, a B+ tree only stores values at the bottom of the tree using linked leaves. The internal (non-leaf) nodes only contain keys [13].

LSM Tree A Log-Structured Merge (LSM) Tree is a disk-optimized search tree for storing Key-Value (KV) pairs [56]. Figure 5 depicts a three-level LSM tree. Observe that in this figure, level 0 is an unsorted append-only log located in DRAM. If a level 0 log runs out of space, a compaction routine is started in the background. This compact routine performs Garbage Collection, which in essence iteratively compacts KV pairs at level x into larger, sorted segments located at level $x+1$. Subsequently, the segments are flushed to disk.

An LSM tree avoids Write Amplification (WA) by performing sorting in the background; users can write directly to the in-DRAM log.

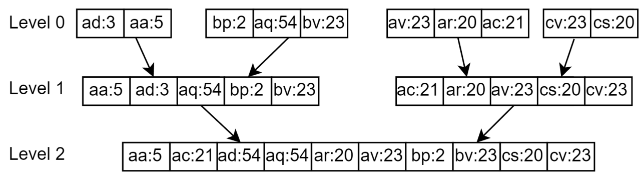

Figure 5: Three-level Log-Structured Merge (LSM) tree

Extent and Radix Tree Both the Extent and Radix tree are B-trees. An *extent* is a data structure that represents a range of contiguous physical blocks, e.g., $11-30$ in Figure 6a. Together, the extents can form a tree that allows efficient *logical block* to physical block number translation. Compared to *file offset*, the offset from the beginning of a file, a logical block number is defined in the OS as a multiple of the device *block size*, usually 512 bytes. The physical block number represents this location on the actual storage device [48]. In Figure 6, the logical block 17 is translated into the physical block 100. Indirect blocks are included to enable file growth and shrinking operations.

[3]In *ext2*, a directory is considered a special type of file.

Radix trees use a different mapping scheme. Instead of using the block number, it uses the corresponding binary representation to perform the lookup. For example, we can map every group of 9 bits to one Radix node, as shown by the dotted arrows in Figure 6b.

An advantage of Extent trees is that they consume less memory than Radix trees, as they grow slower over time. However, Radix trees are computationally less expensive compared to Extent trees as they only require simple arithmetic offset operations for lookups [54].

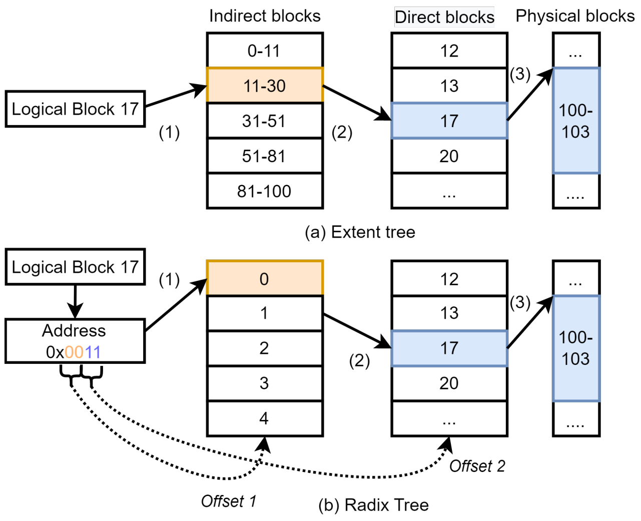

Figure 6: Example of Extent Tree (a) and Radix Tree (b) logical to physical block translation

3.4 Crash-Consistency Techniques

Crash consistency guarantees are essential for file systems. Without these guarantees, a crash may cause data corruption or, in extreme cases, leave the file system inoperable. Some associate data persistence exclusively with the actual data blocks stored on the device; however, data persistence must be enforced in multiple areas [79]. For example, writing to a file involves updating the corresponding data blocks, the last modification date, and the file length. Therefore, in addition to data consistency, metadata consistency is essential.

Generally speaking, modern crash consistency techniques can be categorized into three areas: *Journaling*, *Shadow Paging*, and *Log-Structuring* [25, 18, 79]. We briefly discuss these techniques below.

Journaling A journal is a data structure that keeps track of changes in the file system, separated from the on-disk data blocks. In essence, it is a chronologically ordered log of (meta)data changes, i.e., transactions. A transaction consists of operations that are *idempotent*, which means that they can be repeated infinitely many times without side effects [72].

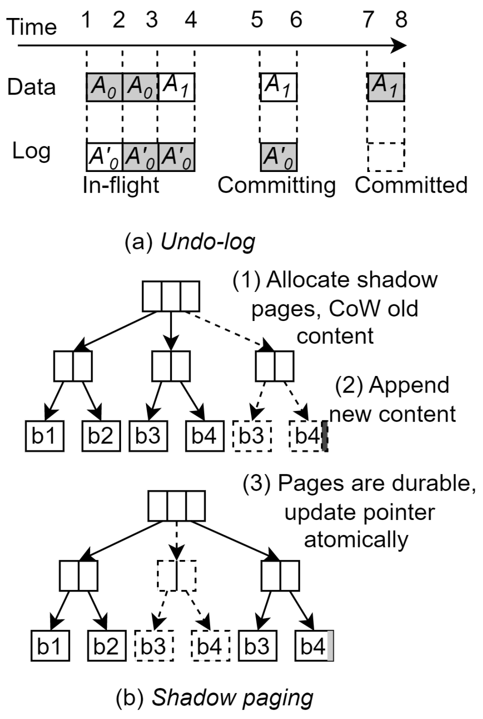

Figure 7: Crash-consistency techniques, (a): *undo-log*, (b): *shadow-paging*

If a crash occurs, the file system can be restored to a consistent state using a Write-Ahead Log (WAL): an append-only disk-resident structure used for crash recovery. In journaling, there are two types of WAL, namely *undo logging* and *redo logging* [72].

In undo logging, a copy of the original data is inserted into the log before a transaction starts. In the event of a rollback, the modifications are restored using the data contained in the log. This process is depicted in Figure 7a. In this example, events are temporally ordered, just as in the journal log. At $t = 1$, the transaction starts by pushing the original data to the log: A'_0. At $t = 2$, this data is in stable storage, indicated by the grey coloring. In-flight data can now be sent to the storage device. When the transaction eventually *commits* ($t = 5$) and the changes are in stable storage, the original data can be erased from the log ($t = 7$).

In redo logging, transactions append data modifications to the log. Only when the transaction commits the corresponding data changes are stored in PM.

Undo/redo logging has its own (dis)advantages. The most crucial difference observed is that redo logging outperforms undo logging in transactions that update a large number of different objects, while it underperforms in workloads with intensive read operations [72]. As performance is heavily affected by the type of workload, there is no consensus on which form of logging is best. However, in the PM literature, there appears to be a slight bias towards 'undo logging', as it is easier to implement [25, 57]. Alternatively, one can also use an *operation log*: a log that only stores file/directory operations, for example, `APPEND data, # num bytes, filename, inode number` [39, 15, 37].

Shadow Paging *Shadow Paging* is a consistency method for file systems based on the *Copy-on-Write* (CoW) technique: a technique that defers resource duplication to the last possible moment, also known as *lazy copying* or *implicit sharing* [18]. Operating Systems use CoW to increase *memory paging* performance. Memory paging is a technique to read/write data in the smallest unit of data storage access (usually 512 bytes), *blocks*, from a storage device for use in main memory [34]. An OS accesses main memory at *page* granularity, usually 4096 bytes. In CoW-based memory paging, a page copy initially refers to the original page (to save resources) and is copied at the last moment, i.e., when a write comes in [7, 18].

File systems that use Shadow Paging use trees to structure metadata and file blocks, see Figure 7b. In the event of a file modification, the original block content is copied to a new page. Then, file modifications are performed on this copied block. When the transaction completes, the changes become persistent by modifying the pointer to the new pointer block. An example is provided in Figure 7b. In this example, a user writes to the block *b*4, which triggers a CoW for the blocks contained in the same *pointer block*. Data modifications are made by modifying the corresponding copied blocks. Eventually, the transaction commits by changing the pointer value in the pointer block located at the top of the tree [10, 25].

Log-Structuring In contrast to Journal-based file systems, which keep track of changes in a separate log on the disk, log-structured file systems store file system metadata and data updates together. This implies that all file system data is structured in the form of a log, also known as *Log-structured File Systems* (LFSs) [1]. Originally, LFSs were designed to improve HDD performance, as HDDs offer poor random performance but high sequential performance [64]. Although SSDs offer improved random performance, they still perform better in sequential workloads [12], therefore, it is still beneficial to use sequential accesses as much as possible.

In an LFS, random writes are buffered in DRAM and merged into large sequential writes to the log. To avoid fragmentation, a periodic *Garbage Collection* (GC) run is performed, in which the free blocks are coalesced to form new contiguous free regions.

In summary, we have seen three file system crash-consistency techniques: *Journaling*, *Shadow Paging*, and *Log-Structuring*. Journal-based file systems maintain a chronologically ordered log of file transactions, stored separately from the on-disk data blocks. Shadow Paging uses the Copy-on-Write technique to ensure data durability. In Log-Structured file systems, the disk becomes one long log, containing all (meta) data transactions.

4 Persistent Memory File System Design

Unlike classic block devices, such as HDDs, which communicate through a relatively slow ACHI controller, PM is placed on the memory bus and accessed via processor load-and-store instructions. This change shifts the main performance bottleneck from device to software.

Traditional file systems like *ext2* perform expensive operations on the critical path, for example, logging file updates, file metadata lookup, maintaining persistency, etc. Furthermore, the hierarchical structure in ext2 introduces a high indexing overhead; in the worst case, up to 45% of a simple 4 kB data append [43, p. 37], and up to 4× write amplification [52].

To some extent, this performance impact is mitigated by using the page cache. However, in the case of PM, its byte-addressable properties allow us to access data faster than the I/O stack, making an expensive page cache redundant. Therefore, an efficient design and accompanying data structures are crucial for low-latency/high-throughput PM file systems.

Considering the design of the file system at a high level, the relevant literature can be categorized into multiple distinct areas, as shown in Table 4. To answer sub-question RQ1, we focus on the challenges each design and its associated file systems intend to solve. Specific optimizations (e.g.: guaranteeing data persistence/atomicity, indexing overhead reduction) will be addressed later in Sections 5 and 6, respectively.

4.1 Influenced by Traditional File Systems

This category of work adapts well-established data structures in file systems, such as the inode tree, to work with PM. Such file systems benefit from upstream fixes/patches in the Linux Kernel, which can then easily be integrated into the PM-optimized file system.

BPFS BPFS, released in 2009, is, to the best of our knowledge, the first file system adapted to work with PM. It maintains an indirect block tree similar to conventional file systems, i.e., *ext*2. Its main contribution is a PM-optimized implementation of Shadow Paging (see subsection 3.4), *Short-Circuit Shadow Paging*, which we now elaborate.

