Strata: A Cross Media File System

Youngjin Kwon
The University of Texas at Austin

Henrique Fingler
The University of Texas at Austin

Tyler Hunt
The University of Texas at Austin

Simon Peter
The University of Texas at Austin

Emmett Witchel
The University of Texas at Austin

Thomas Anderson
University of Washington

ABSTRACT

Current hardware and application storage trends put immense pressure on the operating system's storage subsystem. On the hardware side, the market for storage devices has diversified to a multi-layer storage topology spanning multiple orders of magnitude in cost and performance. Above the file system, applications increasingly need to process small, random IO on vast data sets with low latency, high throughput, and simple crash consistency. File systems designed for a single storage layer cannot support all of these demands together.

We present Strata, a cross-media file system that leverages the strengths of one storage media to compensate for weaknesses of another. In doing so, Strata provides performance, capacity, and a simple, synchronous IO model all at once, while having a simpler design than that of file systems constrained by a single storage device. At its heart, Strata uses a log-structured approach with a novel split of responsibilities among user mode, kernel, and storage layers that separates the concerns of scalable, high-performance persistence from storage layer management. We quantify the performance benefits of Strata using a 3-layer storage hierarchy of emulated NVM, a flash-based SSD, and a high-density HDD. Strata has 20-30% better latency and throughput, across several unmodified applications, compared to file systems purpose-built for each layer, while providing synchronous and unified access to the entire storage hierarchy. Finally, Strata achieves up to 2.8× better throughput than a block-based 2-layer cache provided by Linux's logical volume manager.

CCS CONCEPTS

• **Information systems** → **Hierarchical storage management**; **Storage class memory**; • **Software and its engineering** → **File systems management**;

SOSP'17, October 2017, Shanghai, China

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

KEYWORDS

File system, Non-volatile memory, Multi-layer storage

ACM Reference Format:
Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of ACM Symposium on Operating Systems Principles, Shanghai, China, October 2017 (SOSP'17),* 18 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

File systems are being stressed from below and from above. Below the file system, the market for storage devices has fragmented based on a tradeoff between performance and capacity, so that many systems are configured with local solid-state drives (SSD) and conventional hard disk drives (HDD). Non-volatile RAM (NVM) will soon add another device with a third distinct regime of capacity and performance.

Above the file system, modern applications demand performance and functionality far outside the traditional file system comfort zone, e.g., common case kernel bypass [41], small scattered updates to enormous datasets [12, 23], and programmer-friendly, efficient crash consistency [42]. It no longer makes sense to engineer file systems on the assumption that each file system is tied to a single type of physical device, that file operations are inherently asynchronous, or that the kernel must intermediate every metadata operation.

To address these issues, we propose Strata, an integrated file system across different storage media. To better leverage the hardware properties of multi-layer storage Strata has a novel split of user and kernel space responsibilities, storing updates to a user-level log in fast NVM, while asynchronously managing longer-term storage in the kernel. In so doing, Strata challenges some longstanding file system design ideas, while simplifying others. For example, Strata has no system-wide file block size and no individual layer implements a stand-alone file system. Although aspects of our design can be found in NVM [1, 25, 50, 52] and SSD [31, 32]-specific file systems, we are the first to design, build, and evaluate a file system that spans NVM, SSD, and HDD layers.

Closest to the application, Strata's user library synchronously logs process-private updates in NVM while reading from shared, read-optimized, kernel-maintained data and metadata. In the common case, the log holds updates to both user data

and file system metadata, ensuring correct file system behavior during concurrent access, even across failures. Client code uses the POSIX API, but Strata's synchronous updates obviate the need for any `sync`-related system calls and their attendant bugs [42]. Fast persistence and simple crash consistency are a perfect match for modern RPC-based systems that must persist data before responding to a network request [4, 12].

Strata tailors the format and management of data to improve the semantics and performance of the file system as data moves through Strata's layers. For example, operation logs are an efficient format for file system updates, but they are not efficient for reads. To reflect the centrality of data movement and reorganization in Strata, we coin the term *digest* for the process by which Strata reads file system data and metadata at one layer and optimizes the representation written to its next (typically slower and larger) layer. Digestion is periodic and asynchronous. Digestion allows a multitude of optimizations: it coalesces temporary durable writes (overwritten data or temporary files, e.g., due to complex application-level commit protocols [42]), it reorganizes and compacts data for efficient lookup, and it batches data into the large sequential operations needed for efficient writes to firmware-managed SSD and HDD.

