WineFS: a hugepage-aware file system for persistent memory that ages gracefully

Rohan Kadekodi
University of Texas at Austin

Saurabh Kadekodi
Carnegie Mellon University

Soujanya Ponnapalli
University of Texas at Austin

Harshad Shirwadkar
Google

Gregory R. Ganger
Carnegie Mellon University

Aasheesh Kolli
Google

Vijay Chidambaram
University of Texas at Austin
VMware Research

Abstract

Modern persistent-memory (PM) file systems perform well in benchmark settings, when the file system is freshly created and empty. But after being *aged* by usage, as will be the normal mode in practice, their memory-mapped performance degrades significantly. This paper shows that the cause is their inability to use 2MB hugepages to map files when aged, having to use 4KB pages instead and suffering many extra page faults and TLB misses as a result.

We introduce WineFS, a novel *hugepage-aware* PM file system that largely eliminates this effect. WineFS combines a new *alignment-aware* allocator with fragmentation-avoiding approaches to consistency and concurrency to preserve the ability to use hugepages. Experiments show that WineFS resists the effects of aging and outperforms state-of-the-art PM file systems in both aged and un-aged settings. For example, in an aged setup, the LMDB memory-mapped database obtains 2× higher write throughput on WineFS compared to NOVA, and 70% higher throughput compared to ext4-DAX. When reading a memory-mapped persistent radix tree, WineFS results in 56% lower median latency than NOVA.

CCS Concepts: • **Information systems → Storage class memory**; • **Hardware → Non-volatile memory**; • **Software and its engineering → File systems management**.

Keywords: Persistent Memory, File Systems, Hugepages, Fragmentation, Aging

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP '21, October 26–29, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483567

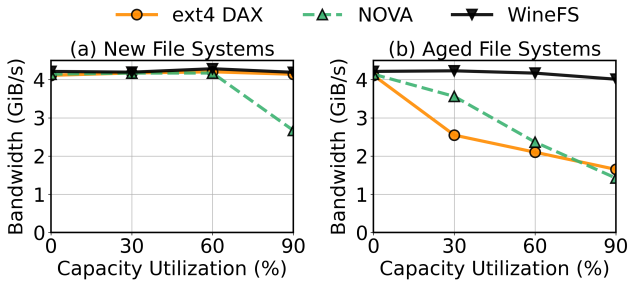

Figure 1. Impact of aging. Write bandwidth to memory-mapped files for three PM FSs on un-aged (left) and aged (right) file systems stored on Intel Optane PM. For ext4-DAX and NOVA, aging reduces bandwidth by ≈50% even when the FS is only 60% full. In contrast, WineFS maintains its high performance whether aged or not. Section §5.1 details the experimental setup, including the hardware, aging process on a 100GiB partition and Section §5.3 details the benchmark (sequential writes using `memcpy()`).

ACM Reference Format:
Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3477132.3483567

1 Introduction

Persistent memory (PM) refers to storage-class memory that offers non-volatility, low latency, and high bandwidth [6, 24, 51]. Researchers have developed new file systems to exploit unique characteristics of PM [16, 25, 28, 40, 44, 49, 50, 53] and shown excellent performance. Like most file system research, their evaluations benchmark on newly created file systems.

In reality, most file systems are *aged*, having undergone multiple allocations and de-allocations, and highly utilized. Companies try hard to keep file systems close-to-fully utilized [42] (*e.g.*, Google: "For disk-based storage, we want to keep disks full and busy to avoid excess inventory and wasted disk IOPs" [20]). Thus, it is important to ensure that

PM file systems have good performance in the aged and utilized setting.

Unfortunately, existing PM file systems provide much lower performance, when aged, for arguably the most performance-critical PM use case: PM-native applications that memory-map files and directly manipulate PM data [4, 5, 22, 23, 35, 39, 52]. Figure 1 illustrates this effect for two PM file systems: ext4-DAX [30] and NOVA [49]. When not aged (Fig. 1a), these file systems provide excellent performance, accommodating peak bandwidth. But, when aged (Fig. 1b), they lose a lot of their bandwidth—even when just 60% full, bandwidth drops by ≈50%.

Drilling down, we trace the difference to whether files are memory-mapped using 4KB base pages or 2MB hugepages. Using base pages results in 512× more page faults and TLB misses. With PM, the cost of handling a page fault (1–2 μs) is significantly higher than the cost of a 64 byte PM read or write (100-200 *ns*). If page faults happen in the critical path, they reduce performance significantly. Even if the pages are pre-faulted, the TLB misses significantly reduce performance, since they require walking the DRAM page table and caching page table entries in processor caches (thus reducing cache space for applications).

Hugepages are used whenever the memory-mapped files are allocated using aligned 2MB PM extents. When existing PM file systems are aged, file layouts and free space tend to be fragmented, preventing use of hugepages. Some sort of defragmentation tool is a tempting option, but is less realistic for PM file systems than disk-based systems—PM throughput means that aging will happen much more quickly in PM file systems, and non-idle-time de-fragmentation would cause significant performance degradation by competing for PM bandwidth with foreground application threads.

This paper presents WineFS, a novel *hugepage-aware* PM file system designed to ensure that hugepages can almost always be used for memory-mapped files, even when the FS is aged and mostly full. Doing so requires a holistic design that both avoids fragmentation-inducing algorithms and proactively considers hugepage boundaries during allocations. The success of WineFS is clear in Figure 1: aging causes minimal performance loss even when 90% full.

WineFS is designed, end-to-end, to avoid disruption of hugepage usage. Many inter-related aspects of a file system, from allocation policies to crash-consistency schemes to concurrency, affect its ability to keep using hugepages as it ages. WineFS uses a novel *alignment-aware* allocator that tries to preserve 2MB-aligned 2MB extents that can be mapped using hugepages. Large allocation requests are satisfied using aligned extents, while smaller allocations are satisfied using "holes". WineFS uses journaling for crash consistency, rather than the log-structuring popular in system-call-focused PM file systems [28, 49], to avoid data re-locations that disrupt its carefully planned layouts. For concurrency, WineFS uses a per-CPU journal rather than a per-file log (as used in NOVA), since we observed that a per-file log contributes to file-system fragmentation.

We evaluate WineFS on Intel Optane DC Persistent Memory [6] using a variety of micro-benchmarks, macro-benchmarks, and applications. We measure performance on both aged file systems (created using Geriatrix [26]) and newly created file systems. For applications like RocksDB [18], LMDB [2], and PmemKV [23], that access PM via memory-mapped files, WineFS outperforms NOVA on an aged setup by up-to 2× and ext4-DAX by up-to 70%. For applications like Filebench [46, 48], PostgreSQL [38], and WiredTiger [19], that access PM via POSIX system calls, WineFS roughly equals the performance of NOVA and ext4-DAX while providing the same guarantees, showing that WineFS does not have to trade system call performance for memory-map performance. Similar to NOVA, WineFS scales well with increasing number of threads. Across nine tested applications, WineFS consistently matches or outperforms existing PM file systems, whether aged or not.

In summary, this paper makes four main contributions:

- It demonstrates that existing PM file systems provide much lower performance for memory-mapped access when aged (i.e., not freshly formatted).
- It exposes that the root cause of the performance degradation is the failure of these PM file systems to maintain use of hugepages when aged. (§2)
- It introduces alignment-aware allocation as a key technique for PM file systems and shows, using WineFS, how it can be combined with fragmentation-avoiding consistency and concurrency mechanisms to achieve a PM file system that maintains its memory-mapped file access performance even when aged. (§3)
- It demonstrates with experiments on Intel persistent memory that it is possible to obtain good performance for applications using memory-mapped files, whether the file system is aged or not, without sacrificing performance for applications using system calls. (§5)

We have made WineFS publicly available at https://github.com/utsaslab/winefs.

2 Background and Motivation

We present background information on persistent memory (PM) and how it is accessed by applications. We describe the overheads involved in memory-mapping a file. We discuss the benefits provided by hugepages, even when all pages are pre-faulted. We examine simple solutions for obtaining hugepages for memory-mapped files, and why these solutions are not satisfactory. We conclude by motivating the need for a new PM file system that is hugepage-aware.

2.1 Persistent Memory

Persistent memory is a new memory technology from Intel [6, 24, 51]. It is similar to DRAM in many respects, being

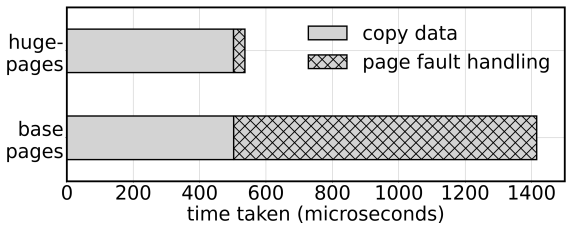

Figure 2. Memory-mapping overhead. This figure shows the time taken to memory-map and write a 2MB file, with and without hugepages. With hugepages, most of the time goes towards copying data. Without hugepages, two thirds of total time goes towards handling page faults and setting up page tables. Note that using hugepages makes writing the file 2× faster.