Although Shadow Paging is a well-proven feature for ensuring consistency, the authors of the BPFS paper [18] name a clear disadvantage: *Write Amplification* (WA). In tree-based file systems, when new data is written in CoW-fashion, the pointers in the parent blocks must also be updated [18, 25], propagating tree node updates upwards the tree (see Figure 7b). The resulting WA is significant as the smallest addressable unit of storage is, in most cases, 512 or 4096 bytes [21], while a pointer update is only 8 bytes in the case of a 64-bit system.

High-level Design	File System	User/Kernel Space	DAX	POSIX-compliant	Main Contribution
Influenced by Traditional File Systems (e.g. *ext2*)					
	BPFS [18]	Kernel	✗	✓	POSIX-compliant file system that reduces write amplification through adapted shadow paging
	PMFS [25]	Kernel	✓	✓	Bypass OS page cache and generic block layer, avoid extensive I/O stack modifications. Lightweight in-place metadata updates
	HiNFS [57]	Kernel	✓	✓	Elimination of double copy overhead in kernel
	Ext4-DAX [29]	Kernel	✓	✓	Include DAX to PM in the existing *ext4* file system
Contiguous File Allocation					
	SCMFS [77]	Kernel	✗	✗	Bypass the generic block layer and perform file mapping via the MMU
	SplitFS [37]	Hybrid	✓	✗	Introduces a *hybrid* architecture in which data operations are handled in user space, while metadata operations are processed in the kernel
	Aerie [71]	Hybrid	✗	✗	Allow user space applications to update metadata directly in user space
	Kuco [15]	User	✓	✓	Address the poor scalability of existing PM hybrid file systems (e.g., SplitFS)
	ZoFS [23]	User	✓	✓	Like Aerie, allow user space applications to update metadata directly in user space, however, with less kernel involvement
Log-Structured					
	NOVA [79]	Kernel	✗	✓	Per inode logs to allow massive parallelism, while providing strong consistency guarantees
	Strata [39]	Hybrid	✓	✓	Capture unique properties of multiple storage devices in one file system

Table 4: PM File Systems categorized by their high-level design

BPFS avoids this WA by performing in-place updates within data blocks, taking advantage of PM's unique byte-addressable properties. As mentioned before, BPFS uses a tree structure to store inodes. A significant difference is that BPFS also includes the actual file blocks within this tree, forming one giant tree consisting of inodes at the top and the corresponding file data blocks at the bottom. Now, block changes can be made by performing in-place updates, in-place appends, or partial CoW. In-place updates/appends can be performed in case the data block is located in a tree leaf. Partial CoW is used when multiple (non-leaf) file blocks are affected.

At the time BPFS was released, the availability of PM was very poor. Therefore, the benchmark results should be taken with a grain of salt. Still, compared to Microsoft's NTFS, BFTS achieves significantly higher throughput in small I/O workloads. However, for large I/O workloads (e.g., moving files between directories), this overhead is still substantial [79].

PMFS PMFS is very similar to BPFS in the sense that both implement a POSIX-compliant file system using system calls. Like PMFS, the authors of BPFS acknowledge that file system consistency imposes a major performance penalty. However, compared to BPFS, there are three differences. First, PMFS uses a B tree instead of the conventional indirect block tree used in BPFS for faster file indexing. Second, PMFS enables applications Direct Access (DAX, subsection 3.2) to PM memory via a `mmap` interface, bypassing the expensive OS page cache. Third, PMFS proposes a hybrid approach to handle file (metadata) consistency. Recall that BPFS uses an optimized version of Shadow Paging to ensure consistency. Although this technique certainly decreased WA, the authors of PMFS [25] found that PBFS fails to capitalize on PM's unique byte-addressable property. They observed that metadata updates are usually small (≤ 64 bytes). Therefore, PMFS proposes an alternative methodology that performs writes at cache-line granularity using atomic store instructions [4]. For metadata updates larger than a single cache line, PMFS resorts to more expensive *undo logging* (subsection 3.3). For larger data updates, PMFS falls back to Shadow Paging.

In short, PMFS performs atomic updates when possible, as those are the cheapest operations. Only in cases where in-place updates are not atomic, PMFS resorts to undo logging and Shadow Paging [25].

HiNFS The PM file systems we discussed so far mainly improve performance by adapting conventional file system design to reduce Write Amplification. Although the related optimizations certainly improved performance, the authors of HiNFS suggest that there is still a performance bottleneck: poor write latency due to double-copy overheads in the kernel. To resolve this, HiNFS proposes two optimizations: extensive *latency hiding* behind the critical path and elimination of double copy overhead [57].

HiNFS achieves latency hiding by an *NVMM-aware Write Buffer* policy. Using this policy, file writes are classified as *eager*-persistent or *lasy*-persistent. In the former, the write operation is performed immediately, without latency hiding. In the latter case, HiNFS buffers the request by moving its payload into a fast DRAM buffer. This 4 kB-sized DRAM buffer uses the Least Recently Written (LRW) policy to order writes in temporal order. DRAM blocks are indexed by building a B-tree per file. A per-core kernel thread writes data from the DRAM buffer into PM, after which the DRAM blocks can be reclaimed. Write consistency is maintained by reusing PMFS's hybrid consistency mechanism.

In the case of a read operation, HiNFS first checks if the corresponding blocks are in DRAM. When this is the case, the requested data can be returned immediately. If the data is not in DRAM, it is fetched from PM and directly copied into the user buffer. This avoids a double copy: device $\rightarrow$ kernel $\rightarrow$ user. Although this seems like a simple mechanism, it is more subtle. Suppose that a block is partly in DRAM, which may happen as PM is byte-addressable. In this case, HiNFS consults the *cacheline bitmap*, which tracks the state of each 64 byte cache line in a DRAM block, to determine which areas need to be fetched from PM.

The way HiNFS avoids double-copy overheads for write operations is quite complex; therefore, we provide only a brief explanation. Essentially, HiNFS uses multiple criteria (e.g., DRAM and PM latencies) to decide whether coalescing writes into one large write operation is beneficial. Periodically, these coalesced writes are moved to persistent memory.

Ext4-DAX As we have seen in previous paragraphs, quite a bit of effort has gone into pushing PM support into the existing I/O stack. PMFS and HiNFS allow applications DAX to PM via a `mmap` interface. Instead of making extensive changes to the I/O stack, *ext4-DAX* only adds DAX support to the existing *ext4* file system [29]. We have already discussed DAX in subsection 3.2. In short, applications can access PM without the interference of the kernel via CPU load and store instructions, bypassing the expensive OS page cache and generic block layer. In terms of throughput and latency, ext4-DAX performs similarly to PMFS [25] and worse than HiNFS [57].

In this section, we have seen four PM file systems that (extensively) modify the I/O stack to support PM. All four file systems are POSIX-compliant (see Table 4), so applications can access PM without rigorous code changes. PBFS and PMFS introduced performant techniques to reduce Write Amplification: *Short-Circuit Shadow Paging*, and atomic metadata updates. Additionally, they bypass the expensive OS page cache using DAX to PM. HiNFS addresses the double-copy overhead in the kernel to improve access latency. Ext4-DAX adds DAX support to the existing *ext4* file system.

4.2 Design: Contiguous File Allocation

As mentioned before, file systems for block devices (e.g., *ext4*) support large files using indirect blocks. An alternative is to position the file system and some of its data structures in virtual memory and take advantage of the hardware capabilities, the Memory Management Unit, better known as the MMU. This approach has three advantages. First, keeping track of where a file is located is reduced to an operation involving two numbers: the starting address of the VMA and the offset within the file. Second, the number of seeks is minimized since the entire file can be read in one operation, instead of separate blocks [77]. Third, DAX gives the user much more flexibility to design custom storage structures.

Now, we will discuss the relevant designs proposed in the literature. We start by discussing SCMFS, a file system that inspires multiple modern PM file systems. Subsequently, we discuss file system designs that solve the limitations of SCMFS or propose other novel techniques, as seen in Table 4.

SCMFS To the best of our knowledge, the SCMFS file system [77] was the first to implement a contiguous file system for Storage Class Memory (SCM), nowadays better known as PM. It uses the MMU to map file addresses to physical addresses. Figure 8 displays a high-level view of SCMFS.

The physical space contains the actual file system, its metadata, and a mapping table. This mapping table is used to initialize the MMU by inserting entries that map user space virtual addresses to physical addresses representing locations on the PM device. Modifications to these data structures are always transferred back to the PM device for consistency. Please note that both virtual and physical addresses need to be relative rather than absolute to account for randomization techniques such as *Address Space Layout Randomization*

[4]Intel's atomic store instructions: Section 8.2.4 - Intel Architecture Software Developer Manual [35]

(ASLR) [46], which randomly arranges data in the virtual address space for security purposes.

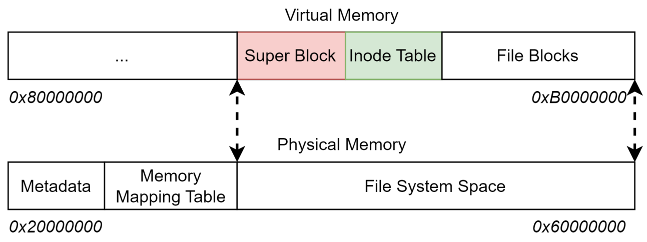

Figure 8: SCMFS physical and virtual memory layout

As depicted in Figure 8, the virtual space consists of three parts: the super block, the inode table, and the actual files. The super block serves the same purpose as in traditional file systems: keeping track of block/inode counts, block size, etc. [30]. The inode table stores file/directory metadata.

Recall that the translation from virtual to physical addresses is done in hardware by the MMU. To speed up translation, the MMU includes a cache, the *Translation Lookaside Buffer* (TLB) [43]. The TLB has a fixed number of address translation entries, usually between 16 and 512. As this cache is quite small, we should use it efficiently to avoid the so-called *TLB misses*. Let us demonstrate this with an example. Consider a 2*MiB* file mapped in virtual memory by 512 conventional 4*kB* pages, which may occupy all TLB entries [5]. Now suppose that we want to map multiple 2*MiB* files in the file system space. In this situation, the number of cache misses increases dramatically, as both files cannot be in the TLB at the same time, increasing the latencies.

SCMFS reduces potential TLB misses for large files by employing *huge pages*. Instead of mapping one 2*MiB* file using conventional 4*kB* pages, it maps one 2*MiB* huge page. This approach also avoids internal fragmentation; the amount of allocated but unused space.

Although SCMFS has shown good speed-ups in both (random) read and write workloads, it still has several limitations:

(1) In SCMFS, non-temporal data persistence and consistency are enforced by the `clflush` and `mfence` instructions, enforcing PM writes are temporally ordered. However, this is done at the expense of exposing the suboptimal write latency of PM devices to the critical path [57, 4]. (2) SCMFS does not avoid the costs of trapping into the kernel frequently: switching CPU protection modes, saving/restoring the trap frame, invoking the scheduler, etc. [63, 3] (3) Li et. al [43] note that SCMFS does not address the challenge of slow file resizing (using appends) and external fragmentation.

We will address the issues related to consistency in section 6. The other limitations are addressed by other PM file systems, which we discuss in a moment.