Strata's first digest happens between a process-local log and the system-shared area, both stored in NVM. For example, a thread can create a file and a different thread can perform several small, sequential writes to it. The file create and the file data are logged in NVM. Each operation completes synchronously and in order. On a system crash, Strata recovers the latest version of the newly created file and its contents from the log. Eventually, Strata digests the log into an extent tree (optimized for reads) requiring physical-level file system work like block allocation for the new file. It will also merge the writes, eliminating any redundancy and creating larger data blocks. Strata resembles a log-structured merge (LSM) tree with a separate log at each layer, but a Strata digest changes the format and the invariants of its data much more than an LSM tree compaction. Strata minimizes re-writing overhead (§5.1) and increases performance significantly in cases where digestion can reduce the amount of work for the lower layer (e.g., a mail server can avoid copying 86% of its log §5.2). Strata currently also has limitations. For example, Strata reduces mean time to data loss (MTTDL) by assuming that all of its storage devices are reliable. It is optimized for applications requiring fast persistence for mostly non-concurrently shared data. Concurrent shared access requires kernel mediation. For more limitations see §4.1.

We implemented our Strata prototype within Linux. The Strata user-level library integrates seamlessly with glibc to provide unmodified applications compatibility with the familiar POSIX file IO interface. Our prototype is able to execute a wide range of applications, successfully completing all 201 unit tests of the LevelDB key-value store test suite, as well as all tests in the Filebench package. Microbenchmarks on real and emulated hardware show that, for small ($\leq$ 16 KB) IO sizes, Strata achieves up to 33% better write tail-latency and up to 7.8× better write throughput relative to the best performing alternative purpose-built NVM, SSD, and HDD file systems. Strata achieves 26% higher throughput than NOVA [52] on a mailserver workload in NVM, up to 27% lower latency than PMFS [25] on LevelDB, and up to 22% higher SET throughput than NOVA on Redis, while providing synchronous and unified access to the entire storage hierarchy. Finally, Strata achieves up to 2.8× better throughput than a block-based two-layer cache provided by Linux's logical volume manager. These performance wins are achieved without changing applications.

Starting with a discussion of the technical background (§2) for Strata's design, we then discuss its contributions.

- We present the Strata architecture (§3). We show how maintaining a user-level operation log in NVM and asynchronously *digesting* data among storage layers in the kernel leads to fast writes with efficient synchronous behavior, while optimizing for device characteristics and providing a unified interface to the entire underlying storage hierarchy.
- We implement a prototype of Strata on top of Linux that uses emulated NVM and commercially available SSDs and HDDs on a commodity server machine (§4).
- We quantify the performance and isolation benefits of providing a unified file system that manages all layers of the modern storage hierarchy simultaneously (§5).

2 BACKGROUND

We review current and near-future storage devices and discuss how Strata addresses this diversified market. We then discuss the demands of modern applications on the file system and how current alternatives fall short.

2.1 Hardware storage trends

Diversification. Storage technology is evolving from a single viable technology (that of the hard disk drive) into a diversified set of offerings that each fill a niche in the design tradeoff of cost, performance, and capacity. Three storage technologies stand out as stable contenders in the near-future: Non-volatile memory (NVM), solid state drives (SSDs), and high-density hard disk drives (HDDs). While HDDs and SSDs are already a commodity today, NVM is expected to be added in the future (Intel's 3D XPoint memory technology was released in March 2017, initially to accelerate SSDs [7, 11]).

Table 1 shows each technology and its expected long-term place in the design space. Latencies are from specifications while sequential read/write bandwidth for 4KB IO sizes are measured (see §5 for details). Prices are derived from the

Memory	Latency	Seq. R/W GB/s	$/GB
DRAM	100 ns	62.8	8.6
NVM	300 ns	7.8	4.0
SSD	10 μs	2.2 / 0.9	0.25
HDD	10 ms	0.1	0.02

Table 1: Memory technology latency, bandwidth, and $/GB as of April 2017. NVM numbers are projected (§5).

lowest device prices found via a Google Shopping search in April 2017. The NVM price is derived from the current price of Intel's 3D XPoint-based Optane SSD DC P4800X. NVM performance is based on a study [54]. Each storage technology offers a unique tradeoff of latency, throughput, and cost, with at least an order of magnitude difference relative to other technologies. This diversity suggests that future systems are likely to require several coexisting storage technologies.

Device management overhead. The physical characteristics of modern storage devices often prevent efficient update in place, even at a block level. SSDs have long needed a multi-block erasure before a physical block can be re-written; typical erasure region size has grown larger over time as vendors optimize for storage density and add lanes for high throughput. Although HDDs traditionally allowed efficient sector overwrites at the cost of a disk head seek, recently disks optimized for storage efficiency have adopted a shingle write pattern, similar to SSDs in that an entire region of disk sectors must be re-written to update any single sector.

To support legacy file systems, SSD and HDD device firmware maintains a persistent virtual to physical block translation; blocks are written sequentially at the physical level regardless of the virtual block write pattern. Depending on the write pattern, this can carry a high cost, where blocks are repeatedly moved and re-written to create empty regions for sequential writes on both SSDs and HDDs. On SSDs, this write amplification wears out the device more rapidly.

Even using a large block size is not enough to avoid the overhead when the disk is in steady-state. For example, using the Tang et al. methodology [46] on our testbed SSD (§5 for details), we observe a throughput slow-down of 12.2× in steady-state for 8 MiB blocks written randomly to a full disk. Similarly, write latency is inflated by a factor of 2.8× for 4 KiB random updates in steady state, and 10× for 128 KiB updates. When writing sequentially within erasure block boundaries, performance does not decline. Write amplification can also negatively affect IO tail latency and throughput isolation among applications, as the overhead is observed due to past use of the device, making it difficult to account performance costs to the originating application.