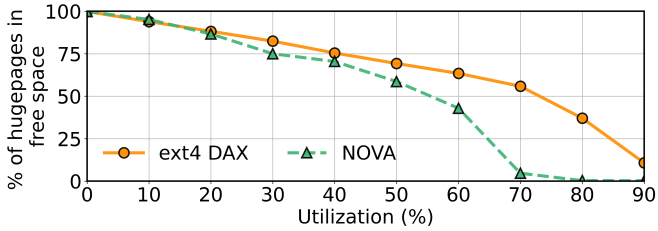

Figure 3. Free space fragmentation. Free-space becomes increasingly fragmented as utilization increases in aged NOVA and ext4-DAX. At 70% utilization, NOVA has close to zero 2MB aligned and contiguous regions.

byte-addressable and attached to the memory bus. It is accessed via processor load and store instructions, similar to DRAM. PM reads have 2–3× higher latency than DRAM, while writes have similar latency. PM read bandwidth is $1/3^{rd}$ that of DRAM, while write bandwidth is about 0.17× that of DRAM. Typical PM installations are expected to be around 6TB per server.

How PM is accessed. Traditional applications designed for magnetic hard drives and solid state drives can access PM through POSIX system calls such as `read()` and `write()`. This is not the most efficient way to access PM though, as the cost of trapping into the kernel and software layers like the Virtual File System (VFS) add significant overhead on the read and write path. A better way to access PM is to memory-map a PM file, and access PM directly using processor loads and stores. This method cuts out significant software inefficiencies and allows the application to read and write at close to device bandwidth. For example, writing sequentially to a 1GB file is 2× faster using memory-mapped files compared to system calls; the application spends 11× more time in the kernel when writing using system calls.

2.2 Performance overhead of memory-mapping

Given the performance benefits of memory-mapping, applications designed specifically for PM tend to use this method to access data (key-value stores such as PmemKV [23], Redis-pmem [5], RocksDB-pmem [4], caching services such as Memcached [35], Pelikan [52], databases such as Memhive-PostgreSQL [39], data-structure libraries such as PMDK [22]). However, memory-mapping a file comes with the overhead of setting up page table entries so that a virtual address in the process can point to a location on PM. This overhead directly depends upon whether hugepages are used to map the underlying file. We run an experiment where we memory-map and write to a 2MB file, with and without hugepages. Figure 2 shows the results: that mapping with hugepages can reduce the overall time taken by 2×, by reducing the time taken to handle page faults.

Conditions for obtaining hugepages. Despite the benefits of hugepages, it is challenging for applications using the memory-mapped access mode to reliably get hugepages. In order to get a hugepage on a page fault, the underlying file must be placed on 2MB aligned physical blocks and must not be fragmented. Even a single byte offset from alignment forces the operating system to fall back to base pages with a high page-fault cost.

2.3 Impact of aging on PM application performance

While it is relatively easier for a clean file system to place files on 2MB aligned and unfragmented regions on PM[1], it becomes increasingly difficult to maintain that alignment as the file system ages. File system instances are routinely used for several years at a time [8, 17, 36]. It is well known that as the file system ages, it suffers file and free space fragmentation because of the file creations, deletions and updates causing significant slowdowns [7, 13, 14, 26, 43]. In the context of PM and hugepages, arbitrary free space fragmentation worsens the problem of obtaining aligned and contiguous 2MB extents. Figure 3 shows the number of hugepages available as a file system is aged. We performed aging using Geriatrix [26], an aging framework to perform aging. In this experiment 100GB file system partitions of ext4-DAX and NOVA were subjected to up to 40TB of file creates and deletes. With increasing utilization both ext4-DAX and NOVA are unable to maintain aligned 2MB extents. In fact at about 70% utilization, NOVA had close to zero 2MB extents left.

We observed that fragmentation of free space due to aging does not impact the performance of applications that access PM through POSIX system calls such as `read()` and `write()`, as PM offers similar bandwidth for sequential and random access of data.

In summary, the performance of applications accessing PM by mmap-ing files is up-to 2× better than applications accessing PM through POSIX system calls but can degrade significantly with age due to fragmentation of free space, while the performance of applications accessing via POSIX system calls remains constant and independent of age.

[1] PMFS and xfs-DAX cannot get hugepages even when they are clean because unlike ext4-DAX, these file systems completely disregard alignment even for large extents.

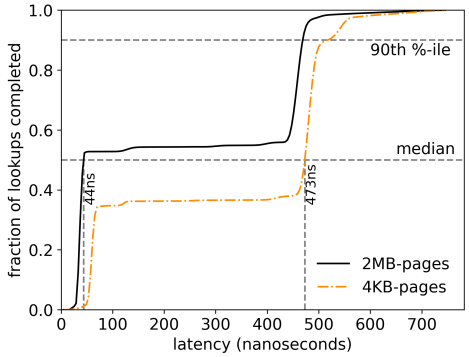

Figure 4. Overhead due to TLB misses. This figure shows the CDF of latencies when reading random elements of a large PM array that has been memory-mapped and pre-faulted. Using hugepages reduces the number of TLB misses. On a TLB miss, page table entries are fetched and cached in the processor caches, reducing cache space for the application. The median latency is 10× higher when using base pages rather than hugepages, as the array element that is read has been knocked out of the processor cache by page table entries.

2.4 Pre-faulting pages

A natural question that arises is: can we simply pre-fault all of the pages outside the critical path. First, this is simply not possible for a number of applications (such as LMDB [2]) as they use sparse mappings, allocating space on demand when they get a page fault. Pre-faulting would lead to unacceptable space overhead for these applications. Second, even if all the pages are pre-faulted, hugepages still provide a performance benefit by reducing TLB misses. We ran an experiment where we memory-mapped a file containing a large array and randomly read elements in the array. The entire file was pre-faulted, so there were no page faults in the critical path. Figure 4 shows the results. There is a 10× reduction in median latency when using hugepages, corresponding to whether the array element that was read was in the processor cache or not. When using base pages, the array element was kicked out to make space for page table entries when handling TLB misses. Thus, even when all pages are pre-faulted, using hugepages to map files on PM improves performance.

2.5 Hugepages without file system support

One solution to obtain hugepages is via defragmentation. In this context, defragmentation would mean re-alignment of extents to 2MB boundaries, and not necessarily focusing on its contiguity. One can imagine a user-space utility for defragmenting memory-mapped files, which would read the fragmented file, and rewrite it using large allocations. However, without file system support, it is impossible to guarantee that large allocation requests are satisfied using 2MB aligned extents. We observe that ext4-DAX and NOVA do not always use aligned extents when they are available; this is natural since these file systems optimize for locality and contiguity, rather than hugepages. For example, in Figure 1 (b), ext4-DAX has 12k aligned extents available at 60% utilization, but ends up using only 3k aligned extents, while the workload requires 8k aligned extents.

One could also imagine a file-system-wide defragmentation utility could be run to reclaim hugepages. However, existing defragmentation utilities do not aim to recover hugepages. Such utilities would consume PM bandwidth when running in the background. The performance for a given PM file would only improve if it had been defragmented by the utility; this could take a lot of time depending on the size of the file system.

Finally, one could use a file system that always allocates in sizes of 2MB; the `bigalloc` mode of ext4 does this. However, this leads to significant space wastage and internal fragmentation, as a large number of user files tend to be small.

2.6 The need for a hugepage-aware PM file system

We believe that it is a better approach to be proactive about conserving hugepages, rather than reactively defragmenting files. We require support from the file system to conserve aligned extents. Current PM file systems do not optimize for this goal. Some file systems such as Strata [28] and NOVA [49] make it harder to map files using hugepages due to their log-structured nature. NOVA could be modified to become hugepage-aware, but would require non-trivial changes to its design which would reduce its performance. For example, dedicating on-PM regions for the per-file journals would increase the load on garbage collection and its interference with foreground threads. Changing the copy-on-write granularity of NOVA to the size of hugepages to avoid fragmentation would result in increased write as well as space amplification.

Mature file systems such as ext4-DAX [30] or xfs-DAX [3, 21] have allocators that care more about contiguity than alignment, which makes them sacrifice hugepages as part of their design. Additionally, in order to achieve high performance for a wide range of legacy as well as newer PM applications, the mature file systems would have to change several fundamental components such as incorporating a hugepage-aware allocator, devising a conducive on-PM layout, supporting low-cost data atomicity and adding fine-grained low-cost journaling for crash consistency.