[5]For simplicity, we ignore that modern operating systems perform extensive software optimizations avoid poor TLB utilization [43, 25]

SplitFS SplitFS [37] resolves two limitations of SCMFS, namely kernel trap overhead and append performance. To accomplish this, SplitFS proposes a *hybrid* design: a file system design in which user and kernel space have distinct responsibilities. A user space library (*U-Split*) services the *data path*, that is, all data operations in virtual memory, e.g.: `read()`, `write()`, or an append. Metadata-related operations, for example, `open()` or `rename()`, are handled by a kernel library (*K-Split*), i.e. the *control path*.

The main performance-degrading aspect of an append operation is data copying in the kernel [37]. To mitigate this issue, the authors of SplitFS propose the use of *staging files*. Instead of appending to the actual file, the operation is redirected to a temporary staging file in PM, managed by U-Split; see Figure 9. Note that in this case file data may be spread over two locations, the stage and actual files. Consequently, U-Split maintains a collection of memory-mapped areas per file for accounting purposes.

Eventually, all staged file modifications must be made persistent. In U-Split, this flushing procedure is initialized after capturing a POSIX `fsync()` using `LD_PRELOAD`. Then, a *relink* procedure is executed. This procedure logically moves the PM blocks from the staging file to the target file by a *zero-copy operation*: an operation that avoids unnecessary data copies [70]. The relink procedure involves several steps, as shown in Figure 9:

(1) In the first step, an append operation is performed. This operation is redirected to a staging file. Note that each staging file is mapped to a physical block, and the auxiliary translation is performed by the MMU, as in SCMFS.

(2) Eventually, `fsync()` is invoked by the user, so the flushing procedure starts. One block of the staging file is decoupled from its corresponding physical block.

(3) & (4) This physical block is relinked with the target file to form a new contiguous file in the VMA.

Note that in the aforementioned steps, data copies are avoided. We modify only the virtual-to-physical page mapping in the page table, which is a relatively inexpensive operation.

In terms of performance, SplitFS achieves good throughput: 27% improvement in sequential reads and 7.85$\times$ speedup in appends. Metadata-heavy workloads still introduce an overhead of up to 13% due to the additional bookkeeping required to manage the staging files.

Aerie Aerie is another file system that implements the hybrid user/kernel space architecture. Compared to SplitFS, Aerie shifts even more responsibility to user space, improving metadata-heavy workloads. Applications can define their own workload-specific file system interfaces in user space. This design allows for higher performance than a single generic interface, e.g., POSIX [37]. Aerie exposes two services in user space: *libFS* and the *Trusted File System Service*, and the *SCM Manager* in kernel space.

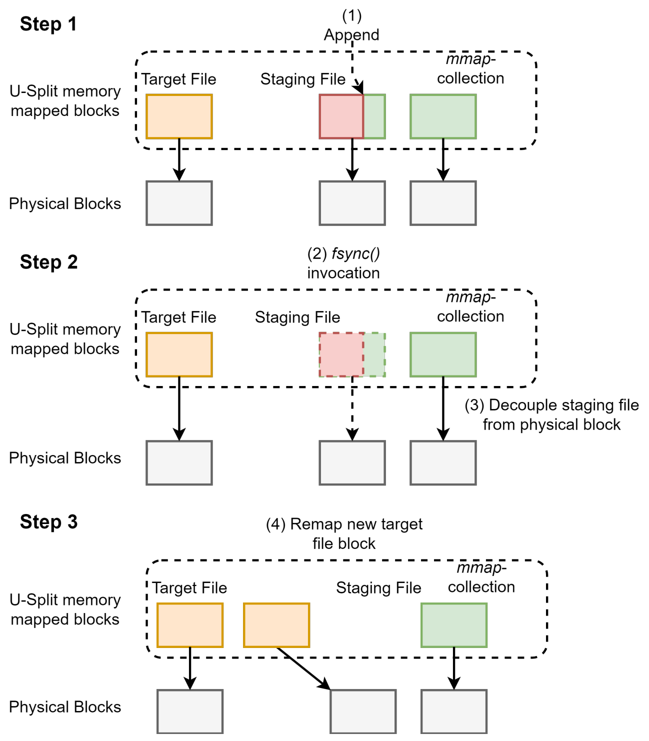

Figure 9: SplitFS U-Split *relink* procedure, partly based on the figure made by Kadekodi et al. [37]

LibFS provides applications with the essential ability to define a file system: the ability to map file names to file metadata, and indexing, which translates a file offset into a byte in memory. The Trusted File System Service (TFS) handles the integrity of metadata updates and concurrency. It runs as a Remote Process Call (RPC) service accessible through a user-mode process. The SCM Manager multiplexes physical PM allocations and maps backing pages into user space. Note that for compatibility purposes, Aerie also defines a POSIX-like interface, PXFS.

The performance of Aerie is evaluated using PXFS. Compared to kernel-mode file systems (i.e., ext4), PXFS achieves 53% $\sim$ 109% higher throughput in single-threaded workloads. Unfortunately, the freedom Aerie provides comes at the cost of poor multi-core scalability. PXFS do not scale linearly beyond four threads due to contention in the TFS's storage allocator.

Kuco The authors of Kuco [23] find that multicore scalability has not been well addressed by other file systems. For example, Aerie relies on a centralized TFS to enforce concurrency control, which becomes a bottleneck in high-concurrency workloads. File systems that avoid the page cache, e.g. SCMFS and SplitFS, still experience software overhead due to kernel traps and the VFS.

Kuco's high-level design is very similar to that of SplitFS, in the sense that it proposes a *client/server* model in which user (*Ulib*) and kernel space (*Kfs*) have their own responsibilities. Kuco's design shifts even more responsibility to the user space to decrease the involvement of the kernel. To do this, Kuco introduces three new techniques: *collaborative indexing*, *two-level locking*, and *versioned read*.

Collaborative indexing: Kuco offloads most of the path name resolution to a user space library, *Ulib*. Applications communicate with a dedicated *Ulib* instance to perform metadata operations, e.g., `chown()`. An Ulib instance looks up file metadata through the so-called *partition trees*. These data structures are quite complex; therefore, we will not elaborate on all the details here. For now, it is sufficient to understand that these complex tree structures maintain all the file/directory inodes contained in a file system partition. When a metadata operation is initiated, *Ulib* performs the required pathname resolution by traversing the corresponding partition tree. It looks up all related metadata items in user space and includes the associated virtual memory pointers in the system call payload. After performing the necessary consistency checks, *Kfs* can perform the metadata operations directly by writing to the corresponding virtual memory. Using this approach, expensive locking is avoided as only *Kfs* can perform metadata updates.

Two-level locking is used to coordinate concurrent file writes. Kuco introduces *direct access range-lock* to serialize fine-grained concurrent writes by performing region locking. Using this lock, multiple threads can write different data pages in the same file simultaneously.

The *versioned read* mechanism allows for user-level reads without any kernel involvement, avoiding an expensive RPC or system call. It ensures that readers never read data that is out-of-date/incomplete by embedding a 'version field' in each data pointer inside the block mapping table.

Combining these techniques, Kuco achieves up to one magnitude higher throughput in small I/O workloads compared to SplitFS, PMFS, and Ext4-DAX. In a 16-thread Filebench [49] benchmark, Kuco outperforms PMFS throughput by 1.2×, and Ext4-DAX by 1.9×.

ZoFS The last hybrid file system that we discuss is ZoFS. Like Aerie, ZoFS gives the user space direct control over both data and metadata, allowing applications to design their own file systems. We already mentioned that Aerie's multicore performance does not scale linearly due to contention inside the user space TFS. To improve parallel performance, ZoFS proposes an alternative implementation in which applications can access metadata without a user-space library.

In Aerie, applications must request permission from the TFS every time it wants to access file data. In ZoFS, an application only requests permission once, avoiding TFS's performance-degrading Remote Procedure Calls (RPCs). This is done by issuing a system call directed to ZoFS's kernel module: *KernFS*. If permission is granted, KernFS assigns the application an *coffer*: a range of PM pages that share the same permission properties [23]. Protection and isolation of a *coffer* are enforced in hardware through Intel's Memory Protection Keys (MPK) [20, 35]. It allows the kernel to restrict the permissions of memory regions mapped in user space. For example, one could map a set of PM pages as read-only in user space. This feature is supplemental to the MMU's page protection bits; both permission checks will be performed during memory access. Once access is granted and associated *coffer* PM mappings are inserted, applications can access the associated memory region without any kernel interference during the application lifetime.

Compared to other PM file systems, ZoFS achieves higher throughput in workloads that affect a fixed set of file blocks, e.g., constantly appending data to the same set of files. This is because the number of context switches into the kernel is significantly reduced. Workloads that involve a dynamic set of files perform worse due to an increase in *coffer* access requests, resulting in a higher number of context switches.

To summarize, a contiguous file system design enables applications to access PM from user space, reducing kernel involvement. These 'hybrid' file systems (SplitFS, Aerie, Kuco, and ZoFS) allow the construction of application-tailored file systems in user space, reducing the role of the kernel, which ultimately results in better parallel performance and throughput compared to PM file systems that extensively modify the existing kernel I/O stack (see subsection 4.1).

4.3 Design: Log-Structured

This category of PM file systems mainly uses logs for storage. As mentioned in subsection 3.4, log-structured file systems optimize for sequential performance and are still relevant today. Modern log-structured file systems for flash devices are SFS [51] and F2FS [40]. SFS proposes the *cost-hotness policy*, where blocks with similar 'hotness' are assigned to groups for faster Garbage Collection. F2FS introduces multi-head logging: data blocks are categorized as "cold", "warm", or "hot" and are located at separate physical *zones* on the flash device. SFS and F2FS are designed to work with a wide range of flash devices, e.g. conventional block SSDs or Zoned Namespace (ZNS) devices. Therefore, they do not take advantage of the byte-addressable properties of PM.

In this subsection, we discuss NOVA [79] and Strata [39], log-structured file systems that take advantage of PM to create faster log-structured file systems.

NOVA The log-structured NOVA file system [79] aims to maximize PM performance while providing stronger consistency guarantees than BPFS [18], SCMFS [77], and Aerie [71]. As mentioned before, BPFS does not perform well in certain operations, e.g. a directory move. Furthermore, SCMFS does not provide any consistency guarantees for (meta) data. Another observation they make is that Aerie does not support atomic data operations.

Conventional log-structured file systems store metadata and actual data blocks in the log. NOVA deviates from this design in three design choices.

First, it proposes a logging structure where each inode is assigned a separate log to allow for parallelism. Synchronization primitives, e.g., mutexes or locks, are avoided by only allowing one open transaction at a time on each core. Logs are stored as linked lists in PM, so they can grow or shrink in length and do not need to be contiguous in memory.

Second, logs only store file metadata, so no data blocks. The data blocks are divided into pools, one per CPU. Each CPU maintains a red-black tree [53] in DRAM to keep track of free blocks in ascending order of addresses to enable fast merging and deallocation.

Third, NOVA uses journaling in cases where metadata updates span multiple inodes. For example, changing file permissions is a metadata-only operation; however, truncating a file updates the file data, file size, and modification date. In such cases, NOVA first commits the data. Then, the updated metadata is appended to the corresponding file inode log. Finally, NOVA journals all affected log tails.