We leverage the multi-layer nature of Strata to achieve the full performance of the SSD and HDD layers, despite firmware management. Migration of blocks from NVM to SSD are made in full erasure block chunks (512 MiB on our testbed SSD); this is only possible because Strata coalesces data as it moves between persistent layers, with frequently updated data filtered by the NVM layer.

2.2 Application demands on the file system

Many modern applications need crash consistency for their files. The performance cost and complexity to achieve user-level crash consistency for files has grown over the past decade, with no relief in sight. Often, files are merely named address spaces that contain many internal objects with frequent, crash-consistent updates. Small, random writes are common on both desktop machines [26] and in the cloud through the use of key-value stores, data base backends, such as SQLite [15] and LevelDB [23], revision management systems, and distributed configuration software, such as Zookeeper [4]. On many file systems, efficient crash consistency for these applications is difficult and slow so many applications sacrifice correctness for performance [42].

Strata provides in-order file system semantics (including writes). This matches developer intuition [38] and simplifies crash recovery, but is usually considered too slow to be a practical goal for a file system. Given NVM devices, such semantics are now possible to provide efficiently [52].

2.3 Current alternatives are insufficient

Existing file systems specialize to a storage technology. Existing file systems make tradeoffs that are appropriate for a specific type of storage device; no single file system is appropriate across different storage media. For example, NOVA [52] and PMFS [25] require byte-addressability, limiting them to NVM; F2FS [32] uses multi-head logging and a buffer cache that are unnecessary on NVM. Strata is built to leverage the strengths of each storage device and compensate for weaknesses. By contrast, layering independent file systems on different media unnecessarily duplicates mechanisms, such as block and inode allocation, and lacks expressive inter-layer APIs. For example, block usage frequency and fragmentation information are not easily relayed across independent file systems (§5.3).

File system write amplification. As shown in §5, many file systems pad updates to a uniform block size (e.g., setting a bit in an in-use block bitmap will write an entire block), and file systems often require metadata writes to complete an update (e.g., a data write can update the file size in the inode). As with device-level write amplification, file system write amplification is often a major factor for application performance, especially for NVM devices that support efficient small writes. Using an operation log at the NVM layer that is later digested into block updates, Strata is able to efficiently aggregate repeated data and metadata updates, significantly lowering file system write amplification.

Block stores are not the only answer. Strata provides a file system rather than a block store interface to applications because of the file system's strong combination of backward compatibility, performance and functionality. The file system name space is a powerful persistent data structure with well understood properties (and limitations); its storage costs are moderate in time and space across a wide variety of access patterns; and it is used to share data by millions of applications and system tools. Multi-layer cloud-persistent block stores [3] face many of the same issues as Strata in managing migration of data across multiple devices, and can be appropriate for standalone applications that do their own block-level operations. We focus our design and evaluation on the unique opportunities provided by having a semantically rich view of application file system behavior.

3 STRATA DESIGN

The goal of Strata is to design a new file system that manages data across different storage devices, combining their strengths and compensating for their weaknesses. In particular, we have the following design goals for Strata.

- **Fast writes.** Strata must support fast, random, and small writes. An important motivation for fast small writes is supporting networked server applications which must persist data before issuing a reply. These applications form the backbone of modern cloud applications.
- **Efficient synchronous behavior.** Today's file systems create a usability and performance problem by guaranteeing persistence only in response to explicit programmer action (e.g., `sync`, `fsync`, `fdatasync`). File systems use a variety of complicated mechanisms (e.g., delayed allocation) to provide performance under the assumption of slow device persistence. Strata supports a superior, programmer-friendly model where file system operations persist in order, including synchronous `writes`, without sacrificing performance.
- **Manage write amplification.** Write amplification at the device and file system level have a first-order effect on performance, wear, and QoS. Examples include metadata updates for EXT4 and PMFS or copies introduced by the flash translation layer in SSDs [46]. Managing write amplification allows us to minimize its effect on performance and QoS. Managing write amplification is simpler once it is decoupled from the write fast-path.
- **High concurrency.** Strata supports concurrent logging from multiple threads in a single process using atomic operations. Logs from multiple processes can be digested in parallel within the kernel because logs are guaranteed to be independent (see §3.4).
- **Unified interface.** We provide a unified file system interface to all devices in the entire underlying storage hierarchy. Strata is backward compatible with existing POSIX applications but easily customizable since the API is provided entirely in a user-level library [41, 50].

Concept	Explanation
Update log	A per-process record of file system updates.
Shared area	Holds file system data in NVM, SSD, and HDD. Read-only for user code, written by the kernel.
File data cache	Read-only cache; caching data from SSD or HDD.
Update log pointers	An index into the update log; mapping file offsets to log blocks.
Strata transaction	A unit of durability; used for file system changes made by a system call.
Digest	Apply changes from an update log to the shared area.
Lease	Synchronizes updates to files and directories.

