MadFS: Per-File Virtualization for Userspace Persistent Memory Filesystems

Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift, *University of Wisconsin–Madison*

https://www.usenix.org/conference/fast23/presentation/zhong

MadFS: Per-File Virtualization for Userspace Persistent Memory Filesystems

Shawn Zhong* Chenhao Ye* Guanzhou Hu Suyan Qu
Andrea Arpaci-Dusseau Remzi Arpaci-Dusseau Michael Swift
University of Wisconsin–Madison

Abstract

Persistent memory (PM) can be accessed directly from userspace without kernel involvement, but most PM filesystems still perform metadata operations in the kernel for security and rely on the kernel for cross-process synchronization.

We present per-file virtualization, where a virtualization layer implements a complete set of file functionalities, including metadata management, crash consistency, and concurrency control, in userspace. We observe that not all file metadata need to be maintained by the kernel and propose embedding insensitive metadata into the file for userspace management. For crash consistency, copy-on-write (CoW) benefits from the embedding of the block mapping since the mapping can be efficiently updated without kernel involvement. For cross-process synchronization, we introduce lock-free optimistic concurrency control (OCC) at user level, which tolerates process crashes and provides better scalability.

Based on per-file virtualization, we implement MadFS, a library PM filesystem that maintains the embedded metadata as a compact log. Experimental results show that on concurrent workloads, MadFS achieves up to 3.6× the throughput of ext4-DAX. For real-world applications, MadFS provides up to 48% speedup for YCSB on LevelDB and 85% for TPC-C on SQLite compared to NOVA.

1 Introduction

Persistent memory (PM) is a promising candidate for next-generation storage devices. PM DIMMs are connected on the memory bus and deliver near-DRAM performance while persisting data across power-offs. They create new opportunities for building storage systems.

With revolutionary hardware available, the software stack needs to evolve accordingly. Traditional kernel filesystems require I/O operations to cross the user-kernel boundary and go through layers of the storage stack, introducing significant software overhead. In response to this observation, many PM filesystems have been proposed to perform I/O in userspace [5, 8, 12, 25, 30, 41, 43]. The challenge is that a userspace process is untrusted and unreliable: it could corrupt metadata and threaten filesystem integrity; it could crash in a shared critical section, blocking other processes. These realities impose challenges for metadata operations and sharing. Existing userspace filesystems bypass the kernel for data operations, but typically still rely on the kernel for metadata management [5, 8, 25, 30] with its inefficient storage stack. In terms of sharing, most userspace filesystems either do not support cross-process sharing [8] or rely on a kernel-granted lease [5, 12, 30].

To address these challenges, we introduce *per-file virtualization*, where a complete set of file functionalities, including metadata management, crash consistency, and concurrency control, are implemented in a userspace virtualization layer and managed on a per-file basis for regular files. For userspace metadata management, we observe that some metadata are private to each file and have a similar trust model to the file data. Thus, we propose *metadata embedding*, where insensitive metadata (e.g., block mapping and file size) are embedded in the file. This enables efficient metadata management in userspace without sacrificing permission enforcement. In particular, embedding block mapping provides additional benefits when copy-on-write (CoW) is used for data crash consistency. For a process with memory-mapped files, existing kernel-level CoW requires updating the page table on file writes, causing expensive TLB shootdowns. With metadata embedding, the block mapping can be changed entirely in userspace without kernel involvement. To support cross-process concurrency control, we use the file data itself as the communication medium and implement non-blocking synchronization. This design simplifies the failure model and provides better concurrency than locks.

Based on per-file virtualization, we present MadFS[1], a library filesystem for persistent memory that provides strong data crash consistency and linearizable concurrency control

*Both authors contributed equally to this work.

[1] MadFS stands for metadata embedded filesystem.

in userspace. MadFS requires no modification to the kernel or application and can run on top of any direct access (DAX) filesystem with `mmap` support (e.g., ext4-DAX). To provide strong data crash consistency, MadFS performs CoW on data updates. MadFS introduces a level of indirection that maps *virtual* blocks seen by applications to *logical* blocks backed by the underlying kernel filesystem. This block mapping is embedded in the file for efficient userspace CoW and maintained as a log for crash consistency. We implement lock-free optimistic concurrency control (OCC) to support concurrent access to the same file cross processes. Specifically, a writer tentatively makes changes in a private workspace. Before committing to the log, the writer detects conflicts by checking the movement of the log tail, and partially redoes the changes if necessary. Compared to lock-based approaches, concurrent readers and writers would not block each other even with overlapping ranges, thus achieving better scalability.

We evaluate MadFS using a variety of microbenchmarks and macrobenchmarks. MadFS achieves up to 3.6× throughput for ext4-DAX on concurrent microbenchmarks. For LevelDB running YCSB workload, MadFS provides up to 48% improvement over NOVA. TPC-C workloads over SQLite on MadFS outperform NOVA by 85%.

This paper makes the following contributions:

- We present per-file virtualization, where a virtualization layer implements a complete set of file functionalities, including metadata management, crash consistency, and concurrency control, entirely in userspace.
- We introduce metadata embedding as a novel metadata management technique for userspace filesystems. Embedding insensitive metadata in the file enables efficient modification in userspace.
- In particular, when CoW is used for data crash consistency, we propose embedding the block mapping, which allows it to be updated without the kernel modifying the page table.
- We introduce lock-free optimistic concurrency control (OCC) for userspace cross-process synchronization, which tolerates process crashes and achieves better scalability.
- Based on per-file virtualization, we present MadFS, a library PM filesystem that maintains the embedded metadata as a compact log. The source code of MadFS is available at `https://github.com/WiscADSL/MadFS`.
- We evaluate MadFS using microbenchmarks and macrobenchmarks to show that it provides high throughput for both single-threaded and multi-threaded workloads.

2 Background and Motivation

Persistent memory (PM) is an emerging hardware technology that provides durability with DRAM-like latency. PM is considered both a new generation of denser memory and a high-performance storage device. In this paper, we explore the storage aspect of PM.

The byte-addressability of PM, like DRAM, enables CPUs to directly read/write data through load/store instructions. After data is stored in a memory location, it may still reside in the CPU cache, so one needs to flush the cache line explicitly (e.g., via `clwb` or `clflushopt`) for persistence. Alternatively, non-temporal stores (e.g., `movnti`) can be used to persist data directly, bypassing the CPU cache. For ordering constraints, a memory fence (e.g., `sfence`) is needed to serialize memory instructions.

One of the commercially available PM products is Intel Optane Persistent Memory [1]. Intel announced the winding down of the Optane business in Q2 2022 [10]. This work is not specific to Intel Optane PM. We only require that the PM is byte-addressable and applications can directly access the data stored on the PM via memory-mapped I/O.

There has been a rich set of work on building more efficient filesystems for PM. In this section, we broadly classify them into userspace and kernel filesystems and then discuss their challenges in metadata management, crash consistency, and concurrency control.

2.1 Filesystems for Persistent Memory

Kernel filesystems. Mature Linux filesystems such as ext4 and XFS introduce direct access (DAX) mode [7, 42], which bypasses the page cache and allows applications to directly access file data stored on PM via memory-mapped I/O. These DAX filesystems only ensure metadata consistency in the presence of failures, while the responsibility of maintaining data consistency on memory-mapped regions falls on the applications. There are also research kernel filesystems designed for PM. BPFS [9] uses a tree layout similar to WAFL [23] and avoids cascading CoW via short-circuit shadow paging. PMFS [14] combines atomic in-place updates, journaling, and CoW to support efficient crash consistency, and also advocates the use of huge pages to reduce paging costs. NOVA [45] implements log-structured metadata for each file and CoW data crash consistency.