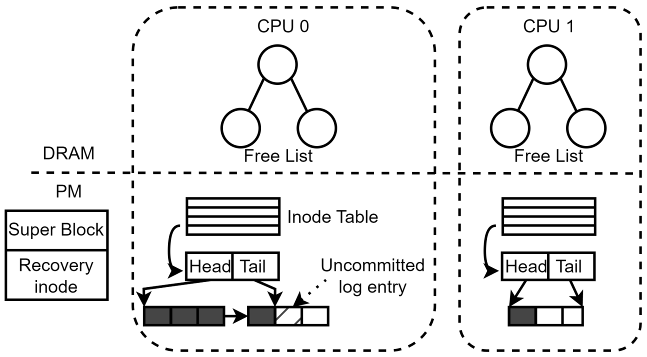

Figure 10: NOVA file system log-structure

Figure 10 graphically displays the layout of the NOVA file system. As mentioned before, note that each CPU maintains its own free list and inode table. Each table entry stores two pointers; one for the inode log head and tail, respectively. Additionally, NOVA stores a super block and a recovery inode. The first has a purpose similar to that of conventional file systems. The latter stores the page allocator state to allow faster recovery after normal shutdown.

In case of an improper shutdown, NOVA performs a recovery routine, consisting of two steps. First, NOVA rolls back any uncommitted file system transactions to bring the file system back into a consistent state. Second, each CPU reconstructs its free list by scanning the inode table, also known as a *log scan*.

Strata Another log-structured file system is Strata. The file system model of Strata is fundamentally different compared to the file systems we discussed before. These file systems assume that each file system is linked to a single physical device, in this case, PM. Strata, on the other hand, proposes a file system that spreads data across different storage devices. This enables Strata to capture the unique properties of multiple storage devices, such as PM, SSD, or HDD, in one file system [39].

To implement such a storage model, Strata implements a split architecture similar to what we have seen in SplitFS [37], as displayed in Figure 11. We will elaborate the most important components in this figure.

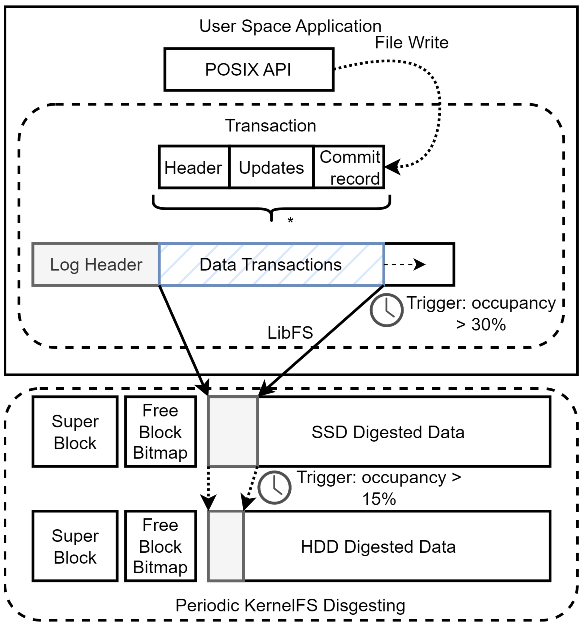

Figure 11: Strata file system log-structure and data digesting, triggered by log occupancy

LibFS exposes a POSIX-compliant interface to applications in user space. To attach fast write performance, each application is assigned a dedicated log for file system I/O operations, stored in fast PM. Note that this multi-log design differs from what we have seen in the NOVA FS; each log is assigned a separate log, instead of one log per file.

Applications access Strata through POSIX calls. File writes are transformed into transactions and appended to the log, as depicted in Figure 11. File reads are handled with the help of an inode-like data structure, which we will discuss in a moment.

In the kernel space, *KernelFS* is responsible for garbage collection and *digesting*: the process of aggregating file data into sequential disk areas to minimize fragmentation [39]. This means that in user space, where writes are not block-aligned, digesting ensures that device-level write amplification is minimized by coalescing writes into block-aligned writes favorable for the selected 'level'. KernelFS maintains multiple digest levels, where each level corresponds to a single storage device, as illustrated in Figure 11. Digesting is performed in the background and is initiated when the application log is filled beyond a threshold, for example, 30%.

Due to digesting, the file's data blocks may be scatted over multiple digest levels, complicating file reads. Therefore, Strata uses adapted inodes to structure file metadata. Each inode contains one or more *extent trees*, each representing a storage device. The tree nodes point directly to the file's data blocks.

Discussion In subsection 4.1, we discussed the first PM-capable file systems released between 2009 and 2016 (Figure 12). These file systems adapt the existing kernel I/O infrastructure to support PM. Additionally, they use DAX to bypass the OS page cache. Hybrid file systems use a different approach. These file systems enable the construction of PM-aware file systems in user space, allowing application-tailored optimizations. In addition, they reduce kernel overhead by offloading metadata management to user space, increasing throughput and parallel performance. This class of PM-file systems became mainstream, as seen in Figure 12.

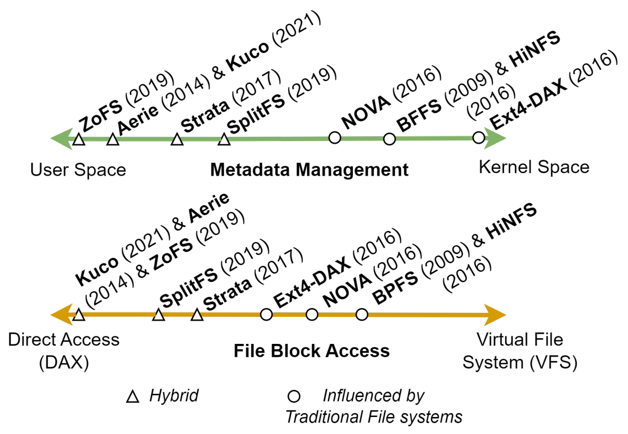

Figure 12: PM file systems positioning: *Metadata Management* and *File Block Access*

5 File Indexing Overhead

As discussed in the introduction (section 1), PM achieves performance close to DRAM, shifting the overhead from the device to the I/O stack. Multiple studies show that file indexing has a significant impact on performance [43, 54, 73]; in extreme cases, up to 45% of the total runtime [9]. In this section, we consider the two most prominent issues related to file indexing when using PM: *file mapping* and the *path walk* overhead in the Linux Virtual File System (VFS).

HashFS and ctFS aim to improve *file mapping* performance, that is, the operation of mapping a logical file offset to a physical location on the underlying storage device [54]. HashFS uses a *hash table* to extract more performance, while ctFS proposes a design in which the translation is performed in hardware using the MMU.

Afterward, we cover FlatFS and ByVFS, which decrease VFS path walk overhead. In a path walk, the VFS traverses a file path, for example, `"/foo/bar.txt"`, to return information about the file `bar.txt`.

5.1 Improving File Mapping Performance

The authors of HashFS [54] mention that multiple properties negatively impact the performance of a PM file system. First of all, file system fragmentation results in files being allocated in non-contiguous PM regions, resulting in larger mapping structures, which in turn causes poor search and insert performance. Take PMFS's B+ tree (section 4), for example. If a file is heavily fragmented the tree grows rapidly in size, resulting in slower tree traversal [6].

Another issue is related to the per-file mapping scheme. Many traditional file systems optimized for slow block devices (e.g. *ext4*) use a simple isolation mechanism: a read/write lock covers the entire structure to enforce mutual exclusion. Consequently, concurrent performance is limited; only one thread may write to a file in any given time frame.

HashFS' main contribution is to solve the aforementioned issues by implementing hash-based file mapping. Compared to Radix/Extent trees (see subsection 3.3), hash tables require considerably fewer and smaller memory accesses, therefore, are convenient to store in PM. It is important to understand that HashFS should be seen as a standalone optimization; its contributions can be used in unison with existing state-of-the-art PM file systems that we discussed in section 4.

The HashFS paper describes two implementations. The first implementation uses *Cuckoo* hashing; a form of hashing in which each entry is hashed twice using two different hash functions to avoid hash collisions [58]. Figure 13 displays an example in which a Cuckoo hash table lookup is performed. It consists of multiple steps:

(1) In the first step, two different hashing functions compute the hash for the logical block number 15. (2) The first hash points to the second entry in the hash table; however, its logical block is not equal to 15. (3) The second hash points to the correct entry as its logical block matches. (4) We find the corresponding physical block number by consulting the metadata structure stored in the hash table entry.

The second (final) HashFS implementation uses *linear probing*: a hashing function that avoids hash collisions by maintaining key-value pairs for each hash table entry that contains conflicting hash values. Compared to Cuckoo hashing, linear probing limits search overhead in the event of a hash collision, as conflicting entries are stored in adjacent

[6]A B+ tree worst-case time complexity is logarithmic: $O(log\ n)$ [6]

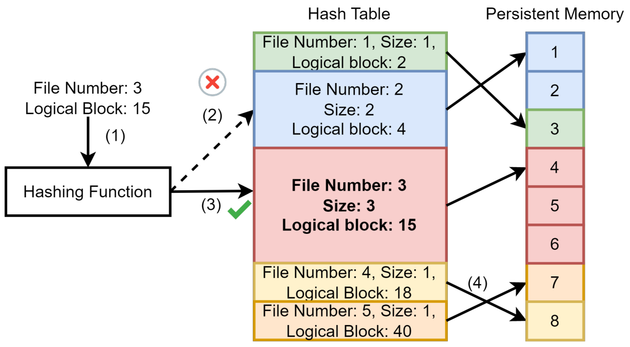
Hash Table
Persistent Memory
File Number: 3
Logical Block: 15
(1)
Hashing Function
(2)
(3)
File Number: 1, Size: 1, Logical block: 2
File Number: 2 Size: 2 Logical block: 4
File Number: 3 Size: 3 Logical block: 15
File Number: 4, Size: 1, Logical Block: 18
File Number: 5, Size: 1, Logical Block: 40
(4)
1
2
3
4
5
6
7
8

Figure 13: Cuckoo hash table lookup example

locations, which is beneficial for PM performance [68]. The operation of mapping a logical file offset to a physical PM location consists of multiple steps, as depicted in Figure 14:

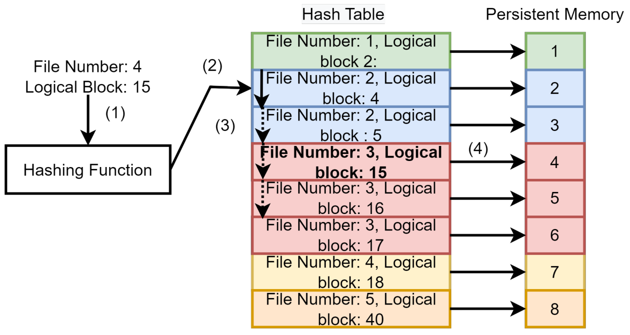
Hash Table
Persistent Memory
File Number: 4
Logical Block: 15
(1)
Hashing Function
(2)
(3)
File Number: 1, Logical block 2:
File Number: 2, Logical block: 4
File Number: 2, Logical block : 5
File Number: 3, Logical block: 15
File Number: 3, Logical block: 16
File Number: 3, Logical block: 17
File Number: 4, Logical block: 18
File Number: 5, Logical block: 40
(4)
1
2
3
4
5
6
7
8

Figure 14: HashFS file mapping operation using *Linear Probing*

(1) First, the logical block number 15 is hashed. (2) In the second step, we jump to the corresponding entry in the hash table. The logical block does not match, so a hash collision must have occurred. (3) We iterate over the conflicting entries in adjacent locations until we have found a match. (4) As the hash table entries and the physical PM locations are mapped one-to-one, the entry offset in the hash table is the physical block number.