Designing a hugepage-aware PM file system requires revisiting all aspects of file system design from the lens of hugepage-awareness, rather than tweaking one or two aspects of an existing file system.

3 WineFS

We present WineFS, a hugepage-aware PM file system. We present its goals, and an overview of how the system achieves these goals. We then describe the design choices that lead

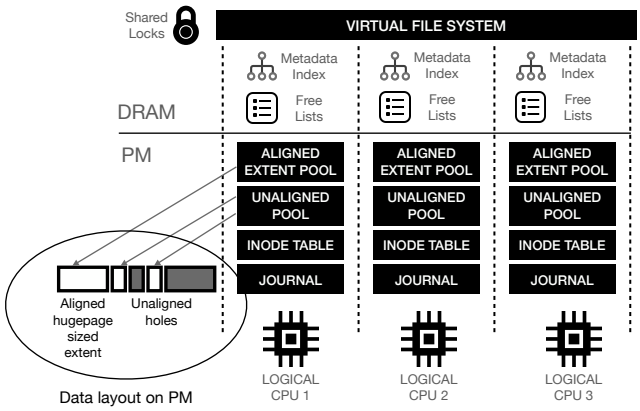

Figure 5. WineFS Architecture. The figure shows the main components of WineFS. WineFS partitions the file system per logical CPU for concurrency. Each logical CPU has its own journal, inode table, and free lists for aligned extents and holes. WineFS uses DRAM indexes for metadata for efficient directory and resource lookups. The shared locks in the VFS layer help WineFS coordinate its multiple journals.

to hugepage-awareness, and the choices that allow good performance for applications using POSIX system calls to access PM.

3.1 Goals

- WineFS should be POSIX-compliant, and must not require any changes in the application.
- WineFS must try to provide hugepage-sized extents aligned at the hugepage boundary for files that are memory-mapped.
- WineFS must not sacrifice performance of applications that use POSIX system calls to access PM.
- WineFS must not sacrifice performance when the file system is new, either for memory-mapped or system-call access to PM.
- The design of WineFS must seek to preserve hugepages wherever possible.
- WineFS must provide strong guarantees such as atomic, synchronous operations.

3.2 Overview

WineFS achieves these goals through two sets of design choices. First, WineFS achieves hugepage-awareness through:

- A novel *alignment-aware* allocator that satisfies large allocation requests using aligned extents, and smaller requests using unaligned holes.
- Using a PM data layout with *contained fragmentation*: metadata structures and journals that are updated in-place.
- Using journaling for crash consistency as it preserves data layout, even at the cost of writing metadata twice to PM.
- Using per-CPU metadata structures (rather than per-file) for obtaining concurrent updates.
- Using data journaling to atomically update files with aligned extents, and using copy-on-write to atomically update holes.

WineFS ensures good performance for applications that access PM via POSIX system calls through a *second* set of design choices. The key observation is this second list is chosen such that it composes well with first list. For example, this would not be the case if WineFS had used log-structuring for accelerating metadata updates. The design choices:

- WineFS uses fine-grained journaling optimized for PM.
- WineFS uses DRAM metadata indexes to accelerate operations such as directory lookups.

Figure 5 presents an overview of the WineFS architecture and how all these design choices come together.

3.3 Guarantees

WineFS can be run in two modes: strict mode and relaxed mode. The default is the strict mode. The mode can be changed using mount options.

Strict Mode. All file system operations, both data operations and metadata operations, are atomic and synchronous. Upon completion of each `write()` system call, the data involved is guaranteed to be durable. NOVA, SplitFS-strict, and Strata provide the same guarantees.

Relaxed Mode. All metadata operations such as `rename()` are atomic and synchronous. Data operations are not atomic, and may be partially completed on a crash. ext4-DAX, xfs-DAX, and PMFS provide the same guarantees.

3.4 hugepage Awareness

We now describe in detail the design choices that make WineFS a hugepage-aware PM file system.

Data Layout: Controlled Fragmentation. WineFS uses *controlled fragmentation* in designing its data layout on PM. Typically, metadata structures cause significant fragmentation as they tend to be small. For example, NOVA has a per-file log that causes fragmentation, using up an aligned extent. WineFS tries to control the fragmentation caused by metadata structures, by assigning *dedicated* locations for metadata structures. The metadata structures are updated in-place within these locations. The space in these locations is recycled for other metadata structures.

Concurrency: Per-CPU data pool and metadata structures. Closely related to the data layout is how WineFS achieves concurrency. NOVA achieves high concurrency by providing a per-file log. While this provides high concurrency, it also fragments the free space and uses up aligned extents. WineFS chooses a different design: per-CPU journal, data structures, and data and metadata pools. The file system is partitioned between logical CPUs. Each logical CPU gets its own journal, inode table, and pool of aligned extents and unaligned holes. Each CPU maintains free lists in DRAM

that keeps track of free inodes and extents in its own pool of inodes and extents. In the common case, an allocation request arising at a logical CPU can be handled locally, without any communication with other logical CPUs. Since the unit of parallelism is a logical CPU, rather than a file, this design provides high concurrency without incurring fragmentation. Experimental results show that WineFS scales as well as NOVA, while being more hugepage-friendly.

A natural question that arises is: how are the per-CPU journals coordinated? WineFS uses the Virtual File System (VFS) layer for coordination. The namespace is shared across all logical CPUs, and VFS provides shared locks for directory inodes, while WineFS holds locks for file modification operations such as `write()` or `fallocate()`. An inode can only be locked by one logical CPU at a time. WineFS ensures that all file system operations that require journaling also grab inode locks, implying that a file can only be part of one per-CPU journal at a given time. Different files can be locked by different CPUs and journaled concurrently.

Allocation: Alignment-Aware Allocation. WineFS uses a novel *alignment-aware* allocator. The allocator splits the entire partition into aligned hugepage-sized extents. Incoming allocation requests are split into requests of hugepage sizes or below. Large allocation requests (that are hugepage-sized) are satisfied using an aligned extent. Small allocation requests, less than the size of a hugepage, are satisfied by smaller, unaligned extents. If required, a single aligned extent is broken up to satisfy small allocation requests.

The allocator uses the following policy to decide which extent to utilize for an allocation request. It always tries to satisfy the request locally, in the data pool of the same logical CPU where the request was made. If the free-space of the local data pool is over, it picks an extent from the data pool of a different logical CPU to satisfy allocation requests, which is based on the size of allocation request. It chooses an aligned extent from the data pool of the logical CPU with the most free aligned extents to satisfy a large (hugepage-sized) allocation request. It chooses an unaligned extent from the data pool of the logical CPU with the most free unaligned extents to satisfy a small allocation request. When the allocated extent is freed, it is inserted back into the free-space of the original data pool from where it was allocated. It is then merged with other contiguous free extents in the data pool and converted to a free aligned extent.

Crash Consistency: Journaling. WineFS chooses to use journaling for updating file system metadata in an atomic fashion. There is a fundamental trade-off between using journaling and copy-on-write or log-structuring. Journaling results in writing metadata twice, once to the journal and once in-place. However, journaling preserves the data layout. In contrast, copy-on-write or log-structuring require only a single write to PM for metadata. However, copy-on-write will write metadata throughout the partition, varying its location. The single write nature of log-structuring is attractive, which is why it has been adopted in NOVA and Strata. However, we believe that trading off an extra write for preserving the data layout is the right trade-off given how small metadata is, and how important hugepages are for performance. In this respect, WineFS makes a fundamentally different decision than NOVA and Strata.

Data Atomicity: Hybrid Techniques. WineFS provides atomic data updates by default. WineFS uses different techniques to update a file atomically, depending upon how the file extents are allocated. WineFS uses data journaling to update aligned extents, preserving their data layout. As a result, atomically updating a file will not cause it to lose its hugepages. WineFS uses copy-on-write to update unaligned extents, with the extents being written to new unaligned holes provided by the alignment-aware allocator. In this manner, WineFS strikes a balance between incurring the extra write for preserving data layout (when it matters), and using copy-on-write when preserving the data layout does not matter.

3.5 Ensuring good performance for applications using POSIX system calls

The design decisions described so far will preserve hugepages and help obtain good performance for applications using memory-mapped files. WineFS also seeks to obtain good performance for applications using POSIX system calls to access PM. The techniques WineFS uses to achieve this are well-known. The challenge lies in adopting techniques that compose well with the hugepage-aware design decisions, such that good performance is obtained for applications using either memory-mapped files or system calls.

Fine-grained journaling. Similar to other PM file systems such as PMFS and SplitFS, WineFS optimizes journaling for PM. WineFS uses a per-CPU, fine-grained, undo journal. Each log entry is only a cache line in size. All metadata operations in WineFS are synchronous, so the journal entries are immediately persisted. Since metadata operations are synchronous, reclaiming journal space can be done immediately once the operation completes.