Userspace filesystems. With ultra-fast hardware, software overhead becomes non-trivial. Thus, many PM filesystems have proposed to bypass the kernel [5, 8, 12, 25, 30, 41, 43]. FLEX [43] calls `mmap` after `open` and intercepts data operations to handle them in userspace via memory instructions. SplitFS [25] similarly handles data operations in userspace with memory-mapped I/O but relies on a modified ext4-DAX for metadata operations. It introduces a new system call `relink`, which reassigns data blocks from one file to another. For append operations, SplitFS redirects data to a temporary staging file and invokes `relink` on `fsync` to publish the newly written data to the target file. Libnvmmio [8] builds on memory-mapped I/O and equips each block with a journal to provide scalable crash-consistent I/O.

2.2 Challenges in Metadata Management

Metadata safety is critical to filesystem integrity. In kernel filesystems, metadata is managed exclusively by the kernel for security reasons. A major challenge of userspace filesystems comes from untrusted libraries. Thus, many of them still rely on the kernel for metadata management (e.g., SplitFS [25], Strata [30], and KucoFS [5]). Unfortunately, data operations can be tightly coupled with metadata operations, defeating the purpose of kernel bypassing and leading to lower performance. For example, SplitFS appends data to a staging file, but still requires the `relink` system call to swap the data blocks from the staging file to the target one on each `fsync`.

A few filesystems also bypass the kernel for metadata operations. Aerie [41] provides applications with direct access to PM for reading/writing data and reading metadata, while metadata updates are handled by a trusted filesystem service via socket-based remote-procedure call (RPC). One drawback of this approach is that RPCs are expensive and incur the overhead of context switches. Aerie uses batching to reduce the number of RPCs at the cost of visibility. ZoFS [12] introduces a new abstraction called coffer. The dentries, inodes, and data blocks for a directory subtree are stored in a coffer if they share the same permission. ZoFS relaxes the protection domain from file to coffer and relies on the Intel Memory Protection Key (MPK) hardware for security. Due to hardware limitations of MPK, the number of simultaneously memory-mapped coffers cannot exceed 15.

2.3 Challenges in Crash Consistency

Crash consistency is critical to filesystems. PM only guarantees the atomicity of a single 64-bit store, so filesystems need to build their own constructs for crash consistency.

To ensure metadata crash consistency, PM filesystems commonly use journaling [5, 14, 25, 30, 41, 42]. Kernel filesystems adapted for PM, such as ext4-DAX, rely on Linux journaling block device (JBD) [28] for metadata journaling. However, JBD was designed with block devices in mind and writes in whole blocks, causing write amplification [4, 43]. SplitFS also uses JBD for the crash consistency of `relink` and suffers the same problem. Many filesystems tailored for PM leverage the byte-addressability to persist journal/log entries with a finer granularity [14, 45]. NOVA equips each inode with a private log. Cross-file updates are implemented via journaling to update multiple log tails. BPFS [9] uses CoW for metadata updates. SoapFS [13] and ZoFS [12] employ soft update [15] for metadata crash consistency.

For data crash consistency, CoW is commonly used [5, 9, 14, 25, 45]. However, CoW has two major drawbacks when used with memory-mapped I/O. First, huge pages have been shown to have significant performance improvements for PM filesystems due to fewer page faults, less TLB shootdown, and shorter page table walk [14, 24, 25]. However, an open issue brought out by PMFS is that CoW does not work well with huge pages: the granularity of CoW is coupled with the page size, which for huge pages is 2 MB or 1 GB on x86-64. Writing to a sub-page results in copying the entire page, causing significant write amplification [14]. SplitFS's `relink` changes the block mapping at the granularity of 4 KB blocks. This breaks the contiguity of the file on the physical PM and thus prevents the use of huge pages [24].

Second, in addition to huge pages, kernel-level CoW causes expensive TLB shootdowns [2,3,8,40]. During CoW, the page table should be updated so that the virtual address region is backed by the new pages. The kernel needs to flush the TLB on the local core, send an inter-processor interrupt (IPI) to the other cores to flush the remote TLB, and wait for all cores to finish. The whole process can take several microseconds to complete [40], which is expensive for PM devices with sub-microsecond latency [47].

Another option for data crash consistency is data journaling. Strata [30] allows applications to write to a private log in PM and relies on the kernel to digest the data to a slower storage device. Libnvmmio [8] equips each block with a journal and implements background checkpointing. In general, data journaling faces the issue of double writes. Both Strata and Libnvmmio make the digestion/checkpointing asynchronous to remove it from the critical path at the cost of visibility.

Some PM filesystems do not provide data crash consistency, including ext4-DAX, FLEX, PMFS, Aerie, and ZoFS. In this case, applications have to detect and react to inconsistent file data upon failures. Previous studies [35, 36] have shown that many applications fail to handle inconsistent data correctly. Data crash consistency is a desirable property for filesystems if the overhead is acceptably low.

2.4 Challenges in Concurrency Control

For kernel filesystems, the kernel itself acts as a single centralized entity for synchronization. The inode lock ensures that only one thread is operating on the same file at a time. For userspace filesystems, however, concurrency control is challenging, especially in cross-process cases. For example, a process could crash while holding a lock, blocking other processes. To prevent this situation, the lock must be visible to the kernel so that the kernel can release it after a crash (e.g., robust mutex [27]). This introduces additional kernel involvement and can cause processes to sleep on the critical path of data operations.

As a result, most userspace PM filesystems either do not support cross-process synchronization [8] or use lease-based locking [5, 12, 30, 41]. Aerie implements a lock service in the filesystem service. Each application process is equipped with an additional clerk thread to communicate with the lock service and synchronize with others. In Strata and ZoFS, leases are granted by the kernel. KucoFS uses a two-level locking scheme with kernel-granted leases for inter-process synchronization and userspace range locks for intra-process synchronization. In all these cases, there exists a centralized coordinator to manage leases. This adds communication overhead when multiple processes access the same file concurrently.

The lease timeout is another source of complexity. Timeout relies on the assumption about the maximum completion time of an operation, which could be unsafe. For example, a writer starting with a valid lease can finish with the lease expired. In this case, other threads will see partial data. A write operation can take an arbitrarily long time to complete due to kernel CPU scheduling or large I/O sizes. This will cause correctness issues with lease-based locking.

3 Per-File Virtualization

To address these challenges, we propose *per-file virtualization*, where a userspace virtualization layer implements a complete set of file functionalities, including metadata management, crash consistency, and concurrency control, on a per-file basis for regular files.

Kernel-bypassing with metadata embedding. We observe that some of the file metadata (e.g., block mapping and file size) are private to each file, and share the same protection domain as the file data. This allows us to embed a subset of the metadata directly into the file to avoid the slow kernel I/O stack for certain metadata operations, especially those that are tightly coupled with data operations (e.g., CoW changing block mapping). Compared to other techniques for userspace metadata management, this method neither relies on a trusted entity as in Aerie nor expands the protection domain beyond a file as in ZoFS. Permission-related metadata (e.g., access mode, owner, and group) must not be embedded. The kernel filesystem shall still manage the permission and enforce access control when a file is opened. Metadata embedding does not apply to directories since the hierarchical structure must be visible to the kernel to enforce access control. We leverage the mature constructs of the kernel to handle directory operations, while the virtualization layer manages the embedded file metadata and ensures its crash consistency.

Decoupling of block- and memory-mapping for CoW. Embedding block mapping, in particular, enables efficient userspace block management since the embedded block mapping can be modified independently from the memory mapping. This provides two major benefits when using CoW for data crash consistency. First, the granularity for CoW is no longer associated with huge page sizes. CoW can operate at block granularity within the file, while the kernel still sees the file as a contiguous region on the PM. This allows the usage of huge pages during `mmap`. Second, block mapping updates can be done in userspace via store instructions. The kernel no longer needs to modify the page table. The nanosecond-level cache coherence protocol [18, 33] ensures cross-core consistency as opposed to microsecond-level TLB shootdown [40].

