SplitFS: Reducing Software Overhead in File Systems for Persistent Memory

Rohan Kadekodi
University of Texas at Austin

Se Kwon Lee
University of Texas at Austin

Sanidhya Kashyap
Georgia Institute of Technology

Taesoo Kim
Georgia Institute of Technology

Aasheesh Kolli
Pennsylvania State University and VMware Research

Vijay Chidambaram
University of Texas at Austin and VMware Research

Abstract

We present SplitFS, a file system for persistent memory (PM) that reduces software overhead significantly compared to state-of-the-art PM file systems. SplitFS presents a novel split of responsibilities between a user-space library file system and an existing kernel PM file system. The user-space library file system handles data operations by intercepting POSIX calls, memory-mapping the underlying file, and serving the read and overwrites using processor loads and stores. Metadata operations are handled by the kernel PM file system (ext4 DAX). SplitFS introduces a new primitive termed relink to efficiently support file appends and atomic data operations. SplitFS provides three consistency modes, which different applications can choose from, without interfering with each other. SplitFS reduces software overhead by up-to 4× compared to the NOVA PM file system, and 17× compared to ext4 DAX. On a number of micro-benchmarks and applications such as the LevelDB key-value store running the YCSB benchmark, SplitFS increases application performance by up to 2× compared to ext4 DAX and NOVA while providing similar consistency guarantees.

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

Keywords Persistent Memory, File Systems, Crash Consistency, Direct Access

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359631

ACM Reference Format:
Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory . In *SOSP '19: Symposium on Operating Systems Principles, October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3341301.3359631

1 Introduction

Persistent Memory (PM) is a new memory technology that was recently introduced by Intel [17]. PM will be placed on the memory bus like DRAM and will be accessed via processor loads and stores. PM has a unique performance profile: compared to DRAM, loads have 2–3.7× higher latency and $1/3^{rd}$ bandwidth, while stores have the same latency but $1/6^{th}$ bandwidth [18]. A single machine can be equipped with up to 6 TB of PM. Given its large capacity and low latency, an important use case for PM will be acting as storage.

Traditional file systems add large overheads to each file-system operation, especially on the write path. The overhead comes from performing expensive operations on the critical path, including allocation, logging, and updating multiple complex structures. The systems community has proposed different architectures to reduce overhead. BPFS [7], PMFS [24], and NOVA [33] redesign the in-kernel file system from scratch to reduce overhead for file-system operations. Aerie [28] advocates a user-space library file system coupled with a slim kernel component that does coarse-grained allocations. Strata [19] proposes keeping the file system entirely in user-space, dividing the system between a user-space library file system and a user-space metadata server. Aerie and Strata both seek to reduce overhead by not involving the kernel for most file-system operations.

Despite these efforts, file-system data operations, especially writes, have significant overhead. For example, consider the common operation of appending 4K blocks to a file (total 128 MB). It takes 671 *ns* to write a 4 KB to PM; thus, if performing the append operation took a total of 675 *ns*,

Rohan Kadekodi and Se Kwon Lee are supported by SOSP 2019 student travel scholarships from the National Science Foundation.

File system	Append Time (*ns*)	Overhead (*ns*)	Overhead (%)
ext4 DAX	9002	8331	1241%
PMFS	4150	3479	518%
NOVA-Strict	3021	2350	350%
SplitFS-Strict	1251	580	86%
SplitFS-POSIX	1160	488	73%

Table 1. Software Overhead. The table shows the software overhead of various PM file systems for appending a 4K block. It takes 671 *ns* to write 4KB to PM. Strict and POSIX indicate the guarantees offered by the file systems (§3.2).

the software overhead would be 4 *ns*. Table 1 shows the software overhead on the append operation on various PM file systems. We observe that there is still significant overhead (3.5 − 12.4×) for file appends.

This paper presents SplitFS, a PM file system that seeks to reduce software overhead via a novel *split* architecture: a user-space library file system handles data operations while a kernel PM file system (ext4 DAX) handles metadata operations. We refer to all file system operations that modify file metadata as *metadata operations*. Such operations include `open()`, `close()`, and even file appends (since the file size is changed). The novelty of SplitFS lies in how responsibilities are divided between the user-space and kernel components, and the semantics provided to applications. Unlike prior work like Aerie, which used the kernel only for coarse-grained operations, or Strata, where all operations are in user-space, SplitFS routes *all* metadata operations to the kernel. While FLEX [32] invokes the kernel at a fine granularity like SplitFS, it does not provide strong semantics such as synchronous, atomic operations to applications. At a high level, the SplitFS architecture is based on the belief that if we can accelerate common-case data operations, it is worth paying a cost on the comparatively rarer metadata operations. This is in contrast with in-kernel file systems like NOVA which extensively modify the file system to optimize the metadata operations.

SplitFS transparently reduces software overhead for reads and overwrites by intercepting POSIX calls, memory mapping the underlying file, and serving reads and overwrites via processor loads and stores. SplitFS optimizes file appends by introducing a new primitive named *relink* that minimizes both data copying and trapping into the kernel. The application does not have to be rewritten in any way to benefit from SplitFS. SplitFS reduces software overhead by up-to 4× compared to NOVA and 17× compared to ext4 DAX.

Apart from lowering software overhead, the split architecture leads to several benefits. First, instead of re-implementing file-system functionality, SplitFS can take advantage of the mature, well-tested code in ext4 DAX for metadata operations. Second, the user-space library file system in SplitFS allows each application to run with one of three consistency modes (*POSIX, sync, strict*). We observe that not all applications require the same guarantees; for example, SQLite does not require the strong guarantees provided by NOVA-strict, and gets 2.5× higher throughput on ext4 DAX and SplitFS-POSIX than on NOVA-strict owing to their weaker guarantees. Applications running with different consistency modes do not interfere with each other on SplitFS.

SplitFS introduces the relink primitive to optimize file appends and atomic data operations. Relink logically and atomically moves a contiguous extent from one file to another, without any physical data movement. Relink is built on top of the `swap_extents ioctl` in ext4 DAX, and uses ext4 journaling to ensure the source and destination files are modified atomically. Both file appends and data overwrites in strict mode are redirected to a temporary PM file we term the staging file. On `fsync()`, the data from the staging file is relinked into the original file. Relink provides atomic data operations without paging faults or data copying.

SplitFS also introduces an optimized logging protocol. In strict mode, all data and metadata operations in SplitFS are atomic and synchronous. SplitFS achieves this by logging each operation. In the common case, SplitFS will write a single cache line worth of data (64B), followed by one memory fence (*e.g.,* `sfence` in x86 systems), for each operation; in contrast, NOVA writes at least two cache lines and issues two fences. As a result of these optimizations, SplitFS logging is 4× faster than NOVA in the critical path. Thanks to relink and optimized logging, atomic data operations in SplitFS are 2–6× faster than in NOVA-strict, providing strong guarantees at low software overhead.

We evaluate SplitFS using a number of micro-benchmarks, three utilities (git, tar, rsync), two key-value stores (Redis, LevelDB), and an embedded database (SQLite). Our evaluation on Intel DC Persistent Memory shows that SplitFS, though it is built on ext4 DAX, outperforms ext4 DAX by up-to 2× on many workloads. SplitFS outperforms NOVA by 10%–2× (when providing the same consistency guarantees) on LevelDB, Redis, and SQLite when running benchmarks like YCSB and TPCC. SplitFS also reduces total amount of write IO by 2× compared to Strata on certain workloads. On metadata-heavy workloads such as git and tar, SplitFS suffers a modest drop in performance (less than 15%) compared to NOVA and ext4 DAX.