Table 2: Major concepts in Strata.

Strata's basic architecture resembles a log-structured merge (LSM) tree [39]. Strata first writes data synchronously to an operation log (*logging*) stored in NVM. Logging provides persistence with low and predictable latency, efficiently represents small updates, serializes operations in order, and supports data consistency, crash recovery and operation atomicity. Logs are highly desirable for writing, but are cumbersome to search and read. Thus, logs are periodically *digested* into a read-optimized tree format. In Strata, this format is based on per-file *extent trees* [35]. Digests happen asynchronously, and the log is garbage-collected. Table 2 summarizes major concepts in Strata, and Figure 1 shows a high-level overview of the Strata design which we now discuss.

Log at user-level, digest in the kernel. To attain fast writes, Strata separates the responsibilities of logging and digesting and assigns them to user-level software and the kernel, respectively. We call the user-level portion of Strata *LibFS*. Using leases to manage concurrent file accesses (explained in §3.4), the kernel grants LibFS direct access to a small private space in NVM for efficient logging of file system updates (the *update log*). The kernel also gives LibFS direct read-only access to portions of the shared extent tree space and data blocks (the *shared area*). Hardware, configured by the kernel, enforces access control [41].

The kernel-level file system (*KernelFS*) is responsible for digesting. Digesting is done in parallel across multiple threads for high throughput and runs asynchronously in the background. The update log is deep to allow the digest to batch log entries, amortizing and aggregating meta-data updates over an entire sequence of operations. KernelFS checks and enforces metadata integrity when digesting an application's log, such that when a digest completes, the digested data can become visible to other processes. Upon a crash, the kernel can recover file system state simply by re-digesting each application's remaining log. A log replay overwrites data structures

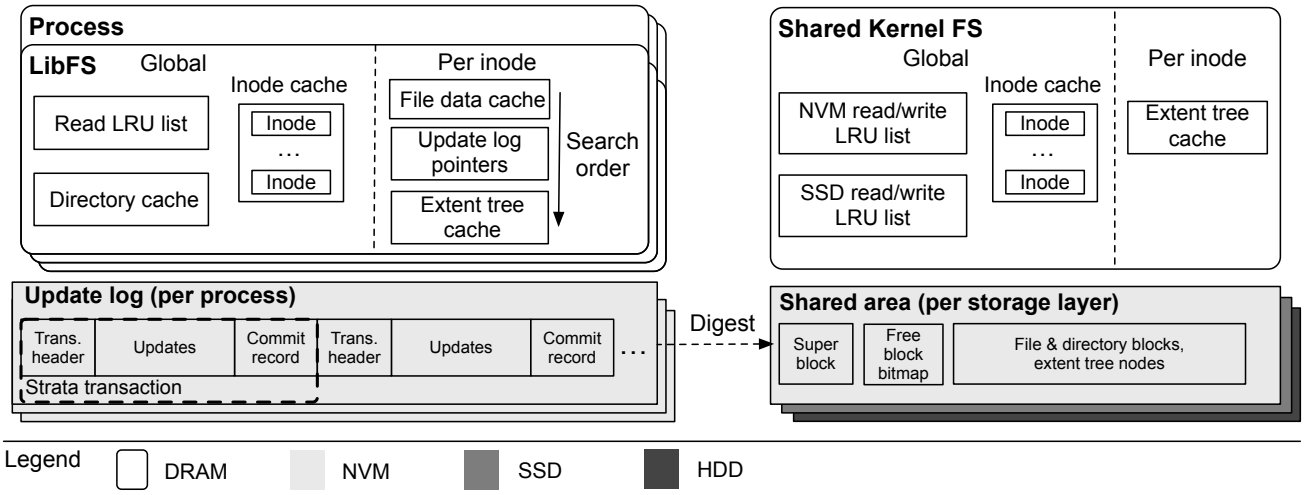


Figure 1: Strata design. Writes go to the update log. Reads are served from the shared area. File data cache is a read-only cache, containing data from SSD or HDD.

with their proper contents even if they were partially written before the crash (log replay is idempotent). The log remains authoritative until garbage collected after a completed digest. Since data is updated in a log-structured way, synchronization of log update and digest are simple. Writers make sure not to overwrite already allocated log blocks, while only allocated blocks are digested (and garbage collected). Write and digest positions are kept in NVM.

Sequential, aligned writes. One benefit of digesting writes in bulk is that, however they are initially written, file data can be coalesced and written sequentially to the shared area, minimizing fragmentation and meta-data overhead. Digestion minimizes device-level write amplification by enabling sequential, aligned writes. Below the NVM layer, all device writes are sequential and aligned to large block boundaries chosen to be efficient for the device, such as erasure blocks for SSDs and write zones for shingled disks. These parameters are determined by Strata for each device [46]. When data is updated, old versions are not immediately overwritten. Instead, Strata periodically garbage collects cold blocks to reclaim free space. Garbage collection consumes entire erasure/shingle block size units so that the device sees only full block deletes, eliminating collection overhead from the device layer. This process is similar to what would occur within device firmware but takes into account application data access patterns and multiple layers, segregating frequently from infrequently accessed data and moving them to appropriate layers for better device utilization and performance isolation.