Non-blocking concurrency control. The embedding of metadata brings new opportunities for concurrency control in userspace. As a file is now a self-contained entity with both metadata and data stored in it, processes that memory-map the same file can use the shared PM region for cross-process synchronization, without relying on external entities. We argue that locking is not a good candidate for cross-process synchronization, as the lock owner can crash in the middle of a critical section. Detecting the lock owner's crash without the kernel is difficult if not impossible. Instead, we propose to use atomic primitives (e.g., compare-and-swap) to implement non-blocking synchronization, where the suspension or crash of a single process does not prevent others from making progress [19–21]. In this way, inter- and intra-process concurrency control is handled uniformly, and the failure model is greatly simplified. Non-blocking synchronization also brings better concurrency, since operations do not block each other, even with overlapping ranges.

Summary. With per-file virtualization, we aim to push file functionalities into userspace as much as possible. Metadata embedding bypasses the kernel for metadata management. Embedding block mapping enables efficient userspace CoW for crash consistency. Non-blocking synchronization allows cross-process concurrency control to be enforced without kernel involvement. All the techniques are applied on a per-file basis and there is no global data structure.

4 MadFS: Design and Implementation

Based on per-file virtualization, we implement MadFS, a userspace library filesystem overlaid on top of any DAX kernel filesystem supporting `mmap` (e.g., ext4-DAX). It intercepts POSIX I/O calls and requires no modifications to the application. MadFS memory-maps the file on open, so subsequent data operations (e.g., `read` and `write`) can be handled in userspace via load and store. MadFS provides data crash consistency through CoW. It embeds metadata in the file to avoid kernel crossing for block mapping updates and delivers instant visibility. MadFS employs lock-free optimistic concurrency control to provide high concurrency with cross-process linearizability.

The architecture of MadFS is shown in Figure 2. A MadFS file is a self-contained file on the underlying DAX filesystem. Upon file creation, MadFS creates the file on the kernel filesystem and initializes the basic structure to identify itself as a MadFS file. The following discussion assumes operations on the same file.

Embedded block map (§4.1). We introduce a level of indirection that maps *virtual* blocks seen by applications to *logical* blocks managed by the underlying kernel filesystem. We call this indirection the *block map*. The block map is embedded in the file, which allows MadFS to efficiently handle CoW operations in userspace.

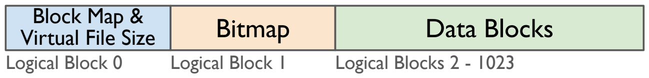

Figure 1: A naive approach for metadata embedding (§4.1)

Compact log-structured metadata (§4.3). To ensure the metadata crash consistency, we maintain the block map as a log persisted in the file. Each block map update is described by a compact 8-byte log entry. For a write operation, MadFS writes to pre-allocated blocks and copies unaligned parts from existing blocks if necessary. MadFS then generates a log entry describing the block map update and finally commits the write by appending the entry to the log. The word-sized (8-byte) log entry ensures the atomicity of the log append and allows for a lock-free concurrency control algorithm.

Lock-free optimistic concurrency control (§4.4). MadFS supports concurrent access to the same file across threads and processes. To achieve high scalability, MadFS employs lock-free optimistic concurrency control (OCC). Concurrent writers do not block each other and are linearized during the log commit. In the case of range overlap, the later writer will detect the conflict during the commit, partially redo the write as needed, and retry the commit. The reader similarly detects overlap and guarantees that it never returns half-written data.

Security. In MadFS, access permission is still enforced by the underlying kernel filesystem during open. To launch an attack, a malicious actor must have permission to write to the file. In this case, the actor could alter the block map, causing others to read the wrong blocks, but this is no different from a traditional filesystem where the actor can directly overwrite file data. For metadata integrity, MadFS treats files as untrusted input and gracefully returns an error on ill-formed files. Furthermore, due to per-file virtualization, the effect of metadata corruption is contained within the file. Similar to other filesystems [12], MadFS does not prevent denial-of-service attacks if the attacker keeps writing to the file.

4.1 Metadata Embedding

To illustrate how metadata embedding allows MadFS to bypass the kernel I/O stack for metadata management, consider a naive design shown in Figure 1. We will later build on this design to add more functionalities.

We denote the blocks backed by the underlying kernel filesystem as *logical* blocks. In this example, the file contains 1024 logical blocks. The first two blocks store metadata; the rest are data blocks, some of which can be unused. We introduce a level of indirection that maps the *virtual* blocks seen by applications to logical data blocks: an application reading the first 4 KB gets the data in the first virtual block, which resides in some logical block. This indirection is maintained in the *block map* as an array of integers. If the virtual block index i maps to logical block index j, then the i-th element of the array is j. The *virtual* file size is the size seen by the applications, and the *logical* file size is the size occupied on the kernel

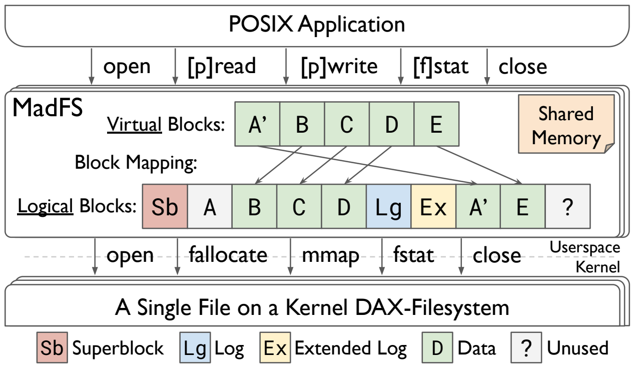

Figure 2: The architecture of MadFS. The application sees *virtual* blocks, which are mapped to the *logical* blocks backed by the kernel filesystem.

filesystem, which is 4 MB in this example. The bitmap indicates whether a data block is in use or not. Security-sensitive metadata is not embedded and is still managed by the kernel.

The embedding of the block map enables MadFS to perform CoW efficiently in userspace. A write operation proceeds in the following steps: ❶ allocate blocks from the bitmap, ❷ write the user buffer to the allocated blocks and copy unaligned parts of existing blocks if any, ❸ update the block map along with the virtual file size, and ❹ return the old blocks to the bitmap.

In this example, we bypass the kernel I/O stack for write and avoid changing the memory mapping for CoW. However, it only considers a file with fixed logical size (§4.2) and does not ensure metadata crash consistency (§4.3) or enforce concurrency control (§4.4).

4.2 Block Management

To facilitate the dynamic growth of the logical file size, we allow the metadata to be stored anywhere in the file. Figure 2 shows the layout of a MadFS file. The metadata is maintained as a log. We will discuss the log structure in detail in §4.3. For the block layout, there are 5 types of logical blocks:

- Sb *Superblock* is the first block, which contains a magic number that identifies MadFS files and a pointer to the first log block.
- Lg *Log blocks* consist of an array of fixed-size log entries, each corresponding to a metadata update (§4.3). Each log block also carries a pointer to the next one, forming a linked list (Fig. 3).
- Ex *Extended log blocks* store extended log entries, which contain additional information about a metadata update that does not fit into the fixed-size log entry (§4.3).
- D *Data blocks* contain the user data. Each virtual block seen by the application is backed by a logical data block.
- ? *Unused blocks* are blocks that are not referenced by the block map. They appear due to pre-allocation from the kernel filesystem and garbage collection (§4.5).

The rest of this section explains the block allocation mechanism. MadFS stores a per-file bitmap in shared memory for coarse-grained coordination. Each thread maintains a local free list as a cache to avoid frequent accesses to the bitmap. To grow the underlying file from the kernel filesystem, hugepage-aware pre-allocation is used to reduce kernel involvement and minimize page faults.

Per-file bitmap in shared memory. Unlike the example in the previous section (Fig. 1), we no longer persist the bitmap on PM, since we can derive from the log whether a logical block is in use or not. Keeping the bitmap as a soft state is common in log-structured filesystems [37, 45] to simplify crash consistency. We maintain the per-file bitmap information in shared memory to coordinate block allocation across processes. If a process opens a file without a bitmap, it constructs the bitmap according to the log. More details about the shared memory initialization are explained in Section 4.6. Blocks are allocated from the bitmap using atomic compare-and-swap (CAS) instructions for lock-free concurrent operations. This implies that the maximum number of contiguous logical blocks we can allocate at a time is 64.