SplitFS is built on top of ext4 DAX; this is both a strength and a weakness. Since SplitFS routes all metadata operations through ext4 DAX, it suffers from the high software overhead and high write IO for metadata operations. Despite these limitations, we believe SplitFS presents a useful new point in the spectrum of PM file-system designs. ext4 DAX is a robust file system under active development; its performance will improve with every Linux kernel version. SplitFS provides

Property	DRAM	Intel PM
Sequential read latency (ns)	81	169 (2.08×)
Random read latency (ns)	81	305 (3.76×)
Store + flush + fence (ns)	86	91 (1.05×)
Read bandwidth (GB/s)	120	39.4 (0.33×)
Write bandwidth (GB/s)	80	13.9 (0.17×)

Table 2. PM Performance. The table shows performance characteristics of DRAM, PM and the ratio of PM/DRAM, as reported by Izraelevitz et al. [18].

the best features of ext4 DAX while making up for its lack of performance and strong consistency guarantees.

This paper makes the following contributions:

- A new architecture for PM file systems with a novel split of responsibilities between a user-space library file system and a kernel file system.
- The novel relink primitive that can be used to provide efficient appends and atomic data operations.
- The design and implementation of SplitFS, based on the split architecture. We have made SplitFS publicly available at https://github.com/utsaslab/splitfs.
- Experimental evidence demonstrating that SplitFS outperforms state-of-the-art in-kernel and in-user-space PM file systems, on a range of workloads.

2 Background

This section provides background on persistent memory (PM), PM file systems, Direct Access, and memory mapping.

2.1 Persistent Memory

Persistent memory is a new memory technology that offers durability and performance close to that of DRAM. PM can be attached on the memory bus similar to DRAM, and would be accessed via processor loads and stores. PM offers 8-byte atomic stores and they become persistent as soon as they reach the PM controller [16]. There are two ways to ensure that stores become persistent: (i) using non-temporal store instructions (e.g., `movnt` in x86) to bypass the cache hierarchy and reach the PM controller or (ii) using a combination of regular temporal store instructions and cache line flush instructions (e.g., `clflush` or `clwb` in x86).

Intel DC Persistent Memory is the first PM product that was made commercially available in April 2019. Table 2 lists the performance characteristics of PM revealed in a report by Izraelevitz *et al.* [18]. Compared to DRAM, PM has 3.7× higher latency for random reads, 2× higher latency for sequential reads, $1/3^{rd}$ read bandwidth, and close to $1/6^{th}$ write bandwidth. Finally, PMs are expected to exhibit limited write endurance (about 10^7 write cycles [25]).

2.2 Direct Access (DAX) and Memory Mapping

The Linux ext4 file system introduced a new mode called Direct Access (DAX) to help users access PM [21]. DAX file systems eschew the use of page caches and rely on memory mapping to provide low-latency access to PM.

A memory map operation (performed via the `mmap()` system call) in ext4 DAX maps one or more pages in the process virtual address space to extents on PM. For example, consider virtual addresses `4K` to `8K-1` are mapped to bytes `0` to `4K-1` on file `foo` on ext4 DAX. Bytes `0` to `4K-1` in `foo` then correspond to bytes `10*4K` to `11*4K -1` on PM. A store instruction to virtual address `5000` would then translate to a store to byte `40964` on PM. Thus, PM can be accessed via processor loads and stores without the interference of software; the virtual memory subsystem is in charge of translating virtual addresses into corresponding physical addresses on PM.

While DAX and `mmap()` provide low-latency access to PM, they do not provide other features such as naming or atomicity for operations. The application is forced to impose its own structure and semantics on the raw bytes offered by `mmap()`. As a result, PM file systems still provide useful features to applications and end users.

2.3 PM File Systems

Apart from ext4 DAX, researchers have developed a number of other PM file systems such as SCMFS [31], BPFS [7], Aerie [28], PMFS [24], NOVA [33], and Strata [19]. Only ext4 DAX, PMFS (now deprecated), NOVA, and Strata are publicly available and supported by modern Linux 4.x kernels.

These file systems make trade-offs between software overhead, amount of write IO, and operation guarantees. NOVA provides strong guarantees such as atomicity for file-system operations. PMFS provides slightly weaker guarantees (data operations are not atomic), but as a result obtains better performance on some workloads. Strata is a cross-media file system which uses PM as one of its layers. Strata writes all data to per-process private log, then coalesces the data and copies it to a shared area for public access. For workloads dominated by operations such as appends, Strata cannot coalesce the data effectively, and has to write data twice: once to the private log, and once to the shared area. This increases the PM wear-out by up to 2×. All these file systems still suffer from significant overhead for write operations (Table 1).

3 SplitFS: Design and Implementation

We present the goals of SplitFS, its three modes and their guarantees. We present an overview of the design, describe how different operations are handled, and discuss how SplitFS provides atomic operations at low overhead. We describe the implementation of SplitFS, and discuss its various tuning parameters. Finally, we discuss how the design of SplitFS affects security.

Mode	*Sync. Data Ops*	*Atomic Data Ops*	*Sync. Metadata Ops*	*Atomic Metadata Ops*	*Equivalent to*
POSIX	✗	✗	✗	✓	ext4-DAX
sync	✓	✗	✓	✓	Nova-Relaxed, PMFS
strict	✓	✓	✓	✓	NOVA-Strict, Strata

Table 3. SplitFS modes. The table shows the three modes of SplitFS, the guarantees provided by each mode, and list current file systems which provide the same guarantees.

Technique	*Benefit*
Split architecture	Low-overhead data operations, correct metadata operations
Collection of memory-mmaps	Low-overhead data operations in the presence of updates and appends
Relink + Staging	Optimized appends, atomic data operations, low write amplification
Optimized operation logging	Atomic operations, low write amplification

Table 4. Techniques. The table lists each main technique used in SplitFS along with the benefit it provides. The techniques work together to enable SplitFS to provide strong guarantees at low software overhead.

3.1 Goals

Low software overhead. SplitFS aims to reduce software overhead for data operations, especially writes and appends.

Transparency. SplitFS does not require the application to be modified in any way to obtain lower software overhead and increased performance.

Minimal data copying and write IO. SplitFS aims to reduce the number of writes made to PM. SplitFS aims to avoid copying data within the file system whenever possible. This both helps performance and reduces wear-out on PM. Minimizing writes is especially important when providing strong guarantees like atomic operations.

Low implementation complexity. SplitFS aims to re-use existing software like ext4 DAX as much as possible, and reduce the amount of new code that must be written and maintained for SplitFS.

Flexible guarantees. SplitFS aims to provide applications with a choice of crash-consistency guarantees to choose from. This is in contrast with PM file systems today, which provide all running applications with the same set of guarantees.

3.2 SplitFS Modes and Guarantees

SplitFS provides three different modes: POSIX, sync, and strict. Each mode provides a different set of guarantees. Concurrent applications can use different modes at the same time as they run on SplitFS. Across all modes, SplitFS ensures the file system retains its integrity across crashes.

Table 3 presents the three modes provided by SplitFS. Across all modes, appends are atomic in SplitFS; if a series of appends is followed by `fsync()`, the file will be atomically appended on `fsync()`.

POSIX mode. In POSIX mode, SplitFS provides metadata consistency [6], similar to ext4 DAX. The file system will recover to a consistent state after a crash with respect to its metadata. In this mode, overwrites are performed in-place and are synchronous. Note that appends are not synchronous, and require an `fsync()` to be persisted. However, SplitFS in the POSIX mode guarantees atomic appends, a property not provided by ext4 DAX. This mode slightly differs from the standard POSIX semantics: when a file is accessed or modified, the file metadata will not immediately reflect that.

Sync mode. SplitFS ensures that on top of POSIX mode guarantees, operations are also guaranteed to be synchronous. An operation may be considered complete and persistent once the corresponding call returns and applications do not need a subsequent `fsync()`. Operations are not atomic in this mode; a crash may leave a data operation partially completed. No additional crash recovery needs to be performed by SplitFS in this mode. This mode provides similar guarantees to PMFS as well as NOVA without data and metadata checksuming and with in-place updates; we term this NOVA configuration *NOVA-Relaxed*.