This one-to-one mapping scheme makes it straightforward to achieve good parallel performance: approximately a $4.5\times$ decrease in latency for 4 kB sequential reads, $5\times$ decrease for 4 kB random reads, and $10\times$ decrease for 4 kB inserts compared to Radix Trees.

ctFS takes a different approach by offloading file mapping operations to the MMU [43]. In a Background section (subsection 3.2), we already mentioned the notion of a *hierachical page table*: a tree structure that maps virtual addresses to physical addresses. Note that a hierarchical page table and a Radix tree (see subsection 3.3) share a common feature; they both use simple offset calculations to traverse the tree. A key difference is that the Radix tree walk is implemented in software, whereas the faster page table walk is implemented entirely in hardware. Therefore, offloading the file mapping operation to the MMU could be very beneficial for performance.

ctFS introduces the notion of a *Persistent Page Table* (PPT): a page table that can be stored in PM. Conventional page tables are volatile structures, which means that, at shutdown, they are lost forever. In the context of performing a file mapping operation, this is undesirable behaviour, as logical to physical block mappings would be irreversibly lost after shutdown. Therefore, ctFS stores the page table entries used for file mapping entirely in PM. During initialization, ctFS copies PTT entries into the kernel's DRAM page table, which is then used by the MMU to perform fast address translation when accessing PM. In the event of a page fault, ctFS allocates a new persistent page, creates a new mapping in the PTT, and finally copies the mapping to the kernel page table.

Like SplitFS, ctFS uses a hybrid architecture where user and kernel space have distinct responsibilities. *ctU* manages the structure of the file system. It maintains various *partitions*. Each partition level contains blocks that are 8× the size of the partitions in the previous level, for example, 32 kB in level 1, since the partition size of level 0 is 8 kB. *ctK* makes sure that the MMU can perform the address translation and that the changes are persistent. It does so by ensuring that the kernel' DRAM page table mappings are an exact copy of those stored in the PPT in PM.

Figure 15 provides an example in which a file mapping operation is performed. Observe that the entire file system is mapped into user space. Suppose a user performs a read operation at virtual address `0x80001222`. To find the corresponding physical location in PM, the MMU performs a page table walk. More specifically, the MMU refers to the kernel page table to find the corresponding PTT entry, which contains the virtual-to-physical memory mapping, in this case `0x80001222` to `0x20008222`.

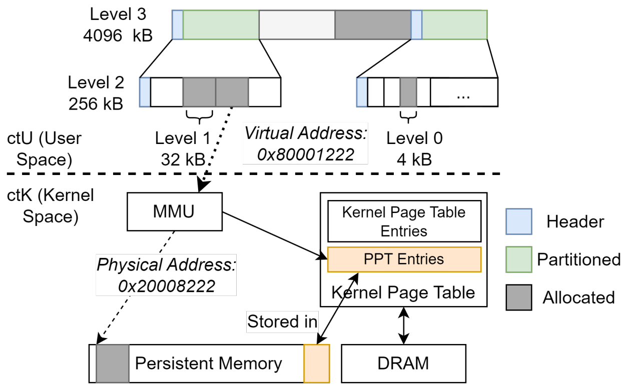

Figure 15: ctFS file mapping operation example

In short, we have seen two techniques to improve file mapping performance. Hash tables are convenient structures to store in PM and allow for fine-grained locking, resulting in improved parallel performance. Offloading the translation to the MMU is another promising technique.

5.2 Virtual File System Overhead

The Virtual File System (VFS) provides a software abstraction that enables access to different file systems using a uniform interface. In addition, it provides protection and concurrency.

The VFS caches multiple structures to improve performance, namely: the super block, directory entries (*dentry*), and file inode. The latter two are especially important for the *path walk*. During the walk, the VFS inserts the corresponding intermediate directory entries into a *dentry* cache [17]. This is beneficial for performance as applications are likely to access the same file again in the near future [73, 9].

In traditional slow block devices, e.g. an HDD, disk latencies are significantly higher than path walk latency. In such cases, caching intermediate directory entries is the logical thing to do. However, this claim does not apply to PM, as the time spent retrieving the entry from the *dentry* cache is greater than the latencies observed when writing/reading to PM, decreasing overall performance.

Based on this insight, the authors of ByVFS [73] suggest that removing the *dentry* cache would be beneficial for performance, especially in small I/O workloads. They claim to reduce the execution time by $\sim$ 48% for a *stat* call, using NOVA as the underlying file system.

The authors of the FlatFS paper [9] identified another issue. They acknowledge that ByVFS improves indexing performance significantly; however, they point out another issue related to the path walk. The files and directories contained in the namespace tree are physically scattered throughout the storage device, resulting in poor data locality and indirect memory accesses [9]. Additionally, the namespace tree traversal introduces random memory accesses as the directory entries of different directories are scattered across the device. Multiple studies [33, 80] have shown that such random access patterns result in suboptimal PM performance.

To solve these problems, FlatFS proposes the novel *coordinated file path model*. This model improves data locality performance by introducing a 'flat' namespace structure in which contiguous directory entries in a namespace are also stored consecutively within PM. A path walk only involves a single lookup, avoiding the aforementioned expensive namespace tree traversal.

The 'coordinated file path model' consists of two components: the traditional component-at-a-time model and the novel full-path-at-a-time model. The traditional walk model is included to accommodate namespace switches. Such a namespace switch may occur when certain semantic path components (e.g., dot-dot (..), a mount point change, or symbolic links) are encountered.

The 'full-path-a-at-a-time model' first performs preprocessing to speed up the lookup, e.g., dots, redundant slashes, or semantic path elements are removed. After preprocessing, this canonical path is passed to the *semantic path component finder*. This component searches for the corresponding inode in a persistent range-optimized B^r tree, after which a permission check occurs. A B^r tree is an adapted B tree that provides faster range operations [9], speeding up file system operations, such as a directory copy. Its tree nodes are 256 bytes aligned, the optimal memory access granularity of Intel Optane DC Persistent Memory [80]. A B^r tree lookup is performed using the *Write-optimized Compressed* (WoC) key as an index. Instead of using the full preprocessed path as an indexing key, FlatFS uses the smaller WoC key to reduce storage consumption. Every index key is divided into two parts, a prefix and a suffix. All keys in the same tree node share the same prefix, allowing for smaller index keys.

In addition, WoC keys use a complex caching layout to avoid write amplification. Suppose that we insert a new key into the B^r tree. This insertion may cause prefix expansion of other keys, leading to many small writes. Instead of storing the entire prefix within each key, FlatFS caches prefixes in DRAM and only adjusts the *prefix* size when necessary, avoiding write amplification and costly cache line flushes.

Using the aforementioned techniques, FlatFS achieves stable path latency, regardless of the file path length. Moreover, FlatFS outperforms a hot *dentry* cache, which strengthens the claim that a directory cache imposes a performance penalty when using PM.

To summarize, we have seen that a substantial amount of VFS overhead comes down to caching and an expensive path walk. Due to PM's low read latency, some caching structures inside the VFS serve no purpose, in particular the directory cache. Path walk performance can be improved by a coordinated file path walk model.

6 Data Crash-Consistency

In this section, we investigate how persistent memory file systems guarantee data crash consistency. First, we classify the PM file systems, discussed in Sections 4 and 5, by the high-level consistency technique(s) they employ, i.e., as shadow paging, log-structuring, journaling, or more exotic/hybrid variants. We already provided the relevant background for the conventional well-known persistence techniques in subsection 3.4. Therefore, we focus on more novel/innovative designs that provide persistence guarantees while maintaining good performance. Subsequently, we address the issue of *write reordering*.

This combination of data consistency and enforced write ordering allows PM file systems to support ACID transactions, that is, transactions that adhere to the properties of Atomicity, Consistency, Isolation, and Durability [32]. An operation is *atomic* if and only if updates are committed in all or none manner [25]. Data *consistency* implies that memory writes must be performed in a strict format/order for correct recovery in the event of a crash. *Isolation* ensures that transactions do not affect each other. *Durability* ensures that the data affected by a completed transaction must be persistent, even in the event of system failure [44, 76].

6.1 Consistency Techniques

Table 5 provides an overview that relates different crash-consistency techniques to the PM file systems we discussed in previous sections. Based on this overview, we can derive multiple interesting trends.

(1) First, note that several file systems (PMFS [25], HiNFS [57], NOVA [79], ctFS [43], Kuco [15], ZoFS [23]) use a hybrid approach. In this approach, small (meta) data updates are performed using atomic in-place updates, whereas more performance-degrading techniques handle larger writes, for example, journaling [18]. Suppose that we want to append to a file. Instead of inserting a new log entry into the journal consisting of all modifications performed, e.g. data blocks, file modification date, file size, etc., we distinguish between small and large writes. Small writes, such as modifying the file modification date or size, can be performed using low-cost atomic instructions, e.g., Intel's 64-byte atomic store instructions [7]. The atomicity and durability of larger writes is enforced through the more performance-diminishing crash-consistency techniques we discussed in subsection 3.4: *Journaling*, *Shadow Paging*, *Log-Structuring*.

(2) Second, an emerging trend is that PM file systems use *operation logging* (subsection 3.4) to record transactions. Bhat et al. [5], authors of ScaleFS, have shown that an in-memory file system, in combination with an operation log, results in reduced write amplification and improved concurrent performance on slow disk devices compared to more conventional undo/redo logging. These forms of logging involve either a full copy to record data in the log (Ext4-DAX), perform a lazy copy using Shadow Paging (BPFS), or require extensive modification of file system metadata structures (PMFS, HiNFS) [39]. SplitFS [37], Strata [39] and Kuco [15] bring the concept of operation logging to PM file systems.

In SplitFS and Kuco, the log entries do not contain file data; instead, they only contain a pointer to the staging file in memory [37]. Log entries persist by performing 64-byte atomic writes. To ensure that log entries persist in the right order, each log write is accompanied by a `sfence` memory barrier. File systems that use redo/undo logging, such as NOVA or PMFS, use the tail pointer to revert changes in case of failure; hence, they update the tail pointer using expensive `clflush` and `sfence` instructions. In SplitFS, the tail pointer can be reconstructed in a crash, so there is no need to store it in PM. It is stored in DRAM and is atomically advanced using atomic Compare-and-Swap (CAS) operations, resulting in better parallel performance.

[7] See Section 8.2.4 - Intel Architecture Software Developer Manual [35].