Use hardware-assisted protection. To bypass the kernel safely and efficiently, Strata makes use of the hardware virtualization capabilities available in modern server systems. Strata specifies access rights for each application to contiguous subsets of each device's storage space, enforced by hardware. The MMU trivially supports this feature at memory page granularity for NVM, while NVMe provides it via *namespaces* that can be attached to hardware-virtualized SSDs [9]. Strata moves all latency-sensitive mechanisms of the file system into a user-level library. HDDs do not require kernel bypass.

We next describe each component of Strata and their interaction. Since Strata breaks the responsibilities of a traditional file system into LibFS and KernelFS, we organize our description along these lines. We start by describing Strata's principal meta-data structures.

3.1 Meta-data Structures

Strata keeps meta-data in superblocks, inodes, and per-layer bitmaps of free blocks. These data structures are similar to structures in other file systems and we only briefly describe them here. Strata caches all recently accessed meta-data structures in DRAM.

Superblock. Strata's superblock is stored in NVM and describes the layout of each storage layer and the locations of all per-application logs. It is updated by KernelFS whenever per-application logs are created or deleted.

Inodes and directories. Inodes store file meta-data, such as access rights, owner, and creation times. As in EXT4, they also store a root of each file's extent tree, though for Strata, an inode has multiple roots, one for each storage device. When unfragmented, extent tree nodes point directly to a file's data blocks. As the extent tree fragments, nodes point to other internal tree nodes before pointing to data blocks. Strata stores inodes ordered by number in a hidden, sparse *inode file* and manages it like a normal file: Strata accesses the inode file via

an extent tree and migrates blocks of the inode file to other layers. An inode for the inode file is located in the superblock.

Strata directories are similar to EXT4, holding a chained array of file names and associated inode numbers in their data blocks. On file reads, LibFS first consults per-inode *update log pointers* to find any updates in the log. The log pointers are invalidated when the local log is digested. We hash entire directory names [49] to improve our directory cache hit rate, reducing full directory traversals by up to 60%. Strata inodes fill 256 bytes. To efficiently protect inodes under kernel-bypass, inodes with different access permissions have to be stored on different pages or within different NVMe namespaces. POSIX specifies that all inodes stored within a directory have access permissions according to the directory inode. Thus, Strata organizes inodes of the same directory together by reserving consecutive inodes in multiples of 16 for each directory. Unused inodes remain reserved until allocated.

Free block bitmap. Strata has a per-layer persistent bitmap to indicate which of its blocks are allocated and free. For high throughput, KernelFS digest threads allocate and free blocks in large batches. These threads reserve blocks (e.g., the size of an erasure block) by adjusting a unit allocation count in DRAM using compare-and-swap, and then marking specific blocks as in use in the bitmap. Once the allocation count reaches the maximum, digesting moves on to a new erasure/shingle unit. Freed blocks are reset in the bitmap.

Multiple device instances. The Strata prototype supports only a single storage device at each level, but the design would generalize to multiple devices at each level, where devices are logically concatenated. For example, Strata can treat two 8TB SSDs as a single 16TB SSD. This approach allows Strata to add capacity, while redundancy is left as future work.

3.2 Library File System (LibFS)

Strata's library file system (LibFS) provides the application-level mechanism to conduct file IO. Its goal is to provide fast, crash-consistent, and synchronous read and write IO to the entire underlying storage hierarchy and a unified API that is fully compatible with existing POSIX applications and can be put underneath an application by re-linking with LibFS.

Fast and synchronous persistence. Synchronous persistence provides clear semantics (e.g., ordering guarantees and crash recovery) to applications [38], but it is not widely used under the assumption that storage devices are slow. Modern NVM storage technology allows Strata to provide synchronous IO semantics without sacrificing performance. In fact, synchronous semantics can accelerate overall IO performance for NVM. Strata writes data once to NVM and does not copy it to a DRAM buffer cache. Memory copy latencies are comparable to NVM write costs [22], so eliminating the memory copy approximately halves write latency.

Upon an application write request, LibFS writes directly to a per-application update log in NVM, bypassing the OS kernel. Favoring the byte-addressable feature in NVM, LibFS does *blind write* for small-sized writes (e.g., less than 4 KB). A small write is written sequentially to the log and turned into a block write when KernelFS digests it, maximizing IO throughput and eliminating write amplification. Synchronous semantics allow Strata to provide zero-copy IO—LibFS performs IO directly between a user's DRAM buffer and NVM. Strata does not use a page cache which eliminates cache lookup and data copy in the write path. However, LibFS does maintain caches of the locations of logged file updates, as well as meta-data, such as inodes, file sizes, and modification times (inode and directory tables in Figure 1).