Strict mode. SplitFS ensures that on top of sync mode guarantees, each operation is also atomic. This is a useful guarantee for applications; editors can allow atomic changes to the file when the user saves the file, and databases can remove logging and directly update the database. This mode does not provide atomicity across system calls though; so it cannot be used to update two files atomically together. This mode provides similar guarantees to a NOVA configuration we term *NOVA-Strict*: NOVA with copy-on-write updates, but without checksums enabled.

Visibility. Apart from appends, all SplitFS operations become immediately visible to all other processes on the system. On `fsync()`, appends are persisted and become visible to the rest of the system. SplitFS is unique in its visibility guarantees, and takes the middle ground between ext4 DAX and NOVA where all operations are immediately visible, and Strata where new files and data updates are only visible to

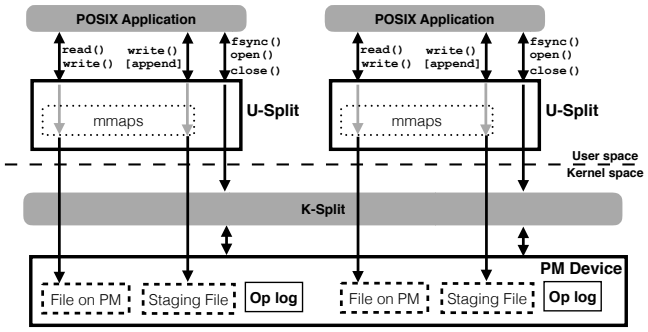

Figure 1. SplitFS Overview. The figure provides an overview of how SplitFS works. Read and write operations are transformed into loads and stores on the memory-mapped file. Append operations are staged in a staging file and *relinked* on `fsync()`. Other metadata POSIX calls like `open()`, `close()`, etc. are passed through to the in-kernel PM file system. Note that loads and stores do not incur the overhead of trapping into the kernel.

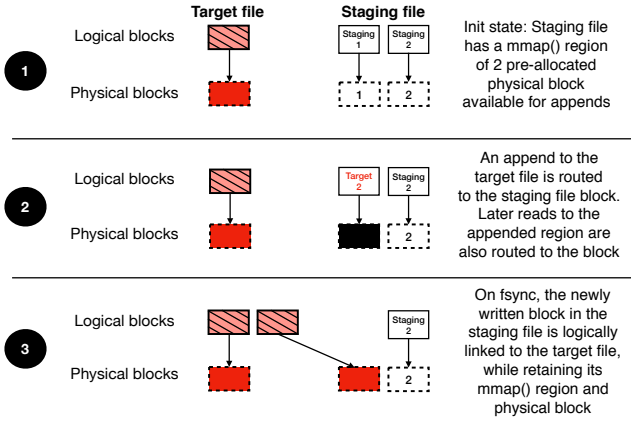

Figure 2. *relink* steps. This figure provides an overview of the steps involved while performing a relink operation. First, appends to a target file are routed to pre-allocated blocks in the staging file and subsequently on an `fsync()`, they are *relinked* into the target file while retaining existing memory-mapped regions.

other processes after the digest operation. Immediate visibility of changes to data and metadata combined with atomic, synchronous guarantees removes the need for leases to coordinate sharing; applications can share access to files as they would on any other POSIX file system.

3.3 Overview

We now provide an overview of the design of SplitFS, and how it uses various techniques to provide the outlined guarantees. Table 4 lists the different techniques and the benefit each technique provides.

Split architecture. As shown in Figure 1, SplitFS comprises of two major components, a user-space library linked to the application called U-Split and a kernel file system called K-Split. SplitFS services all data operations (*e.g.,* `read()` and `write()` calls) directly in user-space and routes metadata operations (*e.g.,* `fsync()`, `open()`, etc.) to the kernel file system underneath. File system crash-consistency is guaranteed at all times. This approach is similar to Exokernel [12] where only the control operations are handled by the kernel and data operations are handled in user-space.

Collection of mmaps. Reads and overwrites are handled by `mmap()`-ing the surrounding 2 MB part of the file, and serving reads via `memcpy` and writes via non-temporal stores (`movnt` instructions). A single logical file may have data present in multiple physical files; for example, appends are first sent to a staging file, and thus the file data is spread over the original file and the staging file. SplitFS uses a *collection of memory-maps* to handle this situation. Each file is associated with a number of open `mmap()` calls over multiple physical files, and reads and over-writes are routed appropriately.

Staging. SplitFS uses temporary files called staging files for both appends and atomic data operations. Appends are first routed to a staging file, and are later relinked on `fsync()`. Similarly, file overwrites in strict mode are also first sent to staging files and later relinked to their appropriate files.

Relink. On an `fsync()`, all the staged appends of a file must be moved to the target file; in strict mode, overwrites have to be moved as well. One way to move the staged appends to the target file is to allocate new blocks and then copy appended data to them. However, this approach leads to write amplification and high overhead. To avoid these unnecessary data copies, we developed a new primitive called *relink*. Relink logically moves PM blocks from the staging file to the target file without incurring any copies.

Relink has the following signature: `relink(file1, offset1, file2, offset2, size)`. Relink atomically moves data from `offset1` of `file1` to `offset2` of `file2`. If `file2` already has data at `offset2`, existing data blocks are de-allocated. Atomicity is ensured by wrapping the changes in a ext4 journal transaction. Relink is a metadata operation, and does not involve copying data when the involved `offsets` and `size` are block aligned. When `offset1` or `offset2` happens to be in the middle of a block, SplitFS copies the partial data for that block to `file2`, and performs a metadata-only relink for the rest of the data. Given that SplitFS is targeted at POSIX applications, block writes and appends are often block-aligned by the applications. Figure 2 illustrates the different steps involved in the relink operation.

Optimized logging. In strict mode, SplitFS guarantees atomicity for all operations. To provide atomicity, we employ an *Operation Log* and use logical redo logging to record the

intent of each operation. Each U-Split instance has its own operation log that is pre-allocated, mmap-ed by U-Split, and written using non-temporal store instructions. We use the necessary memory fence instructions to ensure that log entries persist in the correct order. To reduce the overheads from logging, we ensure that in the common case, per operation, we write one cache line (64B) worth of data to PM and use a single memory fence (`sfence` in x86) instruction in the process. Operation log entries do not contain the file data associated with the operation (*e.g.,* data being appended to a file), instead they contain a logical pointer to the staging file where the data is being held.

We employ a number of techniques to optimize logging. First, to distinguish between valid and invalid or torn log entries, we incorporate a 4B transactional checksum [23] within the 64B log entry. The use of checksum reduces the number of fence instructions necessary to persist and validate a log entry from two to one. Second, we maintain a tail for the log in DRAM and concurrent threads use the tail as a synchronization variable. They use compare-and-swap to atomically advance the tail and write to their respective log entries concurrently. Third, during the initialization of the operation log file, we zero it out. So, during crash recovery, we identify all non-zero 64B aligned log entries as being potentially valid and then use the checksum to identify any torn entries. The rest are valid entries and are replayed. Replaying log entries is idempotent, so replaying them multiple times on crashes is safe. We employ a 128MB operation log file and if it becomes full, we checkpoint the state of the application by calling *relink()* on all the open files that have data in staging files. We then zero out the log and reuse it. Finally, we designed our logging mechanism such that all common case operations (`write()`, `open()`, etc.) can be logged using a single 64B log entry while some uncommon operations, like `rename()`, require multiple log entries.

Our logging protocol works well with the SplitFS architecture. The tail of each U-Split log is maintained only in DRAM as it is not required for crash recovery. Valid log entries are instead identified using checksums. In contrast, file systems such as NOVA have a log per inode that resides on PM, whose tail is updated after each operation via expensive `clflush` and `sfence` operations.