WineFS uses undo journaling instead of the redo journaling used in systems like ext4-DAX or SplitFS. In undo journaling, the old data is first copied to the journal, and then the new data is updated in place. If there is a crash, the data is rolled back to the old version using the journal. While undo journaling and redo journaling are functionally equivalent, their performance characteristics differ. Redo journaling has lower latency for writing transactions (no lock to write to the journal), but higher latency for updating data in place (need to get a global lock or set of locks). In contrast, undo journaling has higher latency for writing transactions (need to get locks to update data in place), but does not incur any delay once the transaction is committed; the log entries can

be discarded. WineFS employs undo journaling as it reduces tail latency and provides more deterministic performance.

DRAM indexes. WineFS uses red-black trees for traversing directory entries and for maintaining inode free-lists in the per-CPU allocation group, similar to NOVA. The DRAM indexes help in fast metadata operations, as opposed to PMFS that does sequential scanning of directory entries and inode free-lists causing significant slowdowns.

3.6 Implementation

WineFS is implemented based on the PMFS code base (6K LOC), and implemented in Linux kernel 5.1. We choose PMFS to build on, as PMFS is a journaling file system and has the on-disk layout that helps WineFS achieve all its goals. It is totally ≈10K LOC. The following optimizations have been added to the PMFS codebase: (a) PM-optimized per-CPU journaling: 1K LOC, (b) alignment-aware allocator and hugepage handling on page faults: 1K LOC, (c) auxiliary metadata indexes: 700 LOC, (d) NUMA-awareness: 300 LOC, (e) Crash recovery: 1K LOC, and (f) hybrid data atomicity mechanism: 500 LOC. We describe some of the additional implementation details below.

Alignment-aware Allocation. The allocator uses two pools to help with allocation. One is a pool of free aligned extents, and the other is a pool of free unaligned extents. These pools are written to PM on unmount. On a crash, they are re-initialized by scanning the set of used inodes in the file system (similar to NOVA).

WineFS breaks down each allocation request internally into a series of smaller allocation requests, which are smaller than or equal to the size of a hugepage. The hugepage-sized requests are satisfied using the pool of free aligned extents. The smaller requests are satisfied using the pool of free unaligned extents.

Aligned extent pool. WineFS maintains a linked list of free aligned extents in each logical CPU. On getting a hugepage-sized allocation request, WineFS removes an extent from the head of the linked list and uses the extent for satisfying the allocation request. Whenever a free aligned extent is deleted, it is added to the tail of the linked list of the corresponding logical CPU.

Unaligned extent pool. WineFS re-uses the implementation of red-black trees in the linux kernel to keep track of free unaligned extents in each logical CPU. The red-black tree is keyed based on block offsets of the free extents. WineFS uses a first-fit approach to allocate an unaligned extent for small allocation requests. Whenever an unaligned extent is deleted, the allocator tries to merge it with its nearby extents. If the extents can be merged into an aligned extent, it is merged and tracked in the aligned extent pool.

Journaling. Each thread in WineFS starts a transaction in its per-CPU journal. Once a transaction is started at a per-CPU journal, it continues there even if the thread is migrated away. Each journal transaction contains transaction-entries 64B in size, along with a start and commit entry to mark the start and end of the transaction.

Transaction entries. Each transaction entry contains the following metadata persisted on PM:

- shared transaction ID: This is an atomic counter shared by the per-CPU journals, which increments on every transaction create. As a result, a transaction ID is unique across all the per-CPU journals.
- per-CPU wraparound-counter: Each per-CPU journal contains a wraparound counter, incremented every time that the journal is wrapped around.
- transaction entry type: This provides information about the type of log entry, it is either START, COMMIT or DATA. The START and COMMIT entries are used to mark the start and end of a journal transaction, while the DATA entry is used to store system-call specific entries.

Reclaiming journal space. All operations in WineFS are immediately durable, allowing WineFS to reclaim the space in per-CPU journals that is occupied by the committed transactions. Every journal transaction reserves the maximum number of log entries that it requires in the per-CPU journal before starting the transaction. Across all system calls, the maximum number of log-entries required are 10, occupying 640 bytes in the journal. If there is not enough space in the journal, the thread waits till enough space is reclaimed before starting the journal transaction.

Handling concurrent updates to shared files. When multiple threads try to modify the same directory by creating files in a shared directory, the VFS locks the directory inode, and only one of the threads is allowed to proceed at any given time. This VFS locking for all the shared metadata updates ensures that there is only a single uncommitted transaction for any file/directory on a crash.

Handling thread migrations. WineFS creates a journal transaction in its respective per-CPU journal. If the OS scheduler migrates the thread to another CPU after creating a journal transaction, WineFS still ensures that the migrated thread uses the per-CPU journal in which the transaction was created, for the duration of the transaction.

Journal Recovery. During recovery, WineFS has to recover multiple per-CPU journals. Note that transaction IDs are global across the per-CPU journals. WineFS rolls-back journal entries across per-CPU journals based on the transaction ID order. The per-CPU wraparound-counter helps WineFS in identifying the valid journal entries during recovery. WineFS rolls-back all the transactions that contain the START log entry but that don't have the COMMIT log entry. WineFS ignores all committed transactions.

Minimizing remote NUMA accesses Given that PM will be deployed on multiple NUMA nodes, it is important that the PM file system try to minimize remote NUMA accesses. WineFS uses a number of mechanisms to try to reduce remote NUMA accesses.

The NUMA-awareness strategy of WineFS builds on the insight that remote writes are more expensive than remote reads [10, 51]. Thanks to temporal locality, if writes are routed to the local NUMA node, reads of the newly written data in the near future will also be local. We recognize that it is challenging to prevent all remote accesses, and thus focus on minimizing remote writes.

Determining the home NUMA node for a process. WineFS assigns a *home* NUMA node to each process when the process first creates or writes a file. The assigned home NUMA node is the NUMA node with most free space.

Writes. On each write, WineFS checks if the process is in its home NUMA node. If required, the process is migrated to its home NUMA node, and space is allocated from one of the per-thread allocation groups on that NUMA node. Further allocations and writes continue at the home NUMA node. If the home NUMA node runs out of free space, a new home is selected, and the process is migrated.

Reads. All reads to recently written data will be local since WineFS ensures the writes happen on the home NUMA node. Older reads will be remote; WineFS does not migrate the process to prevent this situation. Thread migrations are expensive, and the process may access data spread out over different NUMA nodes causing it to thrash if it is migrated too often. Instead, WineFS focuses on keeping writes local.

Child process. Children of a process inherit its home NUMA node, under the assumption that they will be accessing data written by the parent process.

Crash Recovery and unmount. On a clean unmount, the data structures maintained in DRAM (*e.g.,* alignment-aware allocator's free list, inode free list) are serialized and stored on PM. On mount, these data structures are deserialized and reconstructed in memory. If there is a crash, WineFS is first recovered to a consistent state using the per-CPU journals as explained above.

Reactively rewriting a file. If WineFS finds on memory-mapping a file that it is fragmented, it adds it to a list to be rewritten. A background thread in WineFS later reads the file and rewrites it using big allocations. A journal transaction is used to atomically delete the old file and point the directory entry to the new file. This situation may arise if an application uses small allocations when writing to a file that will be later memory-mapped. Due to the small allocation requests, WineFS would have allocated unaligned holes to the file. Note that reactive rewriting of files is an extremely rare operation. Applications that use the memory-mapped interface usually perform occasional large allocations in order to avoid frequently trapping into the kernel.

Supporting extended attributes for preserving alignment of files. Once WineFS provides aligned extents to a file, it makes this information persistent by using a special extended attribute. This is useful if a file allocated using aligned extents is later copied over to another partition or file system by a utility such as `rsync` and `cp`. Ideally, we would want the file to retain aligned extents after the move or copy. Many linux utilities such as `rsync` and `cp` will read and copy extended attributes associated with files. WineFS uses the extended attributes to communicate alignment information of files from one WineFS partition to another (on the same or different servers) no matter how that file is transferred. For example, if an aligned file is transferred from a WineFS partition on server A to a WineFS parition on server B via `rsync`, the receiving partition will allocate aligned extents (and not holes) to the file by referring to its extended attributes, even though `rsync` typically copies data using small allocations. Moreover, WineFS also supports directory level extended attributes where all files directly within a directory (not its subdirectories) will inherit alignment information from the extended attributes of the parent directory.