Method → File System ↓	Atomic in-place updates	Log-structuring	Shadow Paging	Journaling
BPFS			✓ *Short-Circuit Shadow Paging*	
PMFS	✓ Small Metadata Updates		✓ Only for Data Blocks	✓ Undo Logging, for Large Metadata Updates
HiNFS	✓ Small Metadata Updates		✓ Only for Data Blocks	✓ Undo Logging, for Large Metadata Updates
Ext4-DAX				✓ Redo Logging*
SCMFS				
SplitFS			✓ Using `relink` primitive	✓ Operation Logging to record file operations
Aerie				✓ Redo Logging
NOVA	✓ Small Metadata Updates	✓ Large Metadata Updates		✓ Metadata Updates spanning multiple inodes
Strata				✓ Operation Logging in user space, Redo Log for digest areas in kernel space
ctFS			✓ Only for Data Blocks, using `pswap` primitive	✓ Redo Logging for Metadata Updates
Kuco			✓ Only for Data Blocks	✓ Operation Logging to record file operations
ZoFS	✓ Small Metadata Updates			
HashFS	✓ Hash Table Insertions	**	**	**

Table 5: PM file systems crash-consistency techniques. Orange cells represent hybrid implementations. *: optional, **: HashFS only implements PM-optimized file mapping, file block consistency should be enforced through the PM file system [54].

The Strata file system uses a hybrid setup. In user space, operation logging is performed like SplitFS, except that Strata stores per-inode log pointers in PM. Eventually, the data contained in PM gets digested into block updates, which are then stored in redo logs located in the kernel digest areas.

(3) The last trend is related to Shadow Paging. Although Shadow Paging avoids extensive in-place updates, it still suffers from high write amplification due to write propagation [18]. As discussed in section 4, BPFS avoids write amplification by its *Short-Circuit Shadow Paging*. However, it still incurs a large overhead when performing operations that cover a large part of the file system tree [79].

Recent work aims to address this issue by introducing two new primitives that perform Shadow Paging without data movement. We already discussed the first primitive, `relink`, in section 4. Using `relink`, contiguous data regions can be moved atomically without any physical data movement, as seen in Figure 9 [37]. The second primitive, `pswap`, is implemented as a system call within ctFS's kernel space library, *ctK*. Atomically, it swaps the Page Table Entries (PTEs) corresponding to two same-sized contiguous virtual memory regions in the Persistent Page Table (PPT).

We illustrate the purpose of `pswap` using an example. Suppose that a user wants to append to a file x at offset z. First, *ctU* (ctFS's user space library) allocates a *staging* partition $P1$ in the PTT and copies the data to the same offset z within this partition. Then, *ctU* invokes `pswap` to atomically merge the original data contained in partition $P0$ with the new data in $P1$, persisting the data. Metadata updates are recorded in the redo log.

In short, we have seen four consistency techniques used in PM file systems: atomic in-place updates, log-structuring, shadow paging, and journaling. Most PM file systems implement a hybrid approach, as they identified that storing both data and metadata in a log results in poor performance. In the hybrid approach, data blocks are made consistent by performing inexpensive shadow paging using `relink` or `pswap` directives. Metadata persistence is achieved through a low-cost *operation log*: a log that only stores file/directory operations, and not the actual data involved.

6.2 Enforcing Write Ordering

We already touched on the issue of *write ordering* in the Introduction and Background (Sections 1, 3). In short, the order in which data is written to disk may differ from the user's intentions, as a CPU can reorder writes for performance, resulting in data inconsistencies before entering the Persistence

Domain (PD). According to the relevant literature [18, 4, 35, 45], there are four options to enforce write ordering.

(1) The first option is to completely bypass the cache by performing *write-through caching*: the cache and the actual PM location are written at the same time. In 2011, Bhandari et al. [4] mentioned that this form of ordering has a slight advantage in CoW-based PM file systems. However, this claim should be taken with a grain of salt, as PM was not yet mainstream, thus platform support was minimal.

(2) Another option is to flush the entire cache at each memory barrier. A side effect is that the performance of other applications may degrade as its working set may be (partly) evicted from the cache.

(3) Alternatively, we can perform a more fine-grained flush. Instead of flushing the entire cache, we keep track of the cache lines in use and only flush those that contain file system (meta) data. Intel supports selective flushing by the `clflush` instruction. A `mfence` instruction ensures that all load and store instructions issued before `mfence` are serialized in the order they were performed. The `sfence` and `lfence` instructions provide this guarantee solely for store and load instructions, respectively [35].

Although this form of ordering does not degrade the performance of the cache for other applications, it is still expensive. Bhandari et al. [4] mention that one cache line flush takes around 300 CPU cycles on an Intel(R) Xeon(R) E5620 @ 2.4 GHz processor with a total of 12 MB cache.

To improve performance, Intel added two new instructions, namely `clflushopt` and `clwb`. The first provides an unordered version of `clflush`, allowing some concurrency when running multiple PM load/store instructions back-to-back. `clwb` behaves similarly to `clflushopt`, but does not invalidate the cache line [60, 69, 45].

(4) The final option is to allow software to explicitly communicate ordering constraints to hardware. The CPU is free to perform read and write caching, but must ensure that the ordering constraints are satisfied. In 2009, BPFS [18] proposed a new hardware extension called a *epoch barrier*: a sequence of PM writes from the same thread delimited by a memory barrier issued in software. In 2015, Intel released this extension by introducing the `PCOMMIT` instruction. This instruction ensures that a user-specified memory range is written to persistent storage [45] [8].

Table 5 displays an overview that relates the ordering techniques mentioned above to PM file systems. The file systems are ordered chronologically by their release date. One trend we can observe is that more recent work tends to use PM-specialized flush instructions, such as `clflushopt`, while older file systems use a more conventional `clflush` approach.

In summary, we have seen multiple options to enforce write ordering: bypass the cache entirely, coarse/fine-grained flushing, and the *epoah barrier*. Flushing the cache at cache line granularity using `clflushopt` is the preferred method of choice, as it allows for some concurrency when PM load/store instructions are executed.

[8]This instruction was later deprecated in favor of `clflushopt` and `clwb` [62]

Applicable to → File System ↓	Metadata	Data Blocks
BPFS	✓ (4): `epochs`	✓ (4): `epochs`
PMFS	✓ (3, 4)	✗
HiNFS	✓ (3): `clflush` & `mfence`	✗
Ext4-DAX	✗	✗
SCMFS	✓ (3): `clflush` & `mfence`	✓ (2)
SplitFS	✓ (3): `clflush` & `sfence`	✓ (3): `clflush` & `sfence`
Aerie	✓ (3): `clflush` & `sfence`	✓ (3): `clflush` & `sfence`
NOVA	✓ (3): `clflushopt` & `clwb`	✓ (3): `clflushopt` & `clwb`
Strata	✓ (3): `clflushopt`* or `clflush`	✓ (3): `clflushopt`* or `clflush`
Kuco	Not Specified	Not Specified
ZoFS	✓ (3): `clflushopt` & `clwb`	✓ (3): `clflushopt` & `clwb`
HashFS	Not Applicable	Not Applicable
ByVFS	Not Applicable	Not Applicable

Table 6: PM file systems ordering enforcing techniques. *: preferred, if CPU support PMEM extensions

7 Open Problems and Future Work

Although Persistent Memory brings exciting performance improvements to applications, widespread adoption has not (yet) been reached. There seems to be no consensus among application developers about where to use PM and what benefits it provides [26]. In addition, the best-performing file systems, Kuco, ZoFS and ctFS, require massive application refactoring as POSIX-like file semantics are not available. As a result, there is a trade-off to be made between user convenience and performance, which we have not covered in this work.

In July 2022, Intel discontinued the Intel Optane product line. However, we do not expect software support to be discontinued in the near future; hence, PM-related research will continue. There are already emerging alternative devices: Kioxia and Everspin [65]. Another promising alternative is called the Compute Express Link (CXL): a cache-coherent interconnect running on top of PCIe [78]. The early file system prototypes released by Microsoft [42] and Meta [47] show promising results. Further work may include a study that looks at the advantages and differences of this technology compared to Intel's Optane PM. Other studies can dive into the specifics of implementing a CXL file system: "How CXL devices interact with modern workloads?", "What is the position of CXL memory inside the server micro-architecture?".

8 Conclusion

In this survey, we have discussed how file systems address the three most prominent challenges when using Persistent Memory (PM): the overhead shift from the device to the host I/O stack, indexing overhead, and data persistence.

Before we answer the main research question, we first provide an answer for each of the subquestions:

- RQ1 - How have the properties/features of Persistent Memory led to changes in file system design?

  First, PM file systems bypass the page cache using Direct Access (DAX) to allow the user to directly modify file data in user space using MMU mappings. Other work focuses on the role of the kernel. In older PM file systems, the kernel bears full responsibility for maintaining metadata in a consistent state. More recent work shifts more responsibility to the user (see Figure 12), which in turn results in greater flexibility for applications to design custom-tailored file systems and extract the full potential of PM.

- RQ2 - "Which optimizations help to decrease file indexing overhead in small I/O workloads?":

  The file indexing overhead is due to poor file mapping performance and the expensive VFS *path walk*. The first issue is addressed by the introduction of a PM-optimized hash table. Another promising approach is to offload the translation to hardware via the Memory Management Unit. The performance impact of the VFS can be reduced by removing the redundant directory cache and introducing an adapted file path walker.

- RQ3 - "How can Persistent Memory file systems guarantee data crash consistency?":

  Data consistency must be enforced in software and hardware. In software, most file systems use a hybrid approach where data persistence is enforced through inexpensive Shadow Paging and metadata persistence through an *operation log* and atomic in-place updates. At the hardware level, file systems use PM-optimized flushing instructions to enforce data ordering.

Now, we can answer the main research question, "Which file system design changes are needed to cope with the challenges that arise when using Persistent Memory?": We can definitely conclude that significant changes to the existing storage stack are necessary to extract the full performance potential of PM storage. When possible, the page cache should be avoided, as it serves no purpose in PM. Exposing device control to user space allows applications to fully leverage PM's byte-addressable properties. Data persistence must be enforced at the software and hardware level.

Glossary

ADR Asynchronous DRAM Refresh. 5

AIT Address Translation Table. 5

ASLR Address Space Layout Randomization. 11

CAS Compare-and-Swap Operation. 17

CoW Copy-on-Write. 8

CPU Central Processing Unit. 1

CXL Compute Express Link. 19

DAX Direct Access. 1, 5, 6, 9

eADR Extended Asynchronous DRAM Refresh. 5

HDD Hard Disk Drive. 1

KV Key-Value. 3, 6

LSM Log-Structured Merge. 6

MMU Memory Management Unit. 2, 4, 10

PD Persistence Domain. 5, 19

PM Persistent Memory. 1, 3, 20

PPT Persistent Page Table. 16, 18

PTE Page Table Entry. 18

RPC Remote Procedure Call. 12

SCM Storage Class Memory. 1, 10

SSD Solid State Drive. 1, 3

TLB Translation Lookaside Buffer. 11

VFS Virtual File System. 2, 6

WA Write Amplification. 6, 8

WAL Write-Ahead Log. 7

WoC Write-optimized Compressed Key. 17

WPQ Write Pending Queue. 5

ZNS Zoned Namespace. 13

References

[1] R. Agarwal. *Journaling and Log-Structured File Systems*. New York, NY, USA, 2020.