Providing Atomic Operations. In strict mode, SplitFS provides synchronous, atomic operations. Atomicity is provided in an efficient manner by the combination of staging files, relink, and optimized logging. Atomicity for data operations like overwrites is achieved by redirecting them also to a staging file, similar to how appends are performed. SplitFS logs these writes and appends to record where the latest data resides in the event of a crash. On `fsync()`, SplitFS relinks the data from the staging file to the target file atomically. Once again, the data is written exactly once, though SplitFS provides the strong guarantee of atomic data operations. Relink allows SplitFS to implement a form of localized copy-on-write. Due to the staging files being pre-allocated, locality is preserved to an extent. SplitFS logs metadata operations to ensure they are atomic and synchronous. Optimized logging ensures that for most operations exactly one cache line is written and one `sfence` is issued for logging.

3.4 Handling Reads, Overwrites, and Appends

Reads. Reads consult the collection of mmaps to determine where the most recent data for this offset is, since the data could have been overwritten or appended (and thus in a staging file). If a valid memory mapped region for the offsets being read exists in U-Split, the read is serviced from the corresponding region. If such a region does not exist, then the 2 MB region surrounding the read offset is first memory mapped, added to the the collection of mmaps, and then the read operation is serviced using processor loads.

Overwrites. Similar to reads, if the target offset is already memory mapped, then U-Split services the overwrite using non-temporal store instructions. If the target offset is not memory mapped, then the 2MB region surrounding the offset is first memory mmaped, added to the collection of mmaps, and then the overwrite is serviced. However, in strict mode, to guarantee atomicity, overwrites are first redirected to a staging file (even if the offset is memory mapped), then the operation is logged, and finally relinked on a subsequent `fsync()` or `close()`.

Appends. SplitFS redirects all appends to a staging file, and performs a relink on a subsequent `fsync()` or `close()`. As with overwrites, appends are performed with non-temporal writes and in strict mode, SplitFS also logs details of the append operation to ensure atomicity.

3.5 Implementation

We implement SplitFS as a combination of a user-space library file system (9K lines of C code) and a small patch to ext4 DAX to add the relink system call (500 lines of C code). SplitFS supports 35 common POSIX calls, such as `pwrite()`, `pread64()`, `fread()`, `readv()`, `ftruncate64()`, `openat()`, etc; we found that supporting this set of calls is sufficient to support a variety of applications and micro-benchmarks. Since PM file systems PMFS and NOVA are supported by Linux kernel version 4.13, we modified 4.13 to support SplitFS. We now present other details of our implementation.

Intercepting POSIX calls. SplitFS uses `LD_PRELOAD` to intercept POSIX calls and either serve from user-space or route them to the kernel after performing some book-keeping tasks. Since SplitFS intercepts calls at the POSIX level in `glibc` rather than at the system call level, SplitFS has to intercept several variants of common system calls like `write()`.

Relink. We implement relink by leveraging an `ioctl` provided by ext4 DAX. The `EXT4_IOC_MOVE_EXT ioctl` swaps

extents between a source file and a destination file, and uses journaling to perform this atomically. The `ioctl` also de-allocates blocks in the target file if they are replaced by blocks from the source file. By default, the `ioctl` also flushes the swapped data in the target file; we modify the `ioctl` to only touch metadata, without copying, moving, or persisting of data. We also ensure that after the swap has happened, existing memory mappings of both source and destination files are valid; this is vital to SplitFS performance as it avoids page faults. The `ioctl` requires blocks to be allocated at both source and destination files. To satisfy this requirement, when handling appends via relink, we allocate blocks at the destination file, swap extents from the staging file, and then de-allocate the blocks. This allows us to perform relink without using up extra space, and reduces implementation complexity at the cost of temporary allocation of data.

Handling file open and close. On file open, SplitFS performs `stat()` on the file and caches its attributes in user-space to help handle later calls. When a file is closed, we do not clear its cached information. When the file is unlinked, all cached metadata is cleared, and if the file has been memory-mapped, it is un-mapped. The cached attributes are used to check file permissions on every subsequent file operation (*e.g.,* `read()`) intercepted by U-Split.

Handling fork. Since SplitFS uses a user-space library file system, special care needs to be taken to handle `fork()` and `execve()` correctly. When `fork()` is called, SplitFS is copied into the address space of the new process (as part of copying the address space of the parent process), so that the new process can continue to access SplitFS.

Handling execve. `execve()` overwrites the address space, but open file descriptors are expected to work after the call completes. To handle this, SplitFS does the following: before executing `execve()`, SplitFS copies its in-memory data about open files to a shared memory file on `/dev/shm`; the file name is the process ID. After executing `execve()`, SplitFS checks the shared memory device and copies information from the file if it exists.

Handling dup. When a file descriptor is duplicated, the file offset is changed whenever operations are performed on either file descriptor. SplitFS handles by maintaining a single offset per open file, and using pointers to this file in the file descriptor maintained by SplitFS. Thus, if two threads `dup` a file descriptor and change the offset from either thread, SplitFS ensures both threads see the changes.

Staging files. SplitFS pre-allocates staging files at startup, creating 10 files each 160 MB in size. Whenever a staging file is completely utilized, a background thread wakes up and creates and pre-allocates a new staging file. This avoids the overhead of creating staging files in the critical path.

Cache of memory-mappings. SplitFS caches all memory-mappings its creates in its collection of memory mappings. A memory-mapping is only discarded on `unlink()`. This reduces the cost of setting up memory mappings in the critical path on read or write.

Multi-thread access. SplitFS uses a lock-free queue for managing the staging files. It uses fine-grained reader-writer locks to protect its in-memory metadata about open files, inodes, and memory-mappings.

3.6 Tunable Parameters

SplitFS provides a number of tunable parameters that can be set by application developers and users for each U-Split instance. These parameters affect the performance of SplitFS.

`mmap()` size. SplitFS supports a configurable size of `mmap()` for handling overwrites and reads. Currently, SplitFS supports `mmap()` sizes ranging from 2MB to 512MB. The default size is 2 MB, allowing SplitFS to employ huge pages while pre-populating the mappings.

Number of staging files at startup. There are ten staging files at startup by default; when a staging file is used up, SplitFS creates another staging file in the background. We experimentally found that having ten staging files provides a good balance between application performance and the initialization cost and space usage of staging files.

Size of the operation log. The default size of the operation log is 128MB for each U-Split instance. Since all log entries consist of a single cacheline in the common case, SplitFS can support up to 2M operations without clearing the log and re-initializing it. This helps applications with small bursts to achieve good performance while getting strong semantics.

3.7 Security

SplitFS does not expose any new security vulnerabilities as compared to an in-kernel file system. All metadata operations are passed through to the kernel which performs security checks. SplitFS does not allow a user to open, read, or write a file to which they previously did not have permissions. The U-Split instances are isolated from each other in separate processes; therefore applications cannot access the data of other applications while running on SplitFS. Each U-Split instance only stores book-keeping information in DRAM for the files that the application already has access to. An application that uses SplitFS may corrupt its own files, just as in an in-kernel file system.

4 Discussion

We reflect on our experiences building SplitFS, describe problems we encountered, how we solved them, and surprising insights that we discovered.

Page faults lead to significant cost. SplitFS memory maps files before accessing them, and uses `MAP_POPULATE` to pre-fault all pages so that later reads and writes do not incur page-fault latency. As a result, we find that a significant portion of the time for `open()` is consumed by page faults. While

the latency of device IO usually dominates page fault cost in storage systems based on solid state drives or magnetic hard drives, the low latency of persistent memory highlights the cost of page faults.