4 Discussion

Preserving layout is more important than saving extra writes. File systems such as NOVA and Strata that use the log-structured approach write metadata only once to the file system. However, they change the location of metadata frequently as it is updated. We find that preserving the layout of files and reducing fragmentation is crucial to obtaining hugepages in PM file systems. However, preserving the layout using journaling comes at the cost of writing metadata twice. Given that persistent memory currently has a long lifetime (a 256 GB Intel PM module can withstand 350 PB of writes [41]), we believe that the benefits of hugepages are worth the extra write.

Proactive approach is required to maintain hugepages. WineFS shows that by designing the file system to be hugepage-aware, it is possible to preserve hugepages in the face of aging and high utilization. We believe this is the right approach, as it can be implemented at modest additional complexity without sacrificing performance for applications using system calls to access PM. In contrast, reactive approaches like defragmentation provide only temporary relief before the file system becomes fragmented again. The defragmentation utility would need to be run at high frequency to provide benefits equivalent to WineFS. As with any background maintenance task, defragmentation requires IO and steals device bandwidth from the foreground process. We ran an experiment where we read a fragmented 5GB file and rewrote it with aligned extents. In parallel, we also ran a foreground

workload that performed memory-mapped reads on another file. We observed a slowdown of 25–40% when the defragmentation is going on.

Thoughts on adding hugepage-friendliness to existing file systems. We initially tried to add hugepage preservation in ext4-DAX by changing the multi-block allocator to provide 2MB aligned extents for large allocations. To accelerate applications using the system-call access mode, we changed the journaling mechanism of ext4-DAX to perform fine-grained journaling instead of relying on the JBD2 journal. With our changes, ext4-DAX managed to get hugepages reliably in a clean setup for memory-mapped files. However, the allocator spent a significant amount of time in searching for available aligned extents, degrading performance when aged, compared to the original ext4-DAX. With respect to applications using system calls to access PM, the performance increased due to fine-grained journaling, but still suffered overheads such as ensuring consistency between on-disk versions of DRAM indexes.

NOVA uses log-structured layout for its metadata, and contains a per-inode log in the form of a linked list. Although NOVA tries to allocate aligned extents to large files, it is incapable of preserving hugepages due to extensive free-space fragmentation, as shown in Figure 3. NOVA would need to employ frequent (and expensive) garbage collection to retain free-space contiguity and alignment, interfering with foreground application performance.

Our experience shows that hugepage-awareness is an overarching concern and not something that can be easily added to an existing PM file system. When designing WineFS, we had to incorporate hugepage-awareness in multiple core components of the file system.

Supporting different hugepage sizes. The size of a hugepage is not fundamental to the design of WineFS. We used 2MB hugepages in this work since our test machine had only 2MB hugepages available. While WineFS currently has one allocator for 2MB hugepages, it can have additional allocators for each hugepage size that can be supported. For example, since modern kernels and devices support 1GB hugepages, WineFS could have two alignment-aware allocators, one for each hugepage size and one hole-filling allocator.

Using different aging profiles. Throughout the paper, we use the Agrawal aging profile to age all file systems. The Agrawal profile contains a mix of large (>= 2MB) and small files (< 2MB). We also experimented with other profiles, and saw that in some cases, the fragmentation of other file systems is significantly worse compared to the fragmentation seen by the Agrawal profile. For example, in another profile that mimics an HPC environment [47], we see that even with 50% utilization, only 28% of the free-space is aligned and unfragmented in ext4-DAX, while more than 90% of th free-space is aligned and unfragmented in WineFS. Depending upon how the file system is aged, the user might experience more severe performance degradation than what we show in this work.

5 Evaluation

We seek to answer the following questions:

- Does WineFS handle crashes and metadata updates correctly? (§5.2)
- What is the read / write throughput of WineFS in an aged setting? (§5.3)
- What is WineFS performance for applications accessing PM through memory-mapped files in an aged setting? (§5.4)
- What is WineFS performance for benchmarks and applications using system calls to access PM? (§5.5)
- Is WineFS scalable? (§5.6)

5.1 Experimental setup

We use a two-socket machine with 28 cores, 112 threads, and 500GB of Intel Optane DC Persistent Memory module, with Fedora-30 and Linux 5.1 kernel. We use a single socket on this machine for our evaluation and disable NUMA awareness in WineFS, because some other file systems such as NOVA and PMFS are not able to run on multiple NUMA nodes, while the other file systems do not support NUMA affinity when run on multiple NUMA nodes. We compare WineFS with two groups of PM file systems. First, with file systems providing metadata consistency: Ext4-DAX [34], xfs-DAX [45], PMFS [40], NOVA-relaxed [49], and SplitFS [25]. WineFS in Relaxed mode provides metadata consistency. Second, with file systems providing both data and metadata consistency:NOVA and Strata [28]. WineFS by default (in strict mode) provides both data and metadata consistency.

FS aging setup. To reflect PM file systems in the real world, we use the Geriatrix [26] tool to age evaluated file systems. Agrawal et al. [7] is one of the widely cited profiles that are used to measure the performance of aged file systems. We use the Agrawal profile to represent all the file systems aged by 165TB of write activity in a 500GB partition caused by creation and deletion of files, with a mix of small (< 2MB) and large (>= 2MB) files. 56% of the total capacity is occupied by large files while the rest is occupied by small files. We use other profiles as well (e.g., Wang et al. [47]) to avoid overfitting our observations to the Agrawal profile. We default to using the popular aging profiles since there is not much data published on the aging statistics of PM file systems and their workloads.

We also increase the utilization of the file system to 75%. We measure the end-to-end performance applications along with the number of page faults and cache misses they incur. We compare with PMFS in a clean FS setup, as PMFS takes weeks to age under 165 TB of metadata operations. Thus, the PMFS results provide an upper bound on PMFS performance when aged.

5.2 Crash Consistency & POSIX Compliance

Crash consistency. We use a modified form of the CrashMonkey framework [37] to test whether WineFS recovers correctly from crashes. We use the Automatic Crash Explorer (ACE) to generate workloads with system calls that modify file-system metadata. For each workload, we use CrashMonkey to generate crash states corresponding to all possible re-orderings of in-flight writes inside each system call. The number of in-flight writes inside each system call were low, so CrashMonkey was able to exhaustively test crash states. Finally, we check that WineFS had always recovered to a consistent state. This exercise was useful in finding minor bugs in WineFS early during its development. Currently, WineFS passes all the CrashMonkey tests.

As WineFS uses per-CPU journaling, we also check if WineFS is crash-consistent for multi-threaded applications. Note that WineFS shares a single namespace across all its CPUs. It uses the VFS locks to ensure that only one journal transaction (across all CPUs) involves each file. As a result, after a crash, WineFS has at most one per-CPU journal with pending updates for a given file or directory. WineFS recovers multiple per-CPU journals by using the global transaction ID to order different journal entries.

Time to recover. On recovery, WineFS must reconstruct the DRAM data structures such as the alignment-aware free-space allocator and per-CPU inode inuse lists using relevant metadata on PM. WineFS scans the per-CPU inode tables in parallel. Note that the recovery time depends on the number of files, and not the total amount of data in the file system.

By inducing a crash in WineFS with 675GB of data, WineFS recovered in 7.8s. In this experiment, there were 3.5M files in the partition that was recovered.

POSIX compliance. We use Linux POSIX file system test suite [27] to test if WineFS meets POSIX standards. WineFS passes all the tests. This is important because it means applications will obtain the expected POSIX behavior from WineFS without the need for application modifications.

5.3 Read and Write Throughput

We analyze the throughput of WineFS with micro-benchmarks capturing sequential/random read/write workloads. We age the file systems as described in the experimental setup and then run experiments.

Performance for memory-mapped access. We memory-map a 50GB file and use `memcpy()` to perform reads and writes in sequential and random order. WineFS has the highest throughput across all aged PM file systems: WineFS outperforms NOVA by **2.6**× on sequential and random writes, and by **2.3**× on sequential reads, and by **2.7**× on random reads, as shown in the Figure-6(a). WineFS spends about 3% of the total time on handling page faults while NOVA spends 60% of the time handling faults. WineFS has comparable performance with the best performing PM file systems in a clean, unaged setting since all file systems are able to map files using hugepages.