LibFS organizes the update log as an operation log. The operation log reduces IO compared to a data log because the data log usually contain blocks, which are the minimum-sized addressable units for the file system. For example, when updating a directory, the data log requires three (block) writes: directory inode, directory block, and log header. The operation log requires only a record indicating the directory change such as `ADD filename, inode number`. This information is small enough to fit into the log header, resulting in a single write for directory changes.

We arrange the log format so that its effects are idempotent; applying the log multiple times results in the same file system state. For example, log entries use both the inode and offset to refer to locations modified in a file or directory. LibFS allocates inode numbers eagerly to simplify logging. It requests batches of inodes from the kernel, such that inode allocation does not require a system call in the common case.

Crash consistent logging. LibFS logs changes to all file system state, including file and directory meta-data. All data is appended sequentially to the log, naturally capturing the ordering of file system changes. Logging also provides crash consistent updates efficiently. As shown in Figure 1, when an application creates a file and then writes data to the file, LibFS first logs a file creation record (with file length of 0) followed by the data write record in causal order.

LibFS has a unit of durability, called *Strata transaction*. Strata transactions provide ACID semantics up to an application's update log size, allowing Strata to atomically persist multiblock write operations up to the size of the log. To do so, LibFS wraps each POSIX system call in a Strata transaction. However, single system calls with more data than the per-application log size (on the order of GBs) cannot be persisted atomically and are instead broken into multiple, smaller Strata transactions. Many applications desire ordered, atomic multiblock writes and can benefit from these semantics [42].

Each Strata transaction consists of a number of *log entries*: a header, the relevant updates to file (meta-)data, followed

by a commit record. The commit record contains a unique and monotonically increasing Strata transaction number and a checksum of the header contents. When a Strata transaction commits, LibFS ensures atomicity and isolation by atomically allocating log space using a compare-and-swap operation and by first writing the header and data, waiting for persistence, and then persisting the commit record. Log headers contain a pointer to the next log header so the log can be easily replayed.

Digest and garbage collection. The log is a limited resource and needs to be periodically digested into the shared area and garbage collected. Once the log fills beyond a threshold (30% in our prototype), LibFS makes a digest request to KernelFS. KernelFS digests the log asynchronously in the background and replies to LibFS once the digest request is complete. After completion, LibFS can safely reclaim log entries (also in the background) by resetting each log header's valid bit. Strata data structures allow the user to add records to a log that the kernel is concurrently digesting.

If an application completely fills its log, LibFS must wait for an in-progress digest to complete before it can reclaim log space and restart file system activity. Log garbage collection involves trimming log headers (a device-level trim operation zeroes the trimmed data blocks) and invalidating the corresponding entries in the data cache. LibFS garbage collects using a background thread. The application can continue to append log blocks during garbage collection. The log's idempotency ensures crash consistency. If the system crashes during a digest, the log is re-digested on recovery, resulting in the same file system state as a successful digestion without a crash.

Fast reads. LibFS caches data and meta-data in DRAM. However, data is only cached when read from SSD or HDD. NVM does not require caching. The file data cache is managed in 4 KB block units and evicted to the update log in an LRU manner. Meta-data such as file access time and file data locations (in the log and in the shared area) are cached in the inode cache indexed by inode number. LibFS also records update addresses in the log using the update log pointers and it caches extent tree nodes. To optimize performance of sequential reads from SSD or HDD, LibFS uses a read-ahead buffer in DRAM of 256 KB.

To resolve a file location with the most up-to-date data, LibFS searches the file data cache, the update log, and then the (cached) extent trees from highest (NVM) storage layer to lowest (HDD), as shown in Figure 1. If the file data is not found in the data cache, but in the update log pointers, then the latest data is read from the log and (depending on the read size) possibly merged with blocks from the shared area. In that case, both log data and shared blocks are fetched and merged before returning data to the read request. If a lookup misses in the extent tree cache for a layer, then Strata traverses the extent tree stored in that layer's shared area and updates the cache, before advancing to the next layer. Extent trees in multiple layers can be present for a file if subsets of its data blocks have been migrated. Extent trees indicate which of a file's data blocks are present at the tree's layer. Strata's layered data storage is not inclusive and a data block can be simultaneously present in any subset of layers. Strata's migration algorithm ensures that higher layers have the most up-to-date block and thus, higher layers take precedence over lower layers.

3.3 Kernel File System (KernelFS)

Strata's kernel file system (KernelFS) is responsible for managing shared data that can be globally visible in the system and may reside in any layer of the storage hierarchy. To do so, it digests application logs and converts them into *per-file extent trees*. Digestion happens asynchronously in the background, allowing KernelFS to batch Strata transactions and to periodically garbage collect and optimize physical layout. LibFS provides least-recently-used (LRU) information to KernelFS to inform its migration policy among layers of the storage hierarchy. KernelFS also arbitrates user-level concurrent file access via *leases* (§3.4).