Huge pages are fragile. A natural way of minimizing page faults is to use 2 MB huge pages. However, we found huge pages fragile and hard to use. Setting up a huge-page mapping in the Linux kernel requires a number of conditions. First, the virtual address must be 2 MB aligned. Second, the physical address on PM must be 2 MB aligned. As a result, fragmentation in either the virtual address space or the physical PM prevents huge pages from being created. For most workloads, after a few thousand files were created and deleted, fragmenting PM, we found it impossible to create any new huge pages. Our collection-of-mappings technique sidesteps this problem by creating huge pages at the beginning of the workload, and reusing them to serve reads and writes. Without huge pages, we observed read performance dropping by 50% in many workloads. We believe this is a fundamental problem that must be tackled since huge pages are crucial for accessing large quantities of PM.

Avoiding work in the critical path is important. Finally, we found that a general design technique that proved crucial for SplitFS is simplifying the critical path. We pre-allocate wherever possible, and use a background thread to perform pre-allocation in the background. Similarly, we pre-fault memory mappings, and use a cache to re-use memory mappings as much as possible. SplitFS rarely performs heavyweight work in the critical path of a data operation. Similarly, even in strict mode, SplitFS optimizes logging, trading off shorter recovery time for a simple, low overhead logging protocol. We believe this design principle will be useful for other systems designed for PM.

Staging writes in DRAM. An alternate design that we tried was staging writes in DRAM instead of on PM. While DRAM staging files incur less allocation costs than PM staging files, we found that the cost of copying data from DRAM to PM on `fsync()` overshadowed the benefit of staging data in DRAM. In general, DRAM buffering is less useful in PM systems because PM and DRAM performances are similar.

Legacy applications need to be rewritten to take maximum advantage of PM. We observe that the applications we evaluate such as LevelDB spent a significant portion of their time (60 – 80%) performing POSIX calls on current PM file systems. SplitFS is able to reduce this percentage down to 46-50%, but further reduction in software overhead will have negligible impact on application runtime since the majority of the time is spent on application code. Applications would need to be rewritten from scratch to use libraries like `libpmem` that exclusively operate on data structures in `mmap()` to take further advantage of PM.

Application	Description
TPC-C [9] on SQLite [26]	Online transaction processing
YCSB [8] on LevelDB [15]	Data retreival & maintenance
Set in Redis [1]	In-memory data structure store
Git	Popular version control software
Tar	Linux utility for data compression
Rsync	Linux utility for data copy

Table 5. Applications used in evaluation. The table provides a brief description of the real-world applications we use to evaluate PM file systems.

5 Evaluation

In this section, we use a number of microbenchmarks and applications to evaluate SplitFS in relation to state-of-the-art PM filesystems like ext4 DAX, NOVA, and PMFS. While comparing these different file systems, we seek to answer the following questions:

- How does SplitFS affect the performance of different system calls as compared to ext4 DAX? (§5.4)
- How do the different techniques employed in SplitFS contribute to overall performance? (§5.5)
- How does SplitFS compare to other file systems for different PM access patterns? (§5.6)
- Does SplitFS reduce file-system software overhead as compared to other PM file systems? (§5.7)
- How does SplitFS compare to other file systems for real-world applications? (§5.8 & §5.9)
- What are the compute and storage overheads incurred when using SplitFS? (§5.10)

We first briefly describe our experimental methodology (§5.1 & §5.2) before addressing each of the above questions.

5.1 Experimental Setup

We evaluate the performance of SplitFS against other PM file systems on Intel Optane DC Persistent Memory Module (PMM). The experiments are performed on a 2-socket, 96-core machine with 768 GB PMM, 375 GB DRAM, and 32 MB Last Level Cache (LLC). We run all evaluated file systems on the 4.13 version of the Linux kernel (Ubuntu 16.04). We run each experiment multiple times and report the mean. In all cases, the standard deviation was less than five percent of the mean, and the experiments could be reliably repeated.

5.2 Workloads

We used two key-value stores (Redis, LevelDB), an embedded database (SQLite), and three utilities (tar, git, rsync) to evaluate the performance of SplitFS. Table 5 lists the applications and their characteristics.

TPC-C on SQLite. TPC-C is an online transaction processing benchmark. It has five different types of transactions each with different ratios of reads and writes. We run SQLite v3.23.1 with SplitFS, and measured the performance of TPC-C on SQLite in the Write-Ahead-Logging (WAL) mode.

YCSB on LevelDB. The Yahoo Cloud Serving Benchmark [8] has six different key-value store benchmarks, each with different read/write ratios. We run the YCSB workloads on the LevelDB key-value stores. We set the `sstable` size to 64 MB as recommended in Facebook's tuning guide [13].

Redis. We set 1M key-value pairs in Redis [1], an in-memory key-value store. We ran Redis in the Append-Only-File mode, where it logs updates to the database in a file and performs `fsync()` on the file every second.

Utilities. We also evaluated the performance of SplitFS for tar, git, and rsync. With git, we measured the time taken for `git add` and `git commit` of all files in the Linux kernel ten times. With rsync, we copy a 7 GB dataset of 1200 files with characteristics similar to backup datasets [30] from one PM location to another. With tar, we compressed the Linux kernel 4.18 along with the files from the backup dataset.

5.3 Correctness and recovery

Correctness. First, to validate the functional correctness of SplitFS we run various micro-benchmarks and real-world applications and compare the resulting file-system state to the ones obtained with ext4 DAX. We observe that the file-system states obtained with ext4 DAX and SplitFS are equivalent, validating how SplitFS handles POSIX calls in its user-space library file system.

Recovery times. Crash recovery in POSIX and sync modes of SplitFS do not require anything beyond allowing the underlying ext4 DAX file system to recover. In strict mode however, all valid log entries in the operation log need to be replayed on top of ext4 DAX recovery. This additional log replay time depends on the number and type of valid log entries in the log. To estimate the additional time needed for recovery, we crash our real-world workloads at random points in their execution and measure the log replay time. In our crash experiments, the maximum number of log entries to be replayed was 18,000 and that took about 3 seconds on emulated PM (emulation details in §5.8). In a worst-case micro-benchmark where we perform cache-line sized writes and crash with 2M (128MB of data) valid log entries, we observed a log replay time of 6 seconds on emulated PM.

5.4 SplitFS system call overheads

The central premise of SplitFS is that it is a good trade-off to accelerate data operations at the expense of metadata operations. Since data operations are more prevelant, this optimization improves overall application performance. To validate this premise, we construct a micro-benchmark similar to FileBench Varmail [27] that issues a variety of data and metadata operations. The micro-benchmark first creates and appends 16KB to a file (as four appends, each followed by an `fsync()`), closes it, opens it again, read the whole file as one read call, closes it, then opens and closes the file once more, and finally deletes the file. The multiple open and close calls were introduced to account for the fact that their latency varies over time. Opening a file for the first time takes longer than opening a file that we recently closed, due to file metadata caching inside U-Split. Table 6 shows the latencies we observed for different system calls and they are reported for all the three modes provided by SplitFS and for ext4 DAX on which SplitFS was built.

System call	Strict	Sync	POSIX	ext4 DAX
open	2.09	2.08	1.82	1.54
close	0.78	0.69	0.69	0.34
append	3.14	3.09	2.84	11.05
fsync	6.85	6.80	6.80	28.98
read	4.57	4.53	4.53	5.04
unlink	14.60	13.56	14.33	8.60

Table 6. SplitFS system call overheads. The table compares the latency (in us) of different system calls for various modes of SplitFS and ext4 DAX.

We make three observations based on these results. First, data operations on SplitFS are significantly faster than on ext4 DAX. Writes especially are 3–4× faster. Second, metadata operations (*e.g.,* `open()`, `close()`, etc.) are slower on SplitFS than on ext4 DAX, as SplitFS has to setup its own data structures in addition to performing the operation on ext4 DAX. In SplitFS, `unlink()` is an expensive operation because the file mappings that are created for serving reads and overwrites need to be unmapped in the `unlink()` wrapper. Third, as the consistency guarantees provided by SplitFS get stronger, the syscall latency generally increases. This increase can be attributed to more work SplitFS has to do (*e.g.,* logging in strict mode) for each system call to provide stronger guarantees. Overall, SplitFS achieves its objective of accelerating data operations albeit at the expense of metadata operations.