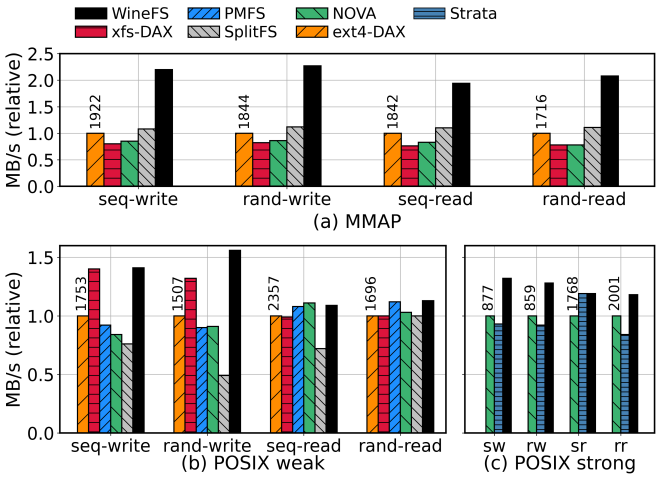

Figure 6. Read and write throughput for system calls and memory-mapped access. This figure shows the throughput of sequential and random reads and writes in different file systems. There is a `fsync()` after every 10 operations. POSIX strong indicates data consistency while POSIX weak indicates metadata consistency. ext4-DAX and xfs-DAX suffer from high overheads in appends due to costly `fsync()`. Across all these workloads, WineFS matches or betters the performance of the best PM file system. Good performance on memory-mapped workloads is due to hugepage-awareness, while good performance on system-call workloads is due to fine-grained journaling and DRAM indexing.

Performance for system-call access. We start with an empty file and append data at 4KB granularity until it fills 50% of the free space. We perform reads and in-place writes at 4KB granularities in sequential and random order. Overall, WineFS has equal or better throughput compared to other file systems on reads and writes, as shown in Figures 6(b), 6(c). On writes, WineFS outperforms NOVA by up-to 25%, as NOVA has to add new log entries, invalidate older entries, and update its DRAM indexes for handling overwrites. Further, WineFS and NOVA perform better than Strata on writes as Strata has to perform expensive data copies from its per-process logs to the shared PM region for making data visible to other processes.

Summary. These results show that WineFS achieves excellent read and write throughput, regardless of whether memory-mapped files or system calls are used. It validates that the design choices made to increase memory-mapped and system-call access modes work well together.

5.4 Performance for memory-mapped access mode

We evaluate WineFS using data stores like RocksDB, LMDB, and PmemKV, and persistent data structures like the adaptive radix tree (P-ART). The configurations of these applications are shown in Table 1. We continue to use the aged file system setting.

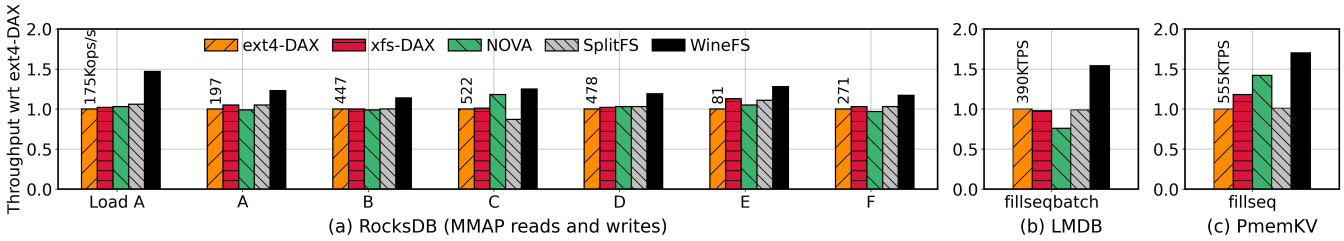

Figure 7. Performance on aged file systems. This figure shows the performance of different file systems when aged to 75% using Geriatrix [26]. Overall, WineFS outperforms all other file systems by up-to 70% compared to ext4-DAX in PmemKV, and up-to 2× compared to NOVA on LMDB. The file systems with metadata consistency guarantees are shown in (a), (b) and (c) whereas the file systems with data and metadata consistency guarantees are shown in (d), (e), (f). Note: we do not compare with PMFS as it was unable to age successfully.

Application & Workload	*Description*
Evaluating memory-mapped access mode	
YCSB [15] on RocksDB [4]	Data retreival & maintenance
LMDB [2] fillseqbatch	Memory-mapped database
PmemKV [23] fillseq	PMDK key-value store
P-ART [29] lookup	PM data structure
Evaluating system-call access mode	
Varmail [46]	16 threads, 1M files
Fileserver [46]	50 threads, 500K files
Webserver [46]	100 threads, 500K files
Webproxy [46]	100 threads, 1M files
PostgreSQL [38] pgbench rw	32 threads, 60GB database
WiredTiger [19] fillrandom	MongoDB's default engine
WiredTiger [19] readrandom	MongoDB's default engine

Table 1. Applications used in evaluation. Macrobenchmarks and real-world applications used to evaluate PM file systems.

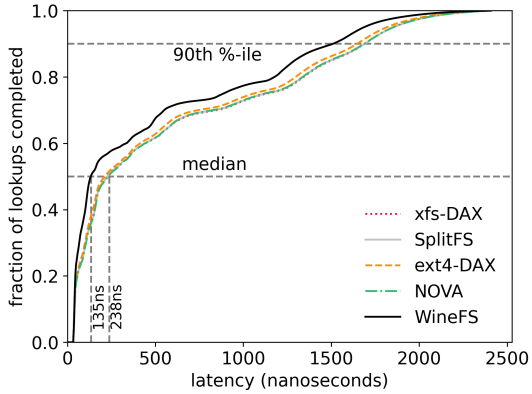

Figure 8. Latency distribution for P-ART lookups. The figure shows the latency distribution for lookups on the P-ART persistent radix tree. The tree is memory-mapped and pre-faulted before the lookups. WineFS results in 56% lower median latency compared to the other PM file systems as WineFS leads to lower TLB misses and LLC cache misses.

YCSB on RocksDB. We run RocksDB configured to use memory-mapped reads and writes, with hugepages enabled and a memory cap of 64GB. We run the industry-standard YCSB workloads on RocksDB with 60GB dataset consisting of 50M keys and operations. We report RocksDB throughput on all file systems in the Figure 7(a). WineFS provides the best throughput, outperforming ext4-DAX and NOVA by up-to **50%** on average across all YCSB workloads. RocksDB incurs the least page faults on WineFS. On other PM file systems, RocksDB incurs up-to 56× higher number of page faults, as shown in the Table 2.

LMDB. We run LMDB [2], a btree-based memory-mapped database, with db_bench benchmark's `fillseqbatch` workload with 50M keys. This workload batches and writes 1KB sized key-value pairs sequentially, which according to LMDB is its best-performing workload [31]. LMDB does on-demand allocations and zero-outs pages on page faults by using `ftruncate()` instead of `fallocate()` for the allocations. This reduces space-amplification, but leads to costly page faults. WineFS outperforms ext4-DAX by **54%** and NOVA by **2**×, as shown in Figure 7(b). LMDB running onWineFS incurs 200× and 250× lower page faults in comparison to ext4-DAX and NOVA, as shown in the Table 2.

PMemKV. We run PMemKV [23], a key-value store from Intel that uses 128MB memory-mapped files for storing data on PM. We configure PMemKV's `cmap` concurrent engine to run with 16 threads. We run the write-only `fillseq` workload that sequentially inserts keys with 4KB-sized values. In this workload, PmemKV creates a PM pool using `fallocate()`, and keeps extending the pool as it gets used up by creating more files and allocating them via `fallocate()`. PMemKV gets the best performance on WineFS, which is **20%** higher than on NOVA, **70%** higher than on ext4-DAX, and **45%** higher than xfs-DAX, as shown in the Figure 7(c). The number of page faults incurred by PMemKV on all PM file systems are shown in Table 2. NOVA does the allocations and zero-out of pages on `fallocate()` while the page fault routine only sets up page tables. On the other hand, ext4-DAX does zero-out of pages on a page fault and not `fallocate()`, making page faults more expensive in ext4-DAX. As a result, the performance of NOVA is better than ext4-DAX even though NOVA suffers from higher number of page faults compared to ext4-DAX.

	YCSB							LMDB	PmemKV
	Load A	A	B	C	D	E	F	fillseqbatch	fillseq
WineFS	**1.59 Mn**	**3.83 Mn**	**3.83 Mn**	**1.34 Mn**	**1.36 Mn**	**0.48 Mn**	**4.85 Mn**	**0.06 Mn**	**0.01 Mn**
ext4-DAX	11.85×	17.86×	6.20×	10.60×	18.36×	45.38×	14.57×	205×	292×
xfs-DAX	28.26×	23.24×	7.04×	11.21×	20.27×	56.38×	17.70×	280×	455×
SplitFS	16.46×	20.52×	6.73×	10.93×	19.94×	50.58×	16.15×	208×	296×
NOVA	32.03×	1.57×	7.65×	1.05×	23.23×	1.15×	22.30×	261×	399×

Table 2. Page faults. This table shows the number of page faults incurred by different applications on aged file systems. Overall WineFS suffers from the least amount of page faults, up-to 450× lower than the other file systems.