Thread-local free list. The bitmap is accessed by multiple threads, possibly from different processes. To avoid contention, each thread reserves a free list of blocks. They are not referenced by the block map but are still marked as "taken" in the bitmap. When a thread attempts to allocate new blocks, it first allocates from the local free list; if unavailable, it falls back to the bitmap. When a block is freed, instead of immediately returning it to the bitmap, the block is temporarily kept in the free list. This way, an overwrite-intensive thread keeps reusing the blocks in the local free list and rarely allocates from the shared bitmap. The reserved blocks are returned to the bitmap when the file is closed. In rare cases, a process may crash before the reserved blocks are returned. This results in a temporary leak and the blocks can be reclaimed the next time the bitmap is constructed (§4.6).

Hugepage-aware pre-allocation. So far, the allocation mechanism only guarantees that two threads do not get the same block, but the blocks may not actually be backed by the kernel filesystem. When a block is allocated, the logical block index is returned. Later, when the logical index needs to be converted to a memory address for writing, MadFS checks to see if the block is backed. If not, MadFS calls the `fallocate` syscall to grow the file to a multiple of 2 MB and memory-maps the newly allocated region[2]. The same technique is also used during file creation. Pre-allocation amortizes the cost of kernel involvement, and the choice of 2 MB takes advantage of the huge page support in Linux to reduce page faults and TLB misses. Note that CoW does not break the contiguity of the huge page since it only changes the virtual mapping, which is agnostic to the kernel filesystem.

[2]`fallocate` and `mmap` are safe to race. `fallocate` is idempotent and commutative. `mmap` supports multiple mappings of the same physical region.

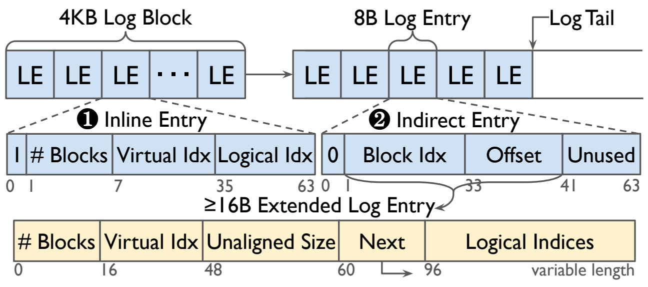

Figure 3: Layout of the log-structured metadata (§4.3). A metadata update is described as either an *inline* log entry or an *indirect* log entry pointing to an *extended* log entry.

4.3 Compact Log-Structured Metadata

In MadFS, a write triggers a block map update, which may span multiple blocks. A write may also expand the virtual file size, which must be modified along with the block map. Therefore, some mechanism is needed to ensure metadata crash consistency. One common choice is journaling. However, journaling is not suitable for non-blocking synchronization because checkpointing requires mutual exclusion. Instead, we structure the metadata as a sequence of log entries, each corresponding to a metadata update. We designed the log entry to be the size of a CPU word (8 bytes) to ensure atomicity and to allow lock-free concurrency control (§4.4).

Log entry layout. As shown in Figure 3, there are two types of 8-byte log entries: ❶ An *inline* log entry is used to represent updates of less than or equal to 64 blocks, which is the maximum number of contiguous logical blocks that the allocator can provide (§4.2). Each inline entry has three fields: a starting virtual block index, a starting logical block index, and the number of blocks the write spans. The three fields together describe a range of virtual blocks mapped to a range of contiguous logical blocks. ❷ An *indirect* log entry is for more complex updates. It carries a pointer to a variable-length *extended* log entry that contains a virtual block range and an array of logical block indices. A write operation with more than 64 blocks will be broken down into multiple allocations, each with a logical index in the extended entry. The unaligned size describes the number of bytes in the last block, which is used to compute the virtual file size. The next field makes it possible to chain multiple extended entries together.

In-memory block table. Because the metadata is now structured as a log, we can no longer directly query the block map and the virtual file size. We use a per-process DRAM data structure called the *block table* to maintain this information. The block table is constructed when a file is opened by scanning through all the log entries. During a read or write, it is queried to obtain the virtual file size and to translate a virtual block index to a logical one. After a new log entry is appended to the log, block table is updated to reflect the metadata update. In the event of a failure, MadFS does not require an explicit recovery phase: the atomicity of the log commit is guaranteed by the CPU's 8-byte atomic store, and the log replay during open always puts the block table in a consistent state. We will discuss the concurrency model of the block table later in Section 4.4.

Example. A write operation proceeds in MadFS as follows: ❶ Allocate new data blocks from the local free list or the bitmap. The writer thread also ensures that the allocated blocks are backed by the underlying filesystem and mapped to memory. ❷ Copy the user buffer and unaligned portions to the newly allocated blocks. ❸ Prepare a log entry describing the block map changes. ❹ Append the entry to the log to publish this write. ❺ For overwrites, return the old data blocks to the local free list for recycling.

4.4 Lock-Free Concurrency Control

MadFS supports cross-process sharing with immediate visibility and guarantees linearizability under concurrent access. To achieve these goals, MadFS uses optimistic concurrency control (OCC) [29]. In this section, we explain the concurrency model of the block table, introduce our lock-free OCC protocol, and then discuss the benefits of OCC.

Concurrency model of the block table. The block table is shared across threads within the same process and operates in a single-writer multi-reader manner. For cross-process visibility, before any data operations, MadFS first checks if the log tail has been moved by other processes. If so, it applies newly committed entries to the block table to keep it up-to-date. Within a single process, only one thread can apply new entries at a time, since this procedure would not benefit from having multiple threads doing the same job. Querying the block table is non-blocking, but the thread may see an inconsistent block table if another thread is concurrently updating it. This is not a problem, since such inconsistency can be caught by our OCC protocol described below.

Lock-free OCC. In database literature [29, 48], an OCC protocol typically takes place in the following four phases:
1. *Begin*: Record the begin timestamp for later validation.
2. *Execute*: Read and modify data in a private workspace.
3. *Validate*: Check if data read have been modified by others.
4. *Commit*: Publish the modified data to make them visible.

Compared to lock-based concurrency control, OCC avoids locking the data during the execution phase. However, the last two phases must be executed in a critical section to avoid race conditions, and locks are still used to protect the critical section [29, 31, 39, 48]. In MadFS, the log-structured metadata makes it a good fit with OCC. The monotonically increasing log tail naturally serves as a timestamp. The word-sized log entry can be committed atomically to the tail via compare-and-swap (CAS), ensuring the atomicity of the validate and commit phases and making the OCC protocol lock-free.

Concurrent writers. A writer first updates the block table and records the current log tail for later validation. It then performs CoW and generates an 8-byte log entry. The writer thread attempts to commit the entry to the recorded tail via CAS. If the recorded tail still points to an empty entry, then the CAS can successfully commit the entry. Otherwise, the tail has been moved, and the current thread needs to check for conflicts. A log entry conflicts with the current one if it modifies the unaligned parts copied during CoW. If there is no conflict, the thread simply recommits the log entry to the new tail. Otherwise, the writer recopies the unaligned parts modified by the conflicting log entry and recommits. Note that the unaligned parts copied do not exceed two blocks.

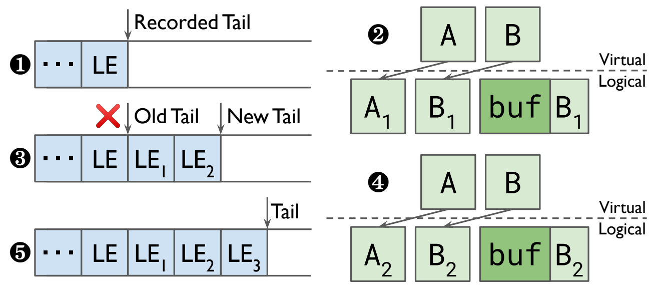

Figure 4: Concurrent writers example (§4.4). Each 🟦 represents an 8-byte log entry. Extended log entries are omitted. Each 🟩 represents a 4 KB data block.