Digest. When the log size grows beyond a threshold, LibFS makes a digest request to KernelFS. Digest latencies have an impact on applications' IO latencies as the log becomes full. To reduce the digest latencies, KernelFS employs a number of optimizations. KernelFS digests large batches of operations from the log, coalescing adjacent writes, as well as identifying and eliminating redundant operations. KernelFS begins digestion by first scanning the submitted log and then computing which operations can be eliminated and which can be coalesced. For example, if KernelFS detects an inode creation followed by deletion of the inode, it skips log entries related to the inode.

These optimizations reduce digest overhead by eliminating work, batching updates to extent trees, and reducing the number of tree lookups. Coalescing writes increases the average size of write operations, minimizing fragmentation and thus extent tree depth. Optimizing the digest reduces bandwidth contention for the storage device between KernelFS and LibFS, as well as write amplification. Experiments with the Filebench [47] benchmark show that optimizations reduce digest latency up to 80%, improving application throughput by up to 15%. Scanning the log before digesting allows KernelFS to determine which new data and metadata blocks are required and to allocate them in large, sequential batches. Log scanning also allows KernelFS to determine if two logs contain disjoint updates and thus can be digested in parallel.

For all data updates, Strata writes new data blocks before deleting old blocks. Even metadata structures like extent trees

are completely written before the updates are committed when the inode's root pointer is updated.

Data access pattern interface. To take advantage of the entire storage hierarchy, KernelFS transparently migrates data among different storage layers, keeping least recently used blocks in better performing layers. In order to migrate data efficiently, KernelFS requires LRU information for each block. Because reads bypass the kernel, LibFS must collect access information on reads and communicate the information to KernelFS via a kernel interface. LRU information is not persisted and only maintained in DRAM, which conserves NVM log space. Writes are observed by the kernel when digesting update logs, so there is no need for LibFS to provide additional metadata about writes.

The KernelFS maintains LRU lists for each storage layer except for the last one. An LRU list is a sequence of arbitrary length, of logical 4KB block numbers. LibFS can submit access information as frequently as it wishes via a system call. KernelFS transforms the LibFS-provided LRU lists into coarser-grained lists for storage layers that have larger block sizes (e.g., 1MB blocks for NVM and 4MB for SSD).

KernelFS does not trust the LRU information provided by a LibFS and enforces that blocks specified as recently used are actually accessible by the process. Applications can misuse the interface to get the kernel to place more blocks in NVM, but this is equivalent to current systems where an application can read data to get the kernel to place it in the DRAM page cache. Resource allocation interfaces like Linux's memory cgroups [2] would further limit the impact of API misuse, though integrating with cgroups is left as future work.

Data migration. To take advantage of the storage hierarchy's capacity, the kernel transparently migrates data among different storage layers in the background. To benefit from concurrency and to avoid latency spikes due to blocking on migration, Strata migrates data before a layer becomes full (at 95% utilization in our prototype). Migration is conducted in a block-aligned, log-structured way, similar to digestion. To make migrations efficient and at the same time reduce fragmentation, Strata moves SSD data in units of flash erasure blocks (order of hundreds of megabytes) and HDD data in shingles (order of gigabytes). After migrating a unit, the whole unit is trimmed (via the device TRIM command) to make a large, unfragmented storage area available. When migrating data, KernelFS tries to place hot data in higher layers of the storage hierarchy, while migrating cold data down to slower layers. To maintain a log-structured write pattern, KernelFS always reserves at least one migration unit on each layer and writes blocks retained in that layer to the reserved migration unit sequentially.

3.4 Sharing (leases)

Strata supports POSIX file sharing semantics, while optimizing application access to files and directories that are not concurrently shared. KernelFS supports *leases* on files and sections of the file system namespace. Leases have low execution time overhead for coarse-granularity sequential sharing of file data. We expect that processes that require fine-grained data sharing will use shared memory or pipes—avoiding the file system altogether due to its generally higher overhead.

Similar to their function in distributed file systems [27], leases allow a LibFS exclusive write or shared read access to a specific file or to a region of the file system namespace rooted at a certain directory. For example, a LibFS can lease one or more directories and then create nested files and directories. Multiple LibFS may hold read leases, while only one write lease may exist.

Write leases are strict, they function like an exclusive lock. As long as a write lease is held, a thread in a process may write to the leased namespace (or file) without kernel mediation, while operations from other processes are serialized before or after the lease period. Threads within the same process see each others' updates as soon as operations complete, using fine-grained inode locks to synchronize file system updates. Leases are independent from file system access control checks, which occur when a file or directory is opened.

A process that holds a write lease is notified via an upcall (via a UNIX socket in our prototype) if another process also wants the write lease. Upon revocation of a write lease, applications can insist on the write-back of new data (via a log digest) to the kernel's shared file system area (e.g., to NVM). Waiting for a digest operation will increase the latency of revoking the lease. Leases are also revoked when an application is unresponsive and the lease times out. Because user-level operations are transactional, Strata can abort any in-progress file system operation upon revocation of a lease if necessary. LibFS caches are invalidated upon loss of a lease.