[2] Katelin Bailey et al. “Operating system implications of fast, cheap, non-volatile memory”. In: *Proceedings of the 13th USENIX conference on Hot topics in operating systems*. HotOS’13. USA: USENIX Association, 2011, p. 2. (Visited on 12/07/2022).

[3] Livio Baldini Soares and Michael Stumm. ““FlexSC: Flexible System Call Scheduling with Exception-Less System Calls,””. In: Jan. 2010, pp. 33–46.

[4] K. Bhandari, D.R. Chakrabarti, and H.-J Boehm. “Implications of CPU caching on byte-addressable non-volatile memory programming”. In: (Jan. 2012).

[5] Srivatsa S. Bhat et al. “Scaling a file system to many cores using an operation log”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 69–86. ISBN: 978-1-4503-5085-3. DOI: *10.1145/3132747.3132779*. URL: *http://doi.org/10.1145/3132747.3132779* (visited on 11/27/2022).

[6] Anastasia Braginsky and Erez Petrank. “A lock-free B+tree”. In: *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. SPAA ’12. New York, NY, USA: Association for Computing Machinery, June 2012, pp. 58–67. ISBN: 978-1-4503-1213-4. DOI: *10.1145/2312005.2312016*. URL: *http://doi.org/10.1145/2312005.2312016* (visited on 12/12/2022).

[7] *btrfs - A modern copy on write(CoW) filesystem for Linux*. en. Aug. 2021. URL: *https://iceberg988.github.io/posts/btrfs/* (visited on 11/20/2022).

[8] Paul Caheny et al. “Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach”. In: *IEEE Transactions on Parallel and Distributed Systems* 29.5 (May 2018). Conference Name: IEEE Transactions on Parallel and Distributed Systems, pp. 1174–1187. ISSN: 1558-2183. DOI: *10.1109/TPDS.2017.2787123*.

[9] Miao Cai. “FlatFS: Flatten Hierarchical File System Namespace on Non-volatile Memories”. In: *Usenix* (July 2022). URL: *https://www.usenix.org/system/files/atc22-cai.pdf* (visited on 05/10/2022).

[10] Miao Cai and Hao Huang. “A survey of operating system support for persistent memory”. en. In: *Frontiers of Computer Science* 15.4 (Feb. 2021), p. 154207. ISSN: 2095-2236. DOI: *10.1007/s11704-020-9395-3*. URL: *https://doi.org/10.1007/s11704-020-9395-3* (visited on 11/20/2022).

[11] Saarland Informatics Campus. *Cache Latencies*. URL: https://uops.info/cache.html (visited on 12/08/2022).

[12] Feng Chen, David A. Koufaty, and Xiaodong Zhang. "Understanding intrinsic characteristics and system implications of flash memory based solid state drives". In: *ACM SIGMETRICS Performance Evaluation Review* 37.1 (June 2009), pp. 181–192. ISSN: 0163-5999. DOI: 10.1145/2492101.1555371. URL: http://doi.org/10.1145/2492101.1555371 (visited on 11/20/2022).

[13] Shimin Chen and Qin Jin. "Persistent B+-trees in non-volatile main memory". In: *Proceedings of the VLDB Endowment* 8.7 (Feb. 2015), pp. 786–797. ISSN: 2150-8097. DOI: 10.14778/2752939.2752947. URL: http://doi.org/10.14778/2752939.2752947 (visited on 11/24/2022).

[14] Shuo-Han Chen et al. "Enabling union page cache to boost file access performance of NVRAM-based storage device". In: *Proceedings of the 55th Annual Design Automation Conference*. DAC '18. New York, NY, USA: Association for Computing Machinery, June 2018, pp. 1–6. ISBN: 978-1-4503-5700-5. DOI: 10.1145/3195970.3196045. URL: http://doi.org/10.1145/3195970.3196045 (visited on 11/14/2022).

[15] Youmin Chen. "Kuco: Scalable Persistent Memory File System with Kernel-Userspace Collaboration". In: *Proceedings of the 19th USENIX Conference on File and Storage Technologies.* (Feb. 2021). URL: https://www.usenix.org/system/files/fast21-chen-youmin.pdf.

[16] Douglas Comer. "Ubiquitous B-Tree". In: *ACM Computing Surveys* 11.2 (June 1979), pp. 121–137. ISSN: 0360-0300. DOI: 10.1145/356770.356776. URL: http://doi.org/10.1145/356770.356776 (visited on 11/22/2022).

[17] Kernel Development Community. *Pathname lookup — The Linux Kernel documentation*. Kernel version 6.1.0. URL: https://www.kernel.org/doc/html/latest/filesystems/path-lookup.html (visited on 12/13/2022).

[18] Jeremy Condit et al. "Better I/O through byte-addressable, persistent memory". en. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*. Big Sky, Montana, USA: ACM Press, 2009, p. 133. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629589. URL: http://portal.acm.org/citation.cfm?doid=1629575.1629589 (visited on 10/05/2022).

[19] B. Jack Copeland. "The Modern History of Computing". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2020. Metaphysics Research Lab, Stanford University, 2020. URL: https://plato.stanford.edu/archives/win2020/entries/computing-history/ (visited on 11/14/2022).

[20] Jonathan Corbet. *Memory Protection Keys*. May 2015. URL: https://lwn.net/Articles/643797/ (visited on 11/28/2022).

[21] Alvin Cox. "JEDEC SSD Specifications Explained". en. In: JEDEC Standard (). URL: https://www.jedec.org/sites/default/files/Alvin_Cox%20[Compatibility%20Mode]_0.pdf.

[22] Biplob Debnath, Sudipta Sengupta, and Jin Li. "FlashStore: high throughput persistent key-value store". en. In: *Proceedings of the VLDB Endowment* 3.1-2 (Sept. 2010), pp. 1414–1425. ISSN: 2150-8097. DOI: 10.14778/1920841.1921015. URL: https://dl.acm.org/doi/10.14778/1920841.1921015 (visited on 10/31/2022).

[23] Mingkai Dong et al. "Performance and protection in the ZoFS user-space NVM file system". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 478–493. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359637. URL: http://doi.org/10.1145/3341301.3359637 (visited on 11/28/2022).

[24] Siying Dong et al. "RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications". In: *ACM Transactions on Storage* 17.4 (Oct. 2021), 26:1–26:32. ISSN: 1553-3077. DOI: 10.1145/3483840. URL: http://doi.org/10.1145/3483840 (visited on 10/31/2022).

[25] Subramanya R. Dulloor et al. "PMFS: System software for persistent memory". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. New York, NY, USA: Association for Computing Machinery, Apr. 2014, pp. 1–15. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592814. URL: http://doi.org/10.1145/2592798.2592814 (visited on 10/27/2022).

[26] Russell Fellows. *The Future of Optane and Persistent Memory*. nl. Mar. 2021. URL: https://www.linkedin.com/pulse/future-optane-persistent-memory-russell-fellows-1e (visited on 11/30/2022).

[27] Steve Fingerhut. *Does Storage break Moore's Law? A Look at SSD vs HDD*. en-US. July 2014. URL: https://blog.westerndigital.com/does-storage-break-moores-law/ (visited on 12/07/2022).

[28] Robert E. Fontana and Gary M. Decad. "Moore's law realities for recording systems and memory storage components: HDD, tape, NAND, and optical". In: *AIP Advances* 8.5 (May 2018). Publisher: American Institute of Physics, p. 056506. DOI: 10.1063/1.5007621. URL: http://aip.scitation.org/doi/10.1063%2F1.5007621 (visited on 12/07/2022).

[29] Linux Foundation. *Direct Access for files*. URL: *https://www.kernel.org/doc/Documentation/filesystems/dax.txt* (visited on 11/08/2022).

[30] Linux Foundation. *ext4 Data Structures and Algorithms — The Linux Kernel documentation*. URL: *https://www.kernel.org/doc/html/latest/filesystems/ext4/globals.html#super-block* (visited on 11/06/2022).

[31] Linux Foundation. *Page Table Management*. URL: *https://www.kernel.org/doc/gorman/html/understand/understand006.html* (visited on 11/23/2022).

[32] Jinyu Gu et al. "Pisces: a scalable and efficient persistent transactional memory". In: *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '19. USA: USENIX Association, July 2019, pp. 913–928. ISBN: 978-1-939133-03-8. (Visited on 11/26/2022).

[33] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. "Understanding the idiosyncrasies of real persistent memory". In: *Proceedings of the VLDB Endowment* 14.4 (Feb. 2021), pp. 626–639. ISSN: 2150-8097. DOI: *10.14778/3436905.3436921*. URL: *http://doi.org/10.14778/3436905.3436921* (visited on 10/27/2022).

[34] IAIK. *Paging on Intel x86-64 – IAIK*. en-US. URL: *https://www.iaik.tugraz.at/teaching/materials/os/tutorials/paging-on-intel-x86-64/* (visited on 11/23/2022).

[35] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Apr. 2022. URL: *https://cdrdv2.intel.com/v1/dl/getContent/671200*.

[36] Joseph Izraelevitz et al. *Basic Performance Measurements of the Intel Optane DC Persistent Memory Module*. arXiv:1903.05714 [cs]. Aug. 2019. DOI: *10.48550/arXiv.1903.05714*. URL: *http://arxiv.org/abs/1903.05714* (visited on 11/16/2022).

[37] Rohan Kadekodi et al. "SplitFS: reducing software overhead in file systems for persistent memory". en. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. Huntsville Ontario Canada: ACM, Oct. 2019, pp. 494–508. ISBN: 978-1-4503-6873-5. DOI: *10.1145/3341301.3359631*. URL: *https://dl.acm.org/doi/10.1145/3341301.3359631* (visited on 10/05/2022).

[38] Jinhyung Koo et al. "Modernizing File System through In-Storage Indexing". en. In: ODSI '21. 2021, pp. 75–92. ISBN: 978-1-939133-22-9. URL: *https://www.usenix.org/conference/osdi21/presentation/koo* (visited on 11/14/2022).

[39] Youngjin Kwon et al. "Strata: A Cross Media File System". en. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. Shanghai China: ACM, Oct. 2017, pp. 460–477. ISBN: 978-1-4503-5085-3. DOI: *10.1145/3132747.3132770*. URL: *https://dl.acm.org/doi/10.1145/3132747.3132770* (visited on 10/05/2022).

[40] Changman Lee et al. "{F2FS}: A New File System for Flash Storage". en. In: 13th USENIX Conference on File and Storage Technologies (FAST 15). 2015, pp. 273–286. ISBN: 978-1-931971-20-1. URL: *https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee* (visited on 11/20/2022).