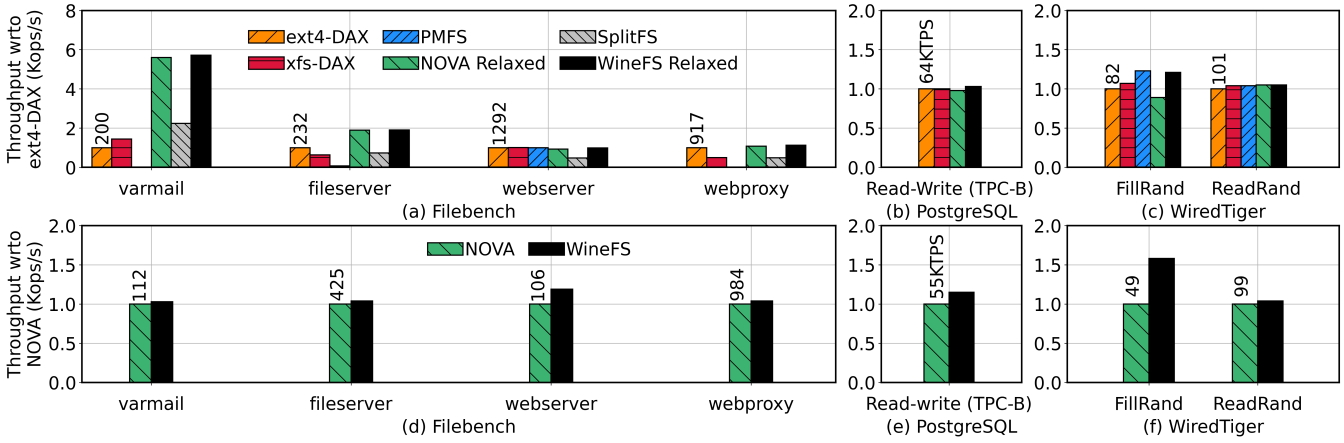

Figure 9. Performance of applications using POSIX system calls on clean file systems. WineFS has equal or better than the best file system in a clean file system setup. Performance of file systems in the relaxed mode (metadata consistency) is in (a), (b) and (c) and in the strict mode (data + metadata consistency) is in (d), (e) and (f).

Persistent radix tree. We study the performance of the persistent adaptive radix tree, P-ART [29], on WineFS. P-ART creates a PM pool using the vmmalloc library and pre-faults this region during initialization to avoid page faults in the critical path. We insert 60M keys to the index; page-table mappings are setup during inserts. We then perform 60M lookups of a hot-set of 125K unique keys in random order. The lookups don't suffer from page faults as page-tables are already setup. Figure 8 shows that the median latency of WineFS is **35%** lower than ext4-DAX and **60%** lower than NOVA. WineFS suffers 400× fewer LLC misses and 2× lower TLB misses compared to the next-best ext4-DAX.

Performance on newly created file system. We repeat all the above-mentioned experiments on newly created PM file systems. In general, PM file systems find it easier to map files with hugepages on a new file system. For the sake of brevity, we do not report the results. We find that all file systems perform similarly in a clean setup, with WineFS performing up-to 30% better compared to ext4-DAX and 35% compared to NOVA on YCSB Load A. WineFS outperforms PMFS by 80% in LMDB as PMFS does not get hugepages even in a clean file system setup. WineFS outperforms xfs-DAX by up-to 35% for reasons similar to PMFS. Across all these applications, WineFS gets the best performance or matches the performance of the best PM file system. This indicates that the design of WineFS is effective for memory-mapped files even on a newly created file system.

5.5 Performance for system-call access mode

We now evaluate WineFS on macro-benchmarks and applications that access PM via system calls. Aging does not impact system call performance on PM. We therefore use newly created file systems for these experiments.

Filebench. We use the Filebench [46, 48] macrobenchmark to evaluate WineFS. We use the varmail, fileserver, webserver, and webproxy benchmarks, with configurations as shown in the Table 1. These benchmarks emulate the I/O behavior of several real-world applications.

WineFS and NOVA-relaxed outperform ext4-DAX by up-to **5**×. Ext4-DAX and xfs-DAX perform poorly on varmail due to costly `fsync()` operations and metadata overheads. Although SplitFS outperforms ext4-DAX due to faster appends, it inherits low scalability for `creates` and `deletes` as it relies on ext4-DAX's JBD2 journal. The poor metadata structures, directory traversals, and inode free-lists, limit PMFS's performance on metadata-heavy workloads like varmail. WineFS outperforms existing file systems on other Filebench macrobenchmarks, as shown in the Figure 9 (a) and (d).

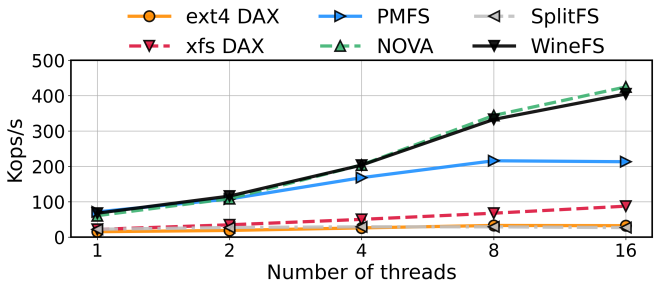

Figure 10. Microbenchmark: Scalability. WineFS throughput scales with increasing #threads on metadata-heavy workloads.

PostgreSQL. We use PostgreSQL [38] database and run the read-write workload of pgbench suite (similar to TPC-B). WineFS outperforms NOVA by **15%**, as shown in Figure 9 (b), (e). The performance improvements trace back to overwrites. NOVA has to delete per-inode log entries, add new entries for handling overwrites, and update DRAM indexes to reflect the new data. WineFS only modifies the inode in a journal transaction to point to the newly allocated blocks.

WiredTiger. We use WiredTiger [19], a key-value store that MongoDB uses by default, and run the `FillRandom` and the `ReadRandom` workloads with 1KB sized values. In FillRandom, WiredTiger on WineFS performs **60%** faster than on NOVA, and outperforms ext4-DAX by **20%**, as shown in Figure 9 (c), (f). WineFS outperforms NOVA because WiredTiger appends data at unaligned offsets and NOVA forces these appends to a new 4KB page to ensure data atomicity, causing high write amplification. NOVA copies the data in the partial block to the new block and then appends new data. WineFS continues to append to partially full blocks without having to copy old data like NOVA, while ensuring data atomicity via journaling. In ReadRandom, WiredTiger's throughput remains the same across different file systems.

Other utilities. We evaluate WineFS using kernel compilation, tar, and rsync. Linux kernel compilation (v5.6; using 64 threads) takes similar time across all PM file systems. WineFS has comparable performance as its competitors across all utilities; we omit further details due to space constraints.

5.6 Scalability

We measure the scalability of WineFS using a multi-threaded system call workload: we `create` a file, `append` at 4KB granularities, `fsync`, and `unlink` in each thread. Figure 10 shows the results. WineFS and NOVA have the best scalability. NOVA achieves its scalability through per-inode logs which have the side effect of fragmenting free space (and reducing performance for memory-mapped files). WineFS achieves similar scalability by using per-CPU fine-grained journals that minimize the fragmentation. ext4-DAX and xfs-DAX have low scalability as they use a stop-the-world approach on `fsync()` to flush the journal to PM. SplitFS inherits low scalability as it runs atop ext4-DAX. Finally, PMFS scales well due to its fine-grained journaling. All the file systems plateau beyond 16 thread due to the scalability bottlenecks in the VFS layer.

5.7 Resource Consumption

WineFS consumes memory for its DRAM metadata indexes (e.g., red-black trees used for directory indexing, keeping track of free extents and inode free lists). It also additionally consumes CPU time to execute background activity such as journal space reclamation and retroactive rewriting of files.

Memory usage. WineFS uses a per-directory RB-tree to index the directory entries. The directory entries are hashed and stored, requiring less than 64B of memory per entry. Filling an entire 500GB partition of PM (used in our evaluation) with small 4KB files requires less than 10GB of DRAM. The memory usage of other DRAM metadata indexes such as the alignment-aware allocator and inode free lists is insignificant compared to the per-directory RB-tree, and can be safely assumed to be less than 1GB.

CPU utilization. WineFS uses a background thread to reclaim space occupied by committed transactions in the per-CPU journals, and uses another background thread in case of reactive re-writing of files to get hugepages. We expect the re-writing of files to be extremely rare (as discussed in §3.6), and in the common case, not utilizing a thread.

5.8 Summary

Overall, we show that WineFS achieves its goals (§3.1). It conserves hugepages and provides excellent performance for applications using memory-mapped files. It does not sacrifice performance for applications using system calls to access PM, performing equal to or better than the state-of-the-art PM file systems. It provides atomic, synchronous data and metadata operations. It achieves these properties while being POSIX-compliant and not requiring any changes to the application.