Concurrent readers. A reader also starts by updating the block table and recording the log tail. The reader then copies these blocks to the user buffer. After the copy, if the tail has moved and the added log entries overlap with the range the current thread is reading, the current thread needs to copy the data again. Since the old data blocks are immediately recycled during write (§4.2), the reader must validate up to the latest log tail, so that the blocks read holds valid data.

Example. Figure 4 shows an example of concurrent writers. Suppose a file starts with two virtual blocks A and B with initial contents A_1 and B_1. A writer wants to `pwrite` 6 KB of data at offset 0 while other threads are concurrently writing to the same file. ❶ We first update the block table and record the current log tail. ❷ The writer does a CoW and generates a log entry to commit. ❸ When the thread tries to commit to the recorded log tail via CAS, it finds that the tail has been moved. ❹ Suppose $\boxed{LE_1}$ remaps block A to A_2 and $\boxed{LE_2}$ remaps B to B_2. Although both log entries overlap with the current write, the thread only needs to recopy the unaligned part of B_2. There is no need to recopy block A since it will be completely overwritten. ❺ The current thread successfully commits the log entry to the latest tail at $\boxed{LE_3}$. ❻ Later, the block table will be updated to reflect the changes in the block map.

Discussion. The non-blocking design ensures that a halted process will not prevent other processes from making progress [19, 20]. This simplifies error handling and eliminates the need to detect process crashes. In addition, this design can provide better concurrency than fine-grained locking because it allows concurrent writers to be non-blocking even if their ranges overlap. Multiple writers can operate on their private blocks in parallel. The order of the operations is linearized during CAS, and conflicts are resolved at the bounded cost of copying 2 blocks. On the other hand, the most fine-grained byte-range locks would not allow them to execute concurrently. Note that the OCC protocol also guarantees system-wide progress, and is thus lock-free [21].

Offset-dependent operations. For concurrent I/O operations, offset-independent calls (e.g., `pread`/`pwrite`) are preferred over offset-dependent ones (e.g., `read`/`write`). However, MadFS still guarantees linearization for offset-dependent operations. MadFS uses a per-process ordered queue: a thread performing an offset-dependent operation adds itself to the queue before proceeding to read/modify the file offset. The order in this queue represents a serial order. When the thread finishes reading or writing the data, it must wait for the previous thread in the queue to finish before committing itself. The whole operation is still optimistic, and the CoW is done in parallel.

4.5 Non-Blocking Garbage Collection

To prevent the log from growing indefinitely, we designed a garbage collector (GC) program to clean up the log. MadFS supports non-blocking GC, which does not block concurrent readers or writers.

Creating a new log. Recall that the log blocks are organized as a linked list with the superblock pointing to the head. This design allows us to use the read-copy update (RCU) [32] technique for non-blocking GC. GC replays the log up to before the currently active block and constructs another linked list of log blocks along with the associated extended log entries. The last block in the new linked list points to the currently active block. Finally, we publish the new log by a CAS on the log head stored in the superblock. A later process that opens the file will use the new log.

Reclaiming the old log. GC cannot recycle the old log immediately because some threads may still be using it. One possible solution is to wait until the next time the bitmap is rebuilt and the space for the old blocks is reclaimed. However, this does not work for long-running processes, which prevents the shared bitmap from being rebuilt. Reference counting the log block is not safe because a process can crash without decrementing the counter.

Our solution is to let each thread report the log block it is currently reading to the shared memory. GC can safely recycle a log block if it is not referenced by any reported log blocks since the block will never be accessed in the future. The reported log blocks and their (direct and indirect) successors cannot be immediately recycled. We call them "orphans" as they no longer have a reference from the log head. To free them in the future, we chain the orphans into a new linked list by adding a `next_orphan` field to each log block in addition

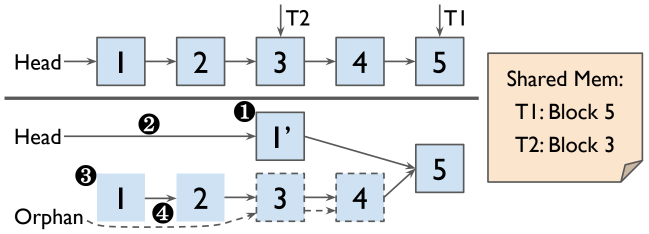

Figure 5: Garbage collection example (§4.5). Each $\boxed{}$ represents a 4 KB log block. The `next` pointer is represented by →, and the `next_orphan` pointer is represented by ⇢. Extended log blocks are omitted.

to the existing `next` pointer. The head of the orphan linked list is persisted in the superblock. The next time the GC runs, it checks to see if any of the orphan log blocks can be freed following the same rule above.

Handling thread crashes. The logical index published in the shared memory is expected to be removed when a thread exits, but a thread may crash before clearing it. We solve this by associating each index with a robust mutex [27] to detect the liveness of the thread. The mutex is locked when the thread is created and unlocked when the process exits or crashes. The GC will try to lock the mutex before accessing the index. Note that the use of the mutex here is only for liveness detection, not mutual exclusion, and no thread is blocked. If GC sees that no thread is currently accessing the file, it can also free the shared memory.

Example. Figure 5 shows an example of garbage collection. There are two I/O threads before GC: T1 is working on the log tail at log block $\boxed{5}$, while T2 is behind at $\boxed{3}$. ❶ GC reads the current log and creates a new linked list of log blocks $\boxed{1'}\rightarrow\boxed{5}$ with the last block untouched. ❷ The log head pointer is atomically changed to point to the new one. ❸ Blocks $\boxed{1}$ and $\boxed{2}$ are immediately recycled since all threads have read beyond them. ❹ For the other log blocks up to the tail block, we organize them into an orphan linked list: Orphan⇢$\boxed{3}$⇢$\boxed{4}$. GC can free the orphans next time when thread T1 moves on to later log blocks.

Discussion. Concurrent readers and writers are never blocked by the garbage collector. An I/O thread only needs to infrequently update a value in the shared memory when it moves to the next log block. Therefore, the impact on the tail latency is minimal. With the compact log format, we expect the log growth to be slow as a single 4 KB log block can store 510 log entries. As a result, GC runs infrequently.

4.6 Implementation

MadFS is implemented in 4.2K lines of C++ code. It supports 24 POSIX functions, including `[f]open`, `[f]close`, `[p]read`, `[p]write`, `mmap`, `fsync`, `lseek`, `stat`, `unlink`, and `rename`. The rest of this section presents implementation details of MadFS.

Shared Memory. The per-file shared memory is created with the same permission as the file. Its name consists of the inode number and the file creation timestamp for uniqueness. The shared memory stores the bitmap (§4.2) and the current-reading log block index (§4.5). When a process opens a file, it tries to memory map the shared memory. If it does not exist, the process reconstructs the bitmap from the log. The shared memory is removed when the file is removed, the garbage collector cleans up, or the operating system cleans up after the user logs out.

Persistence and ordering. We use the non-temporal `memcpy` from PMDK to copy data to persistent memory, bypassing the CPU cache. We use `clwb` to write the log back to PM without flushing the cache, as they may soon be read by other threads. Memory fences are used to ensure that the data blocks are made persistent before log entries, and that extended entries are persisted before indirect entries.

Decoupling of persistence and ordering. Since each log entry only takes 8 bytes, flushing the entire cache line on each log commit is costly. Instead, MadFS only flushes a cache line when a writer attempts to write to the first log entry of the next line[3]. With an explicit `fsync` call, the last cache line written is flushed to ensure durability, which is similar to `dsync` proposed in OptFS [6]. The ordering of writes is always guaranteed by the memory fence of CAS. Note that at most 8 writes are not persistent without any `fsync`.

Handling `mmap` calls. We support `mmap` using a sequence of `mremap` calls to map the data blocks to a contiguous region of memory. This implementation is not optimized for performance and does not provide a crash consistency guarantee.