5.5 SplitFS performance breakdown

We examine how the various techniques employed by SplitFS contribute to overall performance. We use two write-intensive microbenchmarks: sequential 4KB overwrites and 4KB appends. An `fsync()` is issued every ten operations. Figure 3 shows how individual techniques introduced one after the other improve performance.

Sequential overwrites. SplitFS increases sequential overwrite performance by more than 2× compared to ext4 DAX

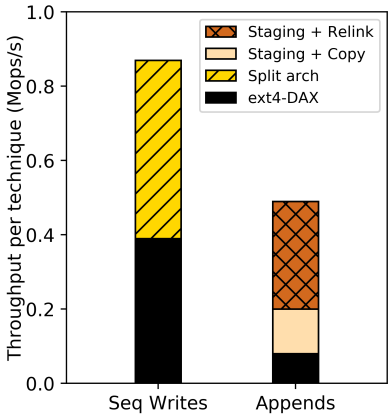

Figure 3. SplitFS techniques contributions. This figure shows the contributions of different techniques to overall performance. We compare the relative merits of these techniques using two write intensive microbenchmarks; sequential overwrites and appends.

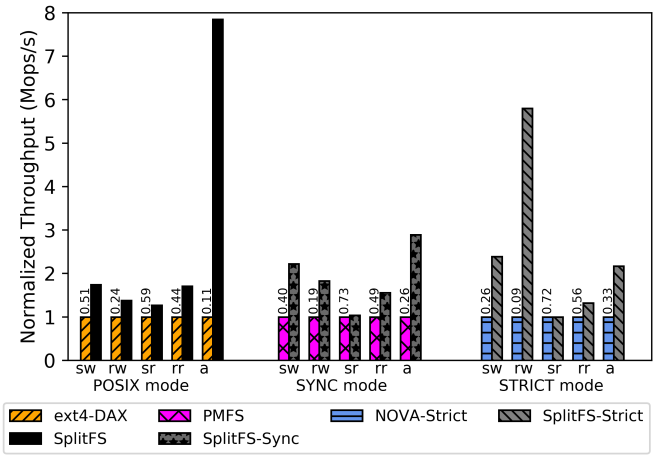

Figure 4. Performance on different IO patterns. This figure compares SplitFS with the state-of-the-art PM file systems in their respective modes using micrbenchmarks that perform five different kinds of file access patters. The y-axis is throughput normalized to ext4 DAX in POSIX mode, PMFS in sync mode, and NOVA-Strict in Strict mode (higher is better). The absolute throughput numbers in Mops/s are given over the baseline in each group.

since overwrites are served from user-space via processor stores. However, further optimizations like handling appends using staging files and relink have negligible impact on this workload as it does not issue any file append operations.

Appends. The split architecture does not accelerate appends since without staging files or relink all appends go to ext4 DAX as they are metadata operations. Just introducing staging files to buffer appends improves performance by about 2×. In this setting, even though appends are serviced in user-space, overall performance is bogged down by expensive data copy operations on `fsync()`. Introducing the relink primitive to this setting eliminates data copies and increases application throughput by 5×.

5.6 Performance on different IO patterns

To understand the relative merits of different PM file systems, we compare their performance on microbenchmarks performing different file IO patterns: sequential reads, random reads, sequential writes, random writes, and appends. Each benchmark reads/writes an entire 128MB file in 4KB operations. We compare file systems providing the same guarantees: SplitFS-POSIX with ext4 DAX, SplitFS-sync with PMFS, and SplitFS-strict with Nova-strict and Strata. Figure 4 captures the performance of these file systems for the different micro-benchmarks.

POSIX mode. SplitFS is able to reduce the execution times of ext4 DAX by at least 27% and as much as 7.85× (sequential reads and appends respectively). Read-heavy workloads present fewer improvement opportunities for SplitFS as file read paths in the kernel are optimized in modern PM file systems. However, write paths are much more complex and longer, especially for appends. So, servicing a write in user-space has a higher payoff than servicing a read, an observation we already made in Table 6.

Sync mode. Compared to PMFS, SplitFS improves the performance for write workloads (by as much as 2.89×) and increases performance for read workloads (by as much as 56%). Similar to ext4 DAX, SplitFS's ability to not incur expensive write system calls translates to its superior performance for the write workloads.

Strict mode. NOVA, Strata, and SplitFS in this mode provide atomicity guarantees to all operations and perform the necessary logging. As can be expected, the overheads of logging result in reduced performance compared to file systems in other modes. Overall, SplitFS improves the performance over NOVA by up to 5.8× on the random writes workload. This improvement stems from SplitFS's superior logging which incurs half the number of log writes and fence operations than NOVA.

5.7 Reducing software overhead

The central premise of SplitFS is that it is possible to accelerate applications by reducing file system software overhead. We define file-system software overhead as the time taken to service a file-system call minus the time spent actually accessing data on the PM device. For example, if a system call takes 100 μs to be serviced, of which only 25 μs were spent read or writing to PM, then we say that the software overhead is 75 μs. To provide another example, for appending 4 KB (which takes 10 μs to write to PM), if file system A writes 10 metadata items (incurring 100 μs) while file system

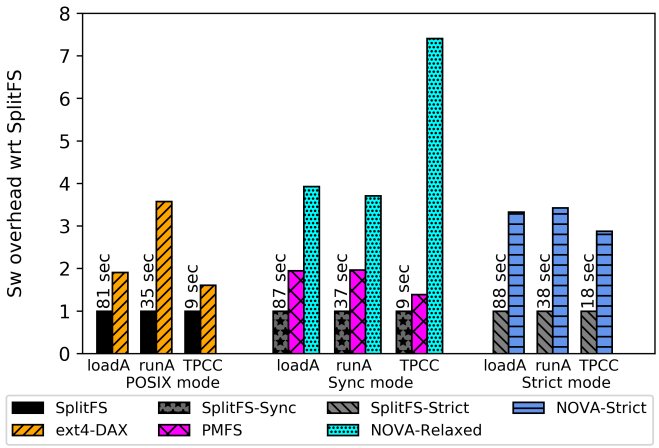

Figure 5. Software overhead in applications. This figure shows the relative file system software overhead incurred by different applications with various file systems as compared SplitFS providing the same level of consistency guarantees (lower is better). The numbers shown indicate the absolute time taken to run the workload for the baseline file system.

Workload	Strata	SplitFS
Load A	29.1 kops/s	1.73×
Run A	55.2 kops/s	1.76×
Run B	76.8 kops/s	2.16×
Run C	94.3 kops/s	2.14×
Run D	113.1 kops/s	2.25×
Load E	29.1 kops/s	1.72×
Run E	8.1 kops/s	2.03×
Run F	73.3 kops/s	2.25×

Table 7. SplitFS vs. Strata. This table compares the performance of Strata and SplitFS strict running YCSB on LevelDB. We present the raw throughput numbers for Strata and normalized SplitFS strict throughput w.r.t Strata. This is the biggest workload that we could run reliably on Strata.

B writes two metadata items (incurring 20 μs), file-system B will have lower overhead. In addition to avoiding kernel traps of system calls, the different techniques discussed in §3 help SplitFS reduce its software overhead. Minimizing software overhead allows applications to fully leverage PMs.