6 Related Work

We now place WineFS in the context of prior work. While WineFS builds on a wealth of prior research, WineFS is the first hugepage-aware file system that achieves good performance for applications using memory-mapped files or system calls to access PM.

Hugepage-friendliness. Prior work studied the high cost of page faults in PM file systems and proposed changes to the memory sub-system [11]; in contrast, WineFS does not require any changes to the memory subsystem. Intel PMDK [22] recommends using ext4-DAX [33] or xfs-DAX [1, 21] with 2MB sized blocks, to ensure hugepage-friendliness. However, the downside of this approach is high space amplification for applications with files smaller than 2MB.

NOVA [49] attempts to allocate hugepage-aligned physical extents, but requires allocation requests to be exact multiples

of 2MB. The log-structured design of NOVA fragments free space; NOVA does not seek to prevent this. The free-space allocator in other PM file systems ignores fragmentation and physical alignment, causing a decrease in hugepages. WineFS is the first PM file system that has hugepage-friendliness as a primary design concern and shows that hugepages can be achieved without high space amplification.

Aging in PM file systems. Prior work has studied aging in file systems on magnetic hard drives [43] and SSDs [13, 14, 26]. While prior work has studied aging on emulated persistent memory [26], our work is the first to not only understand the problems that occur with aging on actual PM, but also address it via WineFS.

TLB effects. The correlation of increased performance due to larger TLB reach and coarser TLB mappings on PM was noted by prior work [32], but WineFS is the first to explain the reason behind these observations. Recent work [9] also speaks about the perils of TLB shootdowns, though not in the context of PM.

Fsync overhead. PM file systems like BPFS [12], PMFS [40], NOVA [49], Strata [28], and SplitFS [25] have reduced the overhead of `fsync()`. However, the log-structuring or copy-on-write design of NOVA, Strata, and SplitFS causes fragmentation and reduces hugepages. PMFS uses a single journal that becomes the bottleneck in multi-threaded applications. WineFS uses fine-grained per-CPU undo journal which minimizes `fsync()` overhead without trading off hugepages.

7 Conclusion

This paper presents WineFS, a hugepage-aware PM file system. WineFS demonstrates that it is possible to design a file system that achieves good performance for applications accessing PM via either memory-mapped files or system calls. WineFS revisits a number of file-system design choices in the light of hugepage-awareness. The design of WineFS allows it to resist aging, offering the same performance in the aged and unaged setting. WineFS is publicly available at https://github.com/utsaslab/winefs.

Acknowledgments

We thank our shepherd, Haibo Chen, and the anonymous reviewers at SOSP 21 and OSDI 21 for their insightful comments and suggestions. This work was supported by NSF CAREER #1751277, the UT Austin-Portugal BigHPC project (POCI-01-0247-FEDER-045924) We thank Intel for generously providing access to the testbed used in our evaluation. We thank the members and companies of the PDL Consortium (Amazon, Facebook, Google, HPE, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Pure Storage, Salesforce, Samsung, Seagate, Two Sigma, and Western Digital) and VMware for their interest, insights, feedback, and support.

References

[1] 1996. XFS Filesystem Structure. https://xfs.org/docs/xfsdocs-xml-dev/XFS_Filesystem_Structure//tmp/en-US/html/index.html.
[2] 2012. Symas Lightning Memory-Mapped Database. https://symas.com/products/lightning-memory-mapped-database/.
[3] 2015. XFS: DAX support. https://lwn.net/Articles/635514/.
[4] 2017. RocksDB | A persistent key-value store. http://rocksdb.org.
[5] 2019. Redis: In-memory data structure store. https://pmem.io/2020/09/25/memkeydb.html.
[6] 2020. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html
[7] Nitin Agrawal, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2009. Generating realistic impressions for file-system benchmarking. *ACM Transactions on Storage (TOS)* 5, 4 (2009), 1–30.
[8] Nitin Agrawal, William J Bolosky, John R Douceur, and Jacob R Lorch. 2007. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)* 3, 3 (2007), 9–es.
[9] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't shoot down TLB shootdowns!. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–14.
[10] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Emmett Witchel. 2020. Assise: Performance and Availability via Client-local {NVM} in a Distributed File System. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 1011–1027.
[11] Jungsik Choi, Jiwon Kim, and Hwansoo Han. 2017. Efficient Memory Mapped File I/O for In-Memory File Systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2017, Santa Clara, CA, USA, July 10-11, 2017*. https://www.usenix.org/conference/hotstorage17/program/presentation/choi
[12] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. 133–146.
[13] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Martin Farach-Colton. 2017. File Systems Fated for Senescence? Nonsense, Says Science!. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17)*. 45–58.
[14] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A Bender, William Jannen, Rob Johnson, Donald Porter, and Martin Farach-Colton. 2019. Filesystem aging: It's more usage than fullness. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*.
[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.
[16] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 478–493. https://doi.org/10.1145/3341301.3359637
[17] John R Douceur and William J Bolosky. 1999. A large-scale study of file-system contents. *ACM SIGMETRICS Performance Evaluation Review* 27, 1 (1999), 59–70.
[18] Facebook. 2017. RocksDB | A persistent key-value store. http://rocksdb.org.
[19] Alexandra Fedorova, Craig Mustard, Ivan Beschastnikh, Julia Rubin, Augustine Wong, Svetozar Miucin, and Louis Ye. 2018. Performance

comprehension at WiredTiger. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 83–94. https://doi.org/10.1145/3236024.3236081

[20] Google. 2021. Colossus under the hood: a peek into Google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system.

[21] Christoph Hellwig. 2009. XFS: the big storage file system for Linux. *login:: the magazine of USENIX & SAGE* 34, 5 (OCT) (2009), 10–18. https://dialnet.unirioja.es/servlet/articulo?codigo=4957012

[22] Intel. 2018. Persistent Memory Development Kit. http://pmem.io.

[23] Intel. 2018. PmemKV | Key/Value Datastore for Persistent Memory. https://github.com/pmem/pmemkv.

[24] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[25] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. SplitFS: A File System that Minimizes Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Ontario, Canada.

[26] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R Ganger. 2018. Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 691–704.

[27] Linux kernel developers. 2008. Linux POSIX file system test suite. https://lwn.net/Articles/276617/.

[28] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas E. Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. 460–477. https://doi.org/10.1145/3132747.3132770

[29] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 462–477. https://doi.org/10.1145/3341301.3359635

[30] Linux. 2019. Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[31] LMDB. 2012. Database Microbenchmarks. http://www.lmdb.tech/bench/microbench/.

[32] Tony Mason, Thaleia Dimitra Doudali, Margo Seltzer, and Ada Gavrilovska. 2020. Unexpected performance of Intel® Optane DC Persistent Memory. *IEEE Computer Architecture Letters* 19, 1 (2020), 55–58.

[33] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Alex Tomas Andreas Dilge and, and Laurent Vivier. 2007. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*. Ottawa, Canada.

[34] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, Vol. 2. Citeseer, 21–33.

[35] Memcached. 2019. The Volatile Benefit of Persistent Memory. https://memcached.org/blog/persistent-memory/.

[36] Dutch T Meyer and William J Bolosky. 2012. A study of practical deduplication. *ACM Transactions on Storage (ToS)* 7, 4 (2012), 1–20.

[37] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding crash-consistency bugs with bounded black-box crash testing. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 33–50.

[38] PostgreSQL. 1996. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.

[39] Memhive PostgreSQL. 2020. Announcing Memhive PostgreSQL. https://www.postgresql.org/about/news/announcing-memhive-postgresql-2088/.

[40] Dulloor Subramanya Rao, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*. 15:1–15:15. https://doi.org/10.1145/2592798.2592814

[41] Storage Review. 2019. Intel Optane DC Persistent Memory Module (PMM). https://www.storagereview.com/intel_optane_dc_persistent_memory_module_pmm.

[42] Denis Serenyi. 2017. Cluster-Level Storage @ Google. Keynote at the 2nd Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Intensive Computing Systems.

[43] Keith A Smith and Margo I Seltzer. 1997. File system aging-increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM, 203–213.

[44] Steven Swanson. 2019. Redesigning File Systems for Nonvolatile Main Memory. *IEEE Micro* 39, 1 (2019), 62–64. https://doi.org/10.1109/MM.2018.2886443

[45] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System.. In *USENIX Annual Technical Conference*, Vol. 15.

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

[47] Yifan Wang. 2012. A statistical study for file system meta data on high performance computing sites. *Master's thesis, Southeast University* (2012).

[48] Andrew Wilson. 2008. The new and improved FileBench. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*.

[49] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*. 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[50] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 478–496. https://doi.org/10.1145/3132747.3132761

[51] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[52] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*.

[53] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. 207–219. https://www.usenix.org/conference/fast19/presentation/zheng