Correctness. We use continuous integration for correctness testing on a per-pull-request basis. MadFS passes all 209 test cases in the LevelDB test suites, which make extensive use of checksums and put a heavy load on the filesystem. We use Intel's pmemcheck [38], a fork of Valgrind [34] for PM, to validate the durability of stores made to the PM. We also compile MadFS with Clang Sanitizers [16] to check for data races, memory problems, and undefined behavior.

Conversion tool. We implement a tool to convert files between the MadFS format and the normal file format. Converting a file to a MadFS format is fast. The tool allocates some unused blocks, relocates the first data block to make space for the superblock, and then initializes the superblock. It then commits two log entries to describe the block map: one for the relocated data block and one for the rest. To convert a MadFS file to a normal file, the tool grows the file by the virtual file size, dumps the data blocks in their virtual order, and then calls `fallocate` with the `FALLOC_FL_COLLAPSE_RANGE` flag to deallocate all the blocks previously occupied by MadFS.

[3]The time to the flush cannot be after the last slot of a cache line has been written, since a writer could crash after CAS but before a flush is called.

5 Evaluation

In this section, we present the experimental results of microbenchmarks and macrobenchmarks. We demonstrate the completeness, performance, and scalability of MadFS by answering the following questions:

- What is the single-thread performance of MadFS? (§5.1)
- Does MadFS scale to multiple threads? (§5.2)
- What is the overhead of open in MadFS? (§5.3)
- Does garbage collection affect tail latency? (§5.3)
- How does MadFS perform on real-world applications (§5.4)

Setup. Our experiments are performed on an Intel x86 machine with a 128 GB Optane DC persistent memory DIMM. The machine is equipped with two Intel Xeon Silver 8-core 4215R CPUs at 3.20 GHz (with 2 hyper-threads for each physical core) and 32 GB of DDR4 memory. We use Ubuntu 22.04 with custom-built Linux kernel 5.1 with NOVA [44,45] and SplitFS [25] included. For all experiments, we pin threads to the core, disable CPU frequency scaling, and drop the kernel cache before each run.

We compare MadFS (on ext4-DAX) to ext4-DAX, SplitFS, and NOVA. Ext4-DAX does not provide data crash consistency. We run SplitFS in the default POSIX mode, which provides a similar crash consistency guarantee as ext4-DAX. In this mode, SplitFS performs overwrites in-place; for appends, it redirects data to a staging file and invokes `relink` system call to update the block mapping on `fsync`. NOVA is a kernel filesystem that uses CoW for data and maintains log-structured metadata. Among the four filesystems, only NOVA and MadFS provide strong data crash consistency.

5.1 Single-Threaded Microbenchmark

To evaluate the baseline performance of MadFS, we designed six microbenchmarks to measure single-threaded throughput under different I/O sizes and access patterns. All operations are repeated 10,000 times, and all writes are followed by `fsync`. Figure 6 shows the results.

Read. For the read experiment, we measure how long it takes to read data under different I/O sizes. MadFS and SplitFS achieve the best performance since the data is served directly from userspace, with most of the time spent on the memory copy. NOVA and ext4-DAX are slower since they need to go through the kernel storage stack. For large read sizes, the difference between NOVA and MadFS becomes small as the kernel overhead is amortized.

Block-aligned overwrite. In both sequential and random cases, MadFS sustains a stable throughput of 2 GB/s for all I/O sizes. ext4-DAX and NOVA do not saturate the device bandwidth due to software stack overhead. ext4-DAX spends non-trivial time on locks (`dax_read_unlock`) and metadata journaling (called in `ext4_iomap_begin/end`). NOVA performs block allocation during CoW with metadata journaling.

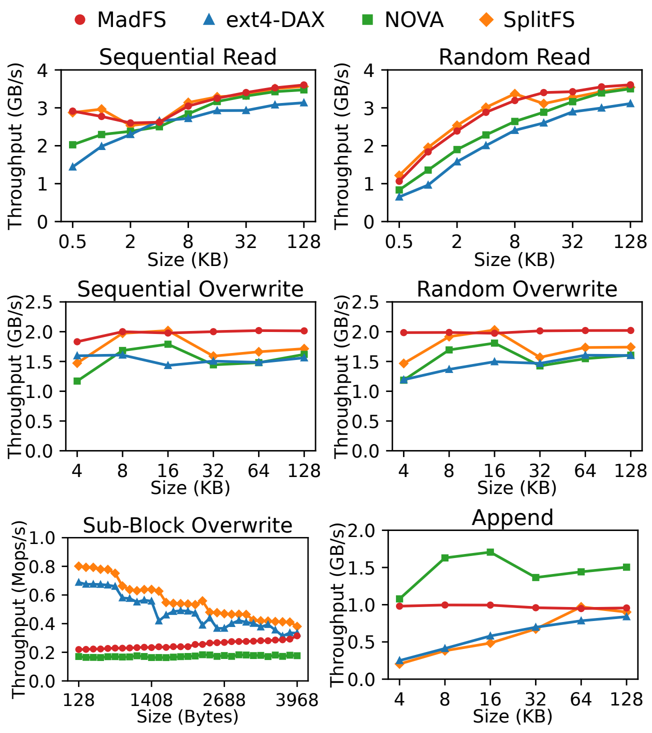

Figure 6: Single-threaded performance. Note for sub-block overwrite, we report throughput in Mops/s instead of GB/s.

SplitFS performs in-place overwrites and does not call the `relink` system call in this experiment.

Sub-block overwrite. For this experiment, we issue sub-block overwrites and report the throughput in Mops/s. MadFS and NOVA employ CoW for data crash consistency and both show an increase in throughput as the write size increases. This is because with a total of 4 KB to be written to the PM, when the size is larger, more data are copied from the user buffer and fewer from the slower PM. Compared with NOVA, MadFS is 30% to 60% faster in terms of throughput with a 1.5*μs* latency margin. SplitFS and ext4-DAX perform in-place overwrites and do not provide a strong data crash consistency guarantee.

Append. For MadFS and SplitFS, the two userspace filesystems running on ext4-DAX, the peak performance does not exceed 1 GB/s, which is half of the throughput for overwrites. This is due to the block allocation zero-out in ext4-DAX. When the userspace filesystem expands the file size via `fallocate`, ext4-DAX reserves the blocks to the file. With memory-mapped I/O, the first access triggers a page fault, which causes the kernel to zero out the blocks [26]. These blocks will soon be overwritten by the user data, which halves the effective bandwidth. This is a fundamental issue for userspace filesystems since un-zeroed blocks cannot be exposed directly to the user for security reasons.

NOVA as a kernel filesystem designed for PM does not have this issue and exhibits similar performance to the overwrite experiment. ext4-DAX should not have this issue. However, it reuses a similar code path with the page fault handler and still zeroed out the blocks (called in `ext4_map_blocks`) before writing to them (in `dax_copy_from_iter`). For small I/O sizes, SplitFS exhibits similar low throughput as ext4-DAX, since each `fsync` triggers a `relink` system call to change the file extent, which involves expensive metadata journaling and inode locking.

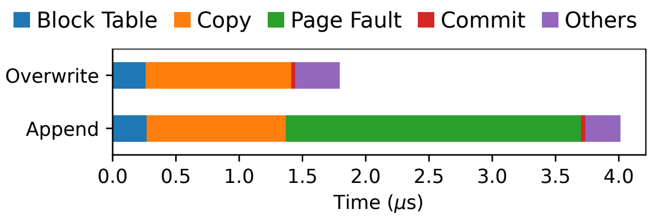

Figure 7: Latency breakdown for 4 KB overwrite and append.

Latency breakdown. Figure 7 shows the time breakdown for 4 KB overwrite and append. Updating the block table involves reading the log entry and applying the changes to the block table. Both overwrite and append take 250 ns on the block table, which is about the same as the latency of accessing 8 bytes from PM. It takes about 1 *μs* to copy the data from DRAM to PM via non-temporal stores. For append, 58% (2.3 *μs*) is spent on the kernel zeroing out. Log commit is as quick as 33 ns, which is about the same latency as a CAS. Others include block allocation, offset calculation, address translation, and block deallocation.