Figure 5 highlights the relative software overheads incurred by different file systems compared to SplitFS providing the same level of guarantees. We present results for three write-heavy workloads, LevelDB running YCSB Load A and Run A, and SQLite running TPCC. ext4 DAX and NOVA (in relaxed mode) suffer the highest relative software overheads, up to 3.6× and 7.4× respectively. NOVA-Relaxed incurs the highest software overhead for TPCC because it has to update the per-inode logical log entries on overwrites before updating the data in-place. On the other hand, SplitFS-sync can directly perform in-place data updates, and thus has significantly lower software overhead. PMFS suffers the lowest relative software overhead, capping off at 1.9× for YCSB Load A and Run A. Overall, SplitFS incurs the lowest software overhead.

5.8 Performance on data-intensive workloads

Figure 6 summarizes the performance of various applications on different file systems. The performance metric we use for these data intensive workloads (LevelDB with YCSB, Redis with 100% writes, and SQLite with TPCC) is throughput measured in KOps/s. For each mode of consistency guarantee (POSIX, sync, and strict), we compare SplitFS to state-of-the-art PM file systems. We report the absolute performance for the baseline file system in each category and relative throughput for SplitFS. Despite our best efforts, we were not able to run Strata on these large applications; other researchers have also reported problems in evaluating Strata [32]. We evaluated Strata with a smaller-scale YCSB workload using a 20GB private log.

Overall, SplitFS outperforms other PM file systems (when providing similar consistency guarantees) on all data-intensive workloads by as much as 2.70×. We next present a breakdown of these numbers for different guarantees.

POSIX mode. SplitFS outperforms ext4 DAX in all workloads. Write-heavy workloads like RunA (2×), LoadA (89%), LoadE (91%), Redis (27%), etc. benefit the most with SplitFS. SplitFS speeds up writes and appends the most, so write-heavy workloads benefit the most from SplitFS. SplitFS outperforms ext4 DAX on read-dominated workloads, but the margin of improvement is lower.

Sync and strict mode. SplitFS outperforms sync-mode file systems PMFS and NOVA (relaxed) and strict-mode file system NOVA (strict) for all the data intensive workloads. Once again, its the write-heavy workloads that show the biggest boost in performance. For example, SplitFS in sync mode outperforms NOVA (relaxed) and PMFS by 2× and 30% on RunA and in strict mode outperforms NOVA (strict) by 2×. Ready-heavy workloads on the other hand do not show much improvement in performance.

Comparison with Strata. We were able to reliably evaluate Strata (employing a 20 GB private log) using LevelDB running smaller-scale YCSB workloads (1M records, and 1M ops for workloads A–D and F, 500K ops for workload E). We were unable to run Strata on Intel DC Persistent Memory. Hence, we use DRAM to emulate PM. We employ the same PM emulation framework used by Strata. We inject a delay of 220ns on every `read()` system call, to emulate the access latencies of the PM hardware. We do not add this fixed 220ns delay for writes, because writes do not go straight to PM

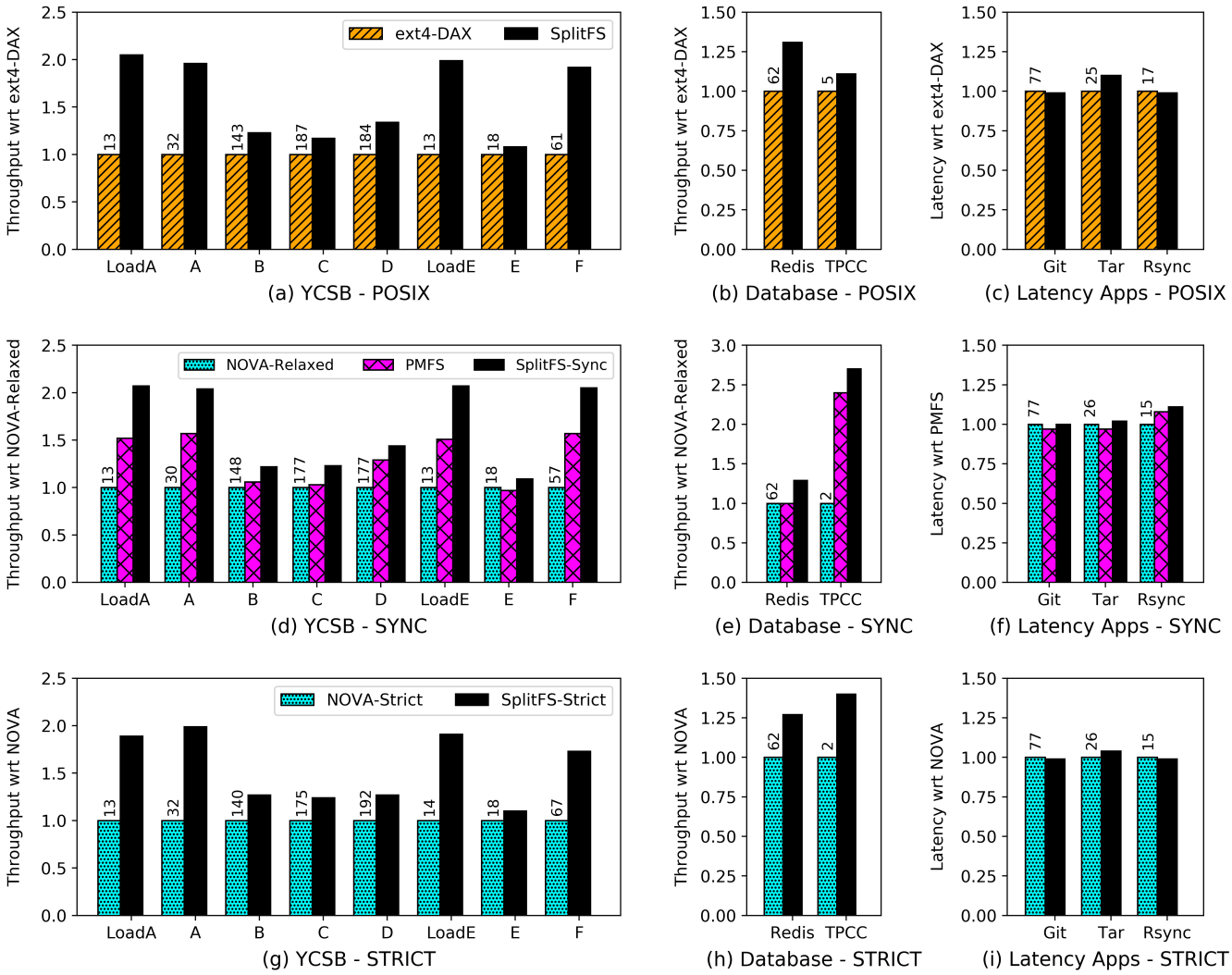

Figure 6. Real application performance. This figure shows the performance of both data intensive applications (YCSB, Redis, and TPCC) and metadata instensive utilities (git, tar, and rsync) with different file systems, providing three different consistency guarantees, POSIX, sync, and strict. Overall, SplitFS beats all other file systems on all data intensive applications (in their respective modes) while incurring minor performance degradation on metadata heavy workloads. For throughput workloads, higher is better. For latency workloads, lower is better. The numbers indicate the absolute throughput in Kops/s or latency in seconds for the base file system.

in the critical path, but only to the memory controller. We add bandwidth-modeling delays for reads as well as writes to emulate a memory device with $1/3^{rd}$ the bandwidth of DRAM, an expected characteristic of PMs [18]. While this emulation approach is far from perfect, we observe that the resulting memory access characteristics are inline with the expected behavior of PMs [19]. SplitFS outperforms Strata on all workloads, by 1.72×–2.25× as shown in Table 7.

5.9 Performance on metadata-heavy workloads

Fig 6 compares the performance of SplitFS with other PM file systems (we only show the best performing PM file system) on metadata-heavy workloads like git, tar, and rsync. These metadata-heavy workloads do not present many opportunities for SplitFS to service system calls in userspace and in turn slow metadata operations down due to the additional bookkeeping performed by SplitFS. These workloads represent the worst case scenarios for SplitFS. The maximum overhead experienced by SplitFS is 13%.