[41] Adam Leventhal. "Flash storage memory". In: *Communications of the ACM* 51.7 (July 2008), pp. 47–51. ISSN: 0001-0782. DOI: *10.1145/1364782.1364796*. URL: *http://doi.org/10.1145/1364782.1364796* (visited on 11/14/2022).

[42] Huaicheng Li et al. "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms". In: *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23) (* (Oct. 2022). arXiv:2203.00241 [cs]. URL: *http://arxiv.org/abs/2203.00241* (visited on 11/30/2022).

[43] Ruibin Li. "ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory". In: *20th USENIX Conference on File and Storage Technologies (FAST 22)*. FAST' 22 (2022). URL: *https://www.usenix.org/conference/fast22/presentation/li*.

[44] Youyou Lu et al. "Loose-Ordering Consistency for persistent memory". In: *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. ISSN: 1063-6404. Oct. 2014, pp. 216–223. DOI: *10.1109/ICCD.2014.6974684*.

[45] Dan Luu. *CLWB and PCOMMIT*. URL: *https://danluu.com/clwb-pcommit/* (visited on 11/27/2022).

[46] Hector Marco-Gisbert and Ismael Ripoll Ripoll. "Address Space Layout Randomization Next Generation". en. In: *Applied Sciences* 9.14 (Jan. 2019). Number: 14 Publisher: Multidisciplinary Digital Publishing Institute, p. 2928. ISSN: 2076-3417. DOI: *10.3390/app9142928*. URL: *https://www.mdpi.com/2076-3417/9/14/2928* (visited on 11/06/2022).

[47] Hasan Al Maruf et al. *TPP: Transparent Page Placement for CXL-Enabled Tiered Memory*. arXiv:2206.02878 [cs]. June 2022. URL: *http://arxiv.org/abs/2206.02878* (visited on 11/30/2022).

[48] Avantika Mathur et al. "The new ext4 filesystem: Current status and future plans". In: *Proceedings of the Linux Symposium* (Jan. 2007).

[49] Richard McDougall. *FileBench*. 2004. URL: *http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf*.

[50] Paul McKenney. "Memory Ordering in Modern Microprocessors, Part I | Linux Journal". In: *Linux Journal* (June 2005). URL: *https://www.linuxjournal.com/article/8211* (visited on 11/16/2022).

[51] Changwoo Min. "SFS: Random Write Considered Harmful in Solid State Drives". en. In: *10th USENIX Conference on File and Storage Technologies (FAST 12)*. 2012. URL: *https://www.usenix.org/conference/fast12/sfs-random-write-considered-harmful-solid-state-drives* (visited on 11/20/2022).

[52] Jayashree Mohan, Rohan Kadekodi, and Vijay Chidambaram. *Analyzing IO Amplification in Linux File Systems*. arXiv:1707.08514 [cs]. July 2017. DOI: *10.48550/arXiv.1707.08514*. URL: *http://arxiv.org/abs/1707.08514* (visited on 11/06/2022).

[53] John Morris. *Data Structures and Algorithms: Red-Black Trees*. URL: *https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html* (visited on 11/21/2022).

[54] Ian Neal. "HashFS: Rethinking File Mapping for Persistent Memory". In: *Proceedings of the 19th USENIX Conference on File and Storage Technologies* (Feb. 2021). URL: *https://www.usenix.org/system/files/fast21-neal.pdf*.

[55] *NVM Programming Model (Version 1.2)*. June 2017. URL: *https://www.snia.org/sites/default/files/technical-work/npm/release/SNIA-NVM-Programming-Model-v1.2.pdf*.

[56] Patrick O'Neil et al. "The log-structured merge-tree (LSM-tree)". en. In: *Acta Informatica* 33.4 (June 1996), pp. 351–385. ISSN: 1432-0525. DOI: *10.1007/s002360050048*. URL: *https://doi.org/10.1007/s002360050048* (visited on 11/24/2022).

[57] Jiaxin Ou, Jiwu Shu, and Youyou Lu. "HiNFS: A high performance file system for non-volatile main memory". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. New York, NY, USA: Association for Computing Machinery, Apr. 2016, pp. 1–16. ISBN: 978-1-4503-4240-7. DOI: *10.1145/2901318.2901324*. URL: *http://doi.org/10.1145/2901318.2901324* (visited on 10/27/2022).

[58] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo hashing". en. In: *Journal of Algorithms* 51.2 (May 2004), pp. 122–144. ISSN: 0196-6774. DOI: *10.1016/j.jalgor.2003.12.002*. URL: *https://www.sciencedirect.com/science/article/pii/S0196677403001925* (visited on 11/22/2022).

[59] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. "System evaluation of the Intel optane byte-addressable NVM". In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, Sept. 2019, pp. 304–315. ISBN: 978-1-4503-7206-0. DOI: *10.1145/3357526.3357568*. URL: *http://doi.org/10.1145/3357526.3357568* (visited on 11/16/2022).

[60] *Persistent Memory Extensions - x86 - WikiChip*. en. May 2021. URL: *https://en.wikichip.org/wiki/x86/persistent_memory_extensions* (visited on 11/27/2022).

[61] *RocksDB: A Persistent Key-Value Store for Flash and RAM Storage Optimized (for PMEM)*. URL: *https://github.com/pmem/pmem-rocksdb*.

[62] Andy Rudoff. *Deprecating the PCOMMIT Instruction*. en. Sept. 2016. URL: *https://www.intel.com/content/www/us/en/developer/articles/technical/deprecate-pcommit-instruction.html* (visited on 11/16/2022).

[63] Georg Sauthoff. *On the Costs of Syscalls*. Aug. 2021. URL: *https://gms.tf/on-the-costs-of-syscalls.html* (visited on 08/11/2022).

[64] Margo Seltzer et al. "An Implementation of a Log-Structured File System for UNIX". In: USENIX '1993. Jan. 1993, pp. 307–326.

[65] Simon Sharwood. "Last week Intel killed Optane. Competing tech keeps coming". en. In: *The A Register* (Feb. 2022). URL: *https://www.theregister.com/2022/08/02/kioxia_everspin_persistent_memory/* (visited on 11/30/2022).

[66] John G. Spooner. "Intel Previews Potential Replacement for Flash Memory". en-US. In: *eWEEK* (Sept. 2006). URL: *https://www.eweek.com/pc-hardware/intel-previews-potential-replacement-for-flash-memory/* (visited on 11/29/2022).

[67] Chao Su and Qingkai Zeng. "Survey of CPU Cache-Based Side-Channel Attacks: Systematic Analysis, Security Models, and Countermeasures". en. In: *Security and Communication Networks* 2021 (June 2021). Publisher: Hindawi, e5559552. ISSN: 1939-0114. DOI: *10.1155/2021/5559552*. URL: *https://www.hindawi.com/journals/scn/2021/5559552/* (visited on 12/08/2022).

[68] Steven Swanson. *Early Measurements of Intel's 3DX-Point Persistent Memory DIMMs*. en-US. Apr. 2019. URL: *https://www.sigarch.org/early-measurements-of-intels-3dxpoint-persistent-memory-dimms/* (visited on 11/22/2022).

[69] Micheal Swift. *Hardware Support for NVM Programming*. Istanbul, Mar. 2015. URL: *https://research.cs.wisc.edu/sonar/tutorial/03-hardware.pdf*.

[70] Liu Tianhua et al. “The Design and Implementation of Zero-Copy for Linux”. In: *2008 Eighth International Conference on Intelligent Systems Design and Applications*. Vol. 1. ISSN: 2164-7151. Nov. 2008, pp. 121–126. DOI: *10.1109/ISDA.2008.102*.

[71] Haris Volos et al. “Aerie: flexible file-system interfaces to storage-class memory”. In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys ’14. New York, NY, USA: Association for Computing Machinery, Apr. 2014, pp. 1–14. ISBN: 978-1-4503-2704-6. DOI: *10.1145/2592798.2592810*. URL: *http://doi.org/10.1145/2592798.2592810* (visited on 10/27/2022).

[72] Hu Wan et al. “Empirical study of redo and undo logging in persistent memory”. In: *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. Aug. 2016, pp. 1–6. DOI: *10.1109/NVMSA.2016.7547178*.

[73] Ying Wang. “ByVFS: Caching or Not: Rethinking Virtual File System for Non-Volatile Main Memory”. In: USENIX HotStorage ’18 (Sept. 2018). URL: *https://www.usenix.org/system/files/conference/hotstorage18/hotstorage18-paper-wang.pdf* (visited on 05/10/2022).

[74] Lukas Waymann. *A Survey of CPU Caches*. en. Oct. 2017. URL: *https://meribold.org/2017/10/20/survey-of-cpu-caches/* (visited on 12/07/2022).

[75] Claes Wohlin. “Guidelines for snowballing in systematic literature studies and a replication in software engineering”. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–10. ISBN: 978-1-4503-2476-2. DOI: *10.1145/2601248.2601268*. URL: *http://doi.org/10.1145/2601248.2601268* (visited on 11/29/2022).

[76] Charles P. Wright et al. “Extending ACID semantics to the file system”. In: *ACM Transactions on Storage* 3.2 (June 2007), 4–es. ISSN: 1553-3077. DOI: *10.1145/1242520.1242521*. URL: *http://doi.org/10.1145/1242520.1242521* (visited on 11/26/2022).

[77] Xiaojian Wu, Sheng Qiu, and A. L. Narasimha Reddy. “SCMFS: A File System for Storage Class Memory and its Extensions”. In: *ACM Transactions on Storage* 9.3 (Aug. 2013), 7:1–7:23. ISSN: 1553-3077. DOI: *10.1145/2501620.2501621*. URL: *http://doi.org/10.1145/2501620.2501621* (visited on 10/27/2022).

[78] Xinyang, Song and Sihang Liu. *Persistent Memory – A New Hope*. en-US. Sept. 2022. URL: *https://www.sigarch.org/persistent-memory-a-new-hope/* (visited on 11/30/2022).

[79] Jian Xu and Steven Swanson. “NOVA: A Log-structured File System for Hybrid {Volatile/Non-volatile} Main Memories”. en. In: FAST ’16. 2016, pp. 323–338. ISBN: 978-1-931971-28-7. URL: *https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu* (visited on 11/03/2022).

[80] Jian Yang. “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory”. In: *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST ’20)* (). URL: *https://www.usenix.org/system/files/fast20-yang.pdf*.

[81] Qing Yang and Jin Ren. “I-CASH: Intelligently Coupled Array of SSD and HDD”. In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. ISSN: 2378-203X. Feb. 2011, pp. 278–289. DOI: *10.1109/HPCA.2011.5749736*.

[82] Yang Yang et al. “SPMFS: A Scalable Persistent Memory File System on Optane Persistent Memory”. In: *50th International Conference on Parallel Processing*. ICPP 2021. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 1–10. ISBN: 978-1-4503-9068-2. DOI: *10.1145/3472456.3472503*. URL: *http://doi.org/10.1145/3472456.3472503* (visited on 11/01/2022).

[83] Yiying Zhang and Steven Swanson. “A study of application performance with non-volatile main memory”. In: *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. ISSN: 2160-1968. May 2015, pp. 1–10. DOI: *10.1109/MSST.2015.7208275*.