5.2 Multi-Threaded Microbenchmark

In this section, we aim to measure how well MadFS scales when multiple threads access the same file concurrently. We pre-fill a 1 GB file and launch a varying number of threads to read/write the file with offset given by a uniform or Zipfian distribution.

Mixed reads/writes with uniform offset. In this experiment, each thread reads or writes 4 KB at block-aligned offset sampled uniformly at random. With a file size of 1 GB, the probability of two threads operating on the same block is relatively low. Figure 8 shows the result of this experiment. MadFS surpasses other filesystems in all four read-write mixes. Most notably for pure writes, MadFS saturates the device bandwidth at a single thread and sustains the high throughput with more threads. Other filesystems use lock-based concurrency control at inode granularity. SplitFS incurs a performance drop from 1 thread to 2 threads and gradually decreases with more threads. For 95% read, MadFS scales well. It reaches its peak at 11 threads, which matches the device characteristic of the Optane DIMM [47]. SplitFS scales until 6 threads. With more threads, the contention becomes more severe and the throughput drops. With pure read, all filesystems perform well since read operations do not conflict with each other.

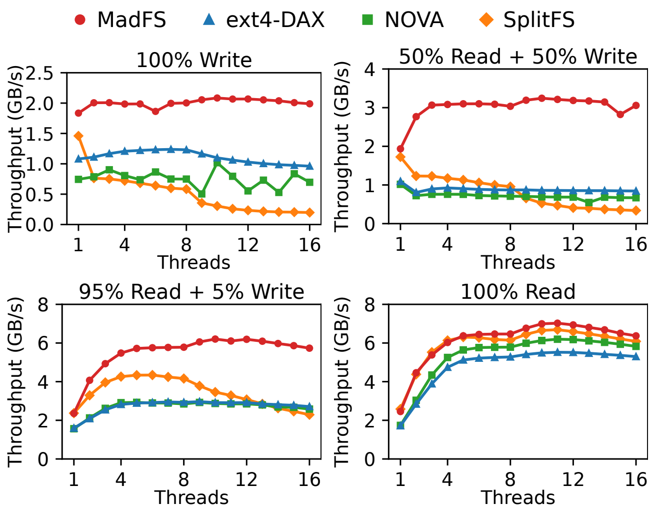

Figure 8: Councurrent 4 KB read/write with uniform offset.

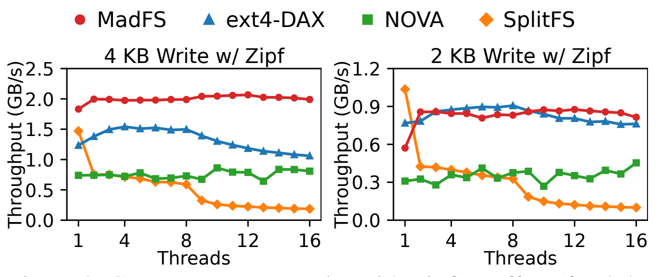

Figure 9: Councurrent pure write with Zipfian offset (θ = 0.9).

Writes with Zipfian offset. To investigate how block-level contention affects scalability, we designed the Zipfian experiments. Each thread writes 4 KB or 2 KB at a block-aligned offset sampled from a Zipfian distribution of θ = 0.9, which results in an access pattern skewed to the first few blocks. Figure 9 shows the result. With 4 KB block-aligned write, the result is similar to the 100% uniform write (Figure 8). The OCC algorithm used by MadFS does not block concurrent threads even if they write to the same block. The order of concurrent writers is linearized during the commit. Since the write is block-aligned, when the commit failed, MadFS only needs to recommit the 8-byte log entry to the new tail and never recopies data (§4.4). Other filesystems use locks at inode granularity, so they do not show significant performance differences between uniform access and Zipfian access. For 2 KB writes, MadFS and NOVA uses CoW and the thread needs to recopy the 2 KB unaligned portion from the new block if newly committed writes overlap with the current one. Nevertheless, MadFS still achieves better performance compared to NOVA. ext4-DAX shows contention with more threads and performs worse than MadFS after 8 threads. Note that only NOVA provides the same strong crash consistency guarantee as MadFS.

Concurrency control. In addition to OCC (§4.4), we experiment with three lock-based concurrency control methods for MadFS and compare their performance under mixed

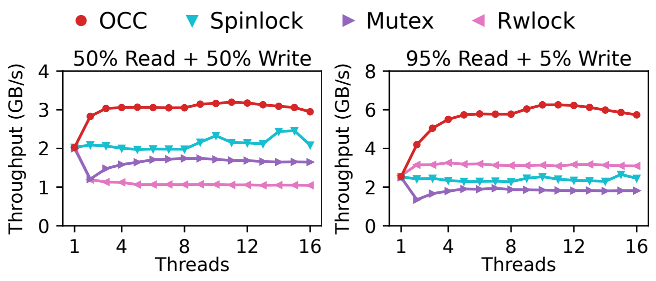

Figure 10: MadFS with different concurrency control methods under uniform 4 KB read/write.

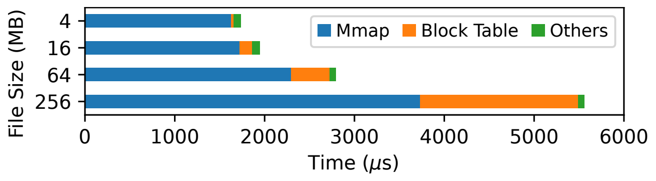

Figure 11: Open latency breakdown. The file size is logical.

read-write 4 KB workload with uniform block-aligned offset. Spinlock is completely in userspace and cannot handle lock-owner crashes in the cross-process scenario. Mutex is set to be robust so the kernel will release it when the owner dies. Reader-writer lock does not support the robustness feature. Only mutex provides the same robustness guarantees as OCC.

Figure 10 shows the result of this experiment. In both workloads, all four concurrency control methods start at the same throughput with a single thread, and OCC surpasses the lock-based concurrency control methods with more threads by a wide margin. With OCC, multiple writers can write to thread-private blocks concurrently without blocking other readers or writers, thus yielding better scalability. The performance of mutex drops from one thread to two threads since mutex puts threads in sleep under contention. Spinlock performs better than mutex as it busy-waits for the lock owner. Reader-writer lock is at the bottom for the 50% read workload due to its operation complexity, but it outperforms spinlock and mutex for the 95% read workload as readers do not block each other.

5.3 Metadata Operations

Open. During file open, in addition to the `open` system call, MadFS need to memory-map the file and replay the log to build the block table. Memory mapping a file takes a fixed cost of 1616 μs plus 17 μs per 2 MB huge page. The same overhead applies to other userspace PM filesystems as well. The log replay is efficient due to the compact log format, taking only 15 ns for an inline entry and 21 ns for an indirect one (with a 16-byte extended entry).

Figure 11 shows the time breakdown to open a file created by repeated 4 KB appends. The majority of the time is spent on memory-mapping the file, especially for small and medium-sized files. Other times include the `open` system call. Due to the open overhead, MadFS may not be suitable for workloads with frequent file opens.

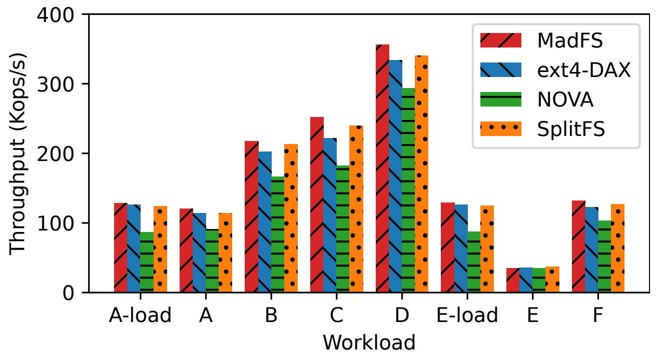

Figure 12: Throughput YCSB workloads on LevelDB.