5.10 Resource Consumption

SplitFS consumes memory for its file-related metadata (*e.g.*,, to keep track of open file descriptors, staging files used). It also additionally consumes CPU time to execute background threads that help with metadata management and to move some expensive tasks off the application's critical path.

Memory usage. SplitFS using a maximum of 100MB to maintain its own metadata to help track different files, the mappings between file offsets and `mmap()`-ed regions, etc. In strict mode, SplitFS additionally uses 40MB to maintain data structures to provide atomicity guarantees.

CPU utilization. SplitFS uses a background thread to handle various deferred tasks (*e.g.*,, stage file allocation, file closures). This thread utilizes one physical thread of the machine, occasionally increasing CPU consumption by 100%.

6 Related Work

SplitFS builds on a large body of work on PM file systems and building low-latency storage systems. We briefly describe the work that is closest to SplitFS.

Aerie. Aerie [28] was one of the first systems to advocate for accessing PM from user-space. Aerie proposed a split architecture similar to SplitFS, with a user-space library file system and a kernel component. Aerie used a user-space metadata server to hand out leases, and only used the kernel component for coarse-grained activities like allocation. In contrast, SplitFS does not use leases (instead making most operations immediately visible) and uses ext4 DAX as its kernel component, passing all metadata operations to the kernel. Aerie proposed eliminating the POSIX interface, and aimed to provide applications flexibility in interfaces. In contrast, SplitFS aims to efficiently support the POSIX interface.

Strata. The Strata [19] cross-device file system is similar to Aerie and SplitFS in many respects. There are two main differences from SplitFS. First, Strata writes all data to a process-private log, coalesces the data, and then writes it to a shared space. In contrast, only appends are private (and only until `fsync`) in SplitFS; all metadata operations and overwrites are immediately visible to all processes in SplitFS. SplitFS does not need to copy data between a private space and a shared space; it instead relinks data into the target file. Finally, since Strata is implemented entirely in user-space, the authors had to re-implement a lot of VFS functionality in their user-space library. SplitFS instead depends on the mature codebase of ext4 DAX for all metadata operations.

Quill and FLEX. Quill [11] and File Emulation with DAX (FLEX) [32] both share with SplitFS the core technique of transparently transforming read and overwrite POSIX calls into processor loads and stores. However, while Quill and FLEX do not provide strong semantics, SplitFS can provide applications with synchronous, atomic operations if required. SplitFS also differs in its handling of appends. Quill calls into the kernel for every operation, and FLEX optimizes appends by pre-allocating data beyond what the application asks for. In contrast, SplitFS elegantly handles this problem using staging files and the relink primitive. While Quill appends are slower than ext4 DAX, SplitFS appends are faster than ext4 DAX appends. At the time of writing this paper, FLEX has not been made open-source, so we could not evaluate it.

PM file systems. Several file systems such as SCMFS [31], BPFS [7], and NOVA [33] have been developed specifically for PM. While each file system tries to reduce software overhead, they are unable to avoid the cost of trapping into the kernel. The relink primitive from SplitFS is similar to the short-circuit paging presented in BPFS. However, while short-circuit paging relies on an atomic 8-byte write, SplitFS relies on ext4's journaling mechanism to make relink atomic.

Kernel By-Pass. Several projects have advocated direct user-space access to networking [29], storage [5, 10, 14, 20], and other hardware features [3, 4, 22]. These projects typically follow the philosophy of separating the control path and data path, as in Exokernel [12] and Nemesis [2]. SplitFS follows this philosophy, but differs in the abstraction provided by the kernel component; SplitFS uses a PM file system as its kernel component to handle all metadata operations, instead of limiting it to lower-level decisions like allocation.

7 Conclusion

We present SplitFS, a PM file system built using the split architecture. SplitFS handles data operations entirely in user-space, and routes metadata operations through the ext4 DAX PM file system. SplitFS provides three modes with varying guarantees, and allows applications running at the same time to use different modes. SplitFS only requires adding a single system call to the ext4 DAX file system. Evaluating SplitFS with micro-benchmarks and real applications, we show that it outperforms state-of-the-art PM file systems like NOVA on many workloads. The design of SplitFS allows users to benefit from the maturity and constant development of the ext4 DAX file system, while getting the performance and strong guarantees of state-of-the-art PM file systems. SplitFS is publicly available at https://github.com/utsaslab/splitfs.

Acknowledgments

We would like to thank our shepherd, Keith Smith, the anonymous reviewers, and members of the LASR group and the Systems and Storage Lab for their feedback and guidance. We would like to thank Intel and ETRI IITP/KEIT[2014-3-00035] for providing access to Optane DC Persistent Memory for conducting experiments for the paper. This work was supported by NSF CAREER grant 1751277 and generous donations from VMware, Google and Facebook. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of other institutions.

References

[1] 2019. Redis: In-memory data structure store. https://redis.io.

[2] Paul R. Barham. 1997. A fresh approach to file system quality of service. In *Proceedings of 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97)*. IEEE, 113–122.

[3] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 335–348. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

[4] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[5] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. 2012. Providing safe, user space access to fast, solid state disks. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*. 387–400. https://doi.org/10.1145/2150976.2151017

[6] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*. San Jose, California, 101–116.

[7] 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.

[8] 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.

[9] Transaction Processing Performance Council. 2001. TPC benchmark C, Standard Specification Version 5.

[10] Matt DeBergalis, Peter F. Corbett, Steven Kleiman, Arthur Lent, Dave Noveck, Thomas Talpey, and Mark Wittle. 2003. The Direct Access File System. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*. http://www.usenix.org/events/fast03/tech/debergalis.html

[11] Louis Alex Eisner, Todor Mollov, and Steven J. Swanson. 2013. *Quill: Exploiting fast non-volatile memory by transparently bypassing the file system.* Department of Computer Science and Engineering, University of California, San Diego.

[12] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. 251–266. https://doi.org/10.1145/224056.224076

[13] Facebook. 2017. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[14] Garth A. Gibson, David Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. 1997. File Server Scaling with Network-Attached Secure Disks. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Seattle, Washington, USA, June 15-18, 1997*. 272–284. https://doi.org/10.1145/258612.258696

[15] Google. 2019. LevelDB. https://github.com/google/leveldb.

[16] Intel Corporation. 2019. Platform brief Intel Xeon Processor C5500/C3500 Series and Intel 3420 Chipset. https://www.intel.com/content/www/us/en/intelligent-systems/picket-post/embedded-intel-xeon-c5500-processor-series-with-intel-3420-chipset.html.

[17] Intel Corporation. 2019. Revolutionary Memory Technology. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html.

[18] 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

[19] 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*. ACM, 460–477. https://doi.org/10.1145/3132747.3132770

[20] Edward K. Lee and Chandramohan A. Thekkath. 1996. Petal: Distributed Virtual Disks. In *ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts, USA, October 1-5, 1996*. 84–92. https://doi.org/10.1145/237090.237157

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

[22] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas E. Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. 1–16. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[23] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*. 206–220. https://doi.org/10.1145/1095810.1095830

[24] 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

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

[26] SQLite. 2019. SQLite transactional SQL database engine. http://www.sqlite.org/.

[27] 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.

[28] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*.

[29] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. 1995. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. 40–53. https://doi.org/10.1145/224056.224061

[30] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. 2012. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*. 33–48. https://www.usenix.org/conference/fast12/characteristics-backup-workloads-production-systems

[31] XiaoJian Wu, Sheng Qiu, and A. L. Narasimha Reddy. 2013. SCMFS: A File System for Storage Class Memory and its Extensions. *TOS* 9, 3 (2013), 7:1–7:23. https://doi.org/10.1145/2501620.2501621

[32] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. 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, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. 427–439. https://doi.org/10.1145/3297858.3304077

[33] 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