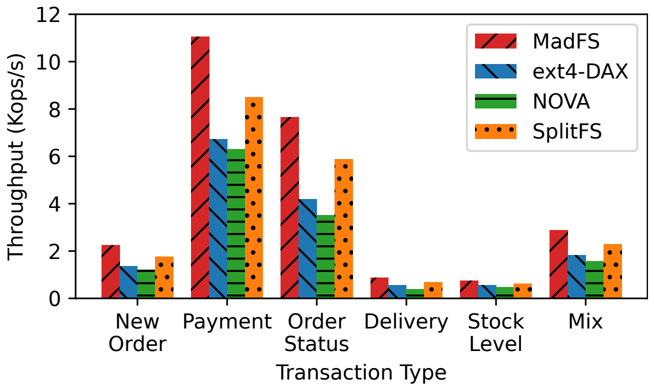

Figure 13: Throughput of TPC-C workloads on SQLite.

Garbage Collection. In this experiment, we aim to measure the effect of the GC on tail latency. We have a writer thread repeatedly doing 4 KB overwrite to a 1 GB file. A GC thread runs every 30 seconds to collect old log entries. The average runtime for GC is 9.1 ms, which is 0.03% of the writer's runtime. With GC, the 99.9%, 99.99%, and 99.999% tail latencies for the writer are 5.06 *μs*, 6.46 *μs*, and 20.77 *μs* respectively, compared to 5.05 *μs*, 6.12 *μs*, and 20.18 *μs* without GC. Overall, the GC finishes quickly and imposes negligible overhead on the I/O thread.

5.4 Real-World Applications

LevelDB with YCSB. To show the completeness of MadFS implementation, we run LevelDB [17], a key-value store based on log-structured merge (LSM) trees. We run the YCSB benchmark [46], a common cloud benchmark for database applications. The benchmark includes 6 workloads: A (50% read + 50% update), B (95% read + 5% update), C (100% read), D (95% read + 5% insert), E (5% insert + 95% scan), and F (50% read + 50% read-modify-write). We issue 1 million operations with a value size of 1 KB.

Figure 12 shows the throughput of all YCSB workloads on LevelDB across four filesystems. The overall trend is MadFS > SplitFS > ext4-DAX > NOVA. For read workload C, MadFS outperforms SplitFS, ext4-DAX, and NOVA by 5%, 12%, and 28% respectively. For write-heavy workloads F, the improvements of MadFS over the other three 4%, 7%, and 22% in the same order. All four filesystems perform similarly on workload E as it has most of the data cached in the memory and is not I/O intensive.

SQLite with TPC-C. SQLite is a widely-used relational database management system [22]. It is used as a library embedded into the end program and stores the entire database as a single file on the filesystem. We drive SQLite with TPC-C, an online transaction processing (OLTP) benchmark that simulates order processing in a multi-warehouse wholesale system [11]. TPC-C includes a mix of 5 transaction types: new order, payment, order status, delivery, and stock level. Each transaction involves a series of SQL statements. We run the TPC-C benchmark using the default configuration: 4 warehouses, 1 district, and 200,000 transactions. The size of the resulting database is 444 MB. The implementation of this benchmark is adopted from SplitFS.

Figure 13 shows the throughput for each of the individual transaction types and the mixed workload. MadFS outperforms other filesystems for all types of transactions since writes in SQLite are mostly block-aligned and do not incur CoW for MadFS. On the mixed workload, MadFS is 26% faster than SplitFS, 58% faster than ext4-DAX, and 85% faster than NOVA.

6 Conclusion

In this paper, we present per-file virtualization which aims to push file functionalities into userspace as much as possible. Metadata embedding allows kernel-bypassing for metadata management. In particular, embedding the block mapping enables efficient userspace CoW for crash consistency. Non-blocking synchronization enables scalable, crash-safe concurrency control without kernel involvement. Based on per-file virtualization, we implement MadFS, a library PM filesystem that maintains embedded metadata as a sequence of compact log entries and employs optimistic concurrency control for linearizability. Our evaluation shows that MadFS yields better performance than ext4-DAX, NOVA, and SplitFS.

Acknowledgments

We are grateful to our shepherd Randal Burns and the anonymous reviewers for their valuable feedback and comments. This material was funded by NSF grants CNS-1838733, CNS-1815656, and CNS-1900758, and supported by gifts from Google, PingCAP, Seagate, and VMware. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or any other institutions.

References

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

[2] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 27–39, 2017.

[3] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down TLB shootdowns! In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[4] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. Fine-grained metadata journaling on nvm. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13. IEEE, 2016.

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

[6] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243, 2013.

[7] Dave Chinner. xfs: DAX support. https://lwn.net/Articles/635514/. Accessed: 2021-01-13.

[8] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 1–16. USENIX Association, July 2020.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[10] Intel Corporation. Intel Reports Second-Quarter 2022 Financial Results. https://www.intc.com/news-events/press-releases/detail/1563/.

[11] Transaction Processing Performance Council. TPC-C: an On-Line Transaction Processing Benchmark. http://www.tpc.org/tpcc/. Accessed: 2021-01-12.

[12] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 478–493, New York, NY, USA, 2019. Association for Computing Machinery.

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

[14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[15] Gregory R Ganger and Yale N Patt. Metadata update performance in file systems. In *OSDI*, volume 94, pages 148–159, 1994.

[16] Google. Google Sanitizers: AddressSanitizer, MemorySanitizer, ThreadSanitizer, LeakSanitizer, and more. https://github.com/google/sanitizers. Accessed: 2021-01-12.

[17] Google. google/leveldb: LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values. https://github.com/google/leveldb, 2011.

[18] Daniel Hackenberg, Daniel Molka, and Wolfgang E Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on microarchitecture*, pages 413–422, 2009.

[19] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[20] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[21] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529. IEEE, 2003.

[22] D. Richard Hipp. SQLite Home Page. https://www.sqlite.org/index.html. Accessed: 2021-01-12.

[23] Dave Hitz, James Lau, and Michael A Malcolm. File system design for an nfs file server appliance. In *USENIX winter*, volume 94, pages 10–5555, 1994.

[24] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R Ganger, Aasheesh Kolli, and Vijay Chidambaram. Winefs: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 804–818, 2021.

[25] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Linux kernel development community. ext4_issue_zeroout identifier - Linux source code - Bootlin. https://elixir.bootlin.com/linux/v5.18.14/source/fs/ext4/inode.c#L417. Accessed: 2021-01-13.

[27] Linux kernel development community. Pthread_mutexattr_setrobust(3) - linux manual page. https://man7.org/linux/man-pages/man3/pthread_mutexattr_setrobust.3.html. Accessed: 2021-01-12.

[28] Linux kernel development community. The Linux Journalling API. https://www.kernel.org/doc/html/latest/filesystems/journalling.html.

[29] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, jun 1981.

[30] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Epoch-based commit and replication in distributed OLTP databases. 2021.

[32] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, 1998.

[33] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *2015 44th International Conference on Parallel Processing*, pages 739–748. IEEE, 2015.

[34] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[35] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 433–448, USA, 2014. USENIX Association.

[36] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can applications recover from fsync failures? In *The 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[37] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[38] PMDK team at Intel Corporation. Pmemcheck - persistent memory analyzer. https://pmem.io/valgrind/generated/pmc-manual.html. Accessed: 2021-01-12.

[39] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[40] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–349. IEEE, 2011.

[41] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael Swift. Aerie: Flexible file-system interfaces to storage-class memory. *Proceedings of the 9th European Conference on Computer Systems, EuroSys 2014*, 04 2014.

[42] Matthew Wilcox. DAX: Page cache bypass for filesystems on memory storage. https://lwn.net/Articles/618064/, 10 2014. Accessed: 2021-10-22.

[43] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 427–439, 2019.

[44] Jian Xu and Steven Swanson. NOVA is a log-structured file system designed for byte-addressable non-volatile memories, developed at the University of California, San Diego. https://github.com/NVSL/linux-nova. Accessed: 2021-01-12.

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

[46] Yahoo. Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB/, 2010.

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

[48] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.