Programs may acquire leases using explicit system calls, which allows user-level control, but is not POSIX compatible. Our prototype lazily acquires an exclusive (shared) lease on the first write (read) to any file or directory (unless the process already has a lease). This policy works for our benchmarks, but other policies are possible. Bad policy choices lead to poor performance, but do not compromise correct sharing semantics because Strata can always fall back to kernel mediation for all file system operations. If a file is opened read/write by multiple processes, the kernel eliminates logging.

To show the worst-case performance overhead of sharing through the file system, we measured update throughput of two processes using a lock file to coordinate small (4KB) updates to a shared data file. In one iteration, a process tries to create the lock file. Once creation succeeds, the winning

process writes a 4KB block to the data file, then it unlinks the lock file. Note that neither file is ever synced. To guarantee strict ordering and synchronous persistence, LibFS must first acquire a lease in order to create the lock file and relinquish the lease and perform a digest after the lock file is unlinked. Strata achieves a throughput of 10,400 updates/s, 4.3× slower than EXT4-DAX and 1.7× slower than NOVA. EXT4-DAX can perform metadata updates in the buffer cache, but unlike Strata and NOVA, it lacks synchronous, ordered file semantics. Both NOVA and EXT4-DAX only write shared data once, while Strata must write it again during digestion to make the data globally visible. We thus view logging in Strata as an optimization to accelerate infrequently shared data. However, in situations with less strict ordering and atomicity guarantees, logging could be used even when sharing frequently.

3.5 Protection and performance isolation

Protection with kernel bypass. Strata supports POSIX file access control, enforced by MMU and NVMe namespaces. The MMU provides protection for kernel-bypass LibFS operations and Strata aligns each per-file extent tree on a page boundary (and pads the page) to facilitate MMU protection. The kernel maps all data and meta-data pages of the accessed file read-only into the caller's virtual address space. Extent tree nodes refer to blocks using logical block numbers. An entire device can be mapped contiguously, making the mapping from logical block number to address a simple addition of the base address. However, more parsimonious mappings are possible along with a table to track the mapping between address and logical block number.

For SSD-resident data, Strata uses NVMe namespaces for protected access to file data. File extent trees must be aligned on a NVMe sector (512 bytes or 4KB, depending on how the device is formatted). Upon opening a file on the SSD, the kernel creates a read-only NVMe namespace for the file if the namespace doesn't already exist and attaches it to the application's NVMe virtual function. The NVMe standard supports up to 2^{32} namespaces, which limits the total number of open files on the SSD to this number. If an SSD does not support virtual functions, namespace management, or a sufficient number of namespaces, this functionality can be efficiently emulated in software, with an overhead of up to 3 μs per system call [41]. HDD access is kernel mediated.

Performance isolation. Write amplification has an effect on IO performance isolation by inflating device bandwidth utilization. When device firmware amplifies writes it can throw off the operating system's management algorithms. Firmware-managed devices often have unpredictable and severe write amplification from wear leveling and garbage collection [46]. Since Strata minimizes firmware write amplification via aligned sequential writes, almost all amplification occurs in software. This has the benefit that it can be accurately observed and controlled by Strata. For example, KernelFS can decide to stop digesting from an application if the incurred write amplification would violate the QoS (specified as per-application I/O bandwidth allocations) of another application.

3.6 Example

To summarize the design, we walk through an example of overwriting the first 1 KB of data in an existing, non-shared file and then reading the first 4 KB.

Open. The application uses the `open` system call to open the file. Upon this call, LibFS first checks to see whether the file exists and whether it can be accessed, by walking all path components from the root. For each component, it acquires read leases and checks the directory and inode caches for cached entries. If a component is not found in a cache, LibFS finds the inode by number from the inode file located in the shared area. Assuming the data is in NVM, LibFS will map the corresponding inode page read-only. The kernel allows the mapping if the inode is accessible by the user running the application. LibFS first copies the inode's content to the inode cache in DRAM. It then reads the inode (from cache) to determine the location of the directory by walking the attached extent tree, storing extent tree entries in the extent tree cache. Finally, LibFS finds the correct entry within the directory. The directory entry contains the inode number of the file, which LibFS resolves in the same manner. The file is now open, and LibFS allocates a file descriptor.

Write. The application issues the `write` system call to write 1 KB to the beginning of the file. LibFS wraps the system call in a Strata transaction and requests a write lease for the corresponding inode. No other processes are accessing the file, so the kernel grants the lease. The Strata transaction can commit and LibFS appends the write request, including payload to the update log, checks the file data cache for invalidation, and updates the corresponding block in the update log pointers with addresses of the update log. The write is complete.

Read. The application issues a `pread` system call to read the first 4 KB from the file. Like the write case, LibFS first tries to obtain a read lease, waiting until KernelFS grants the read lease. LibFS first searches the file data cache with offset 0 and finds that the block is not in the cache (invalidated by the write above). Then, it searches the update log pointers with offset 0, finding a block in the update log. However, the update log does not contain the entire 4 KB (it has a 1 KB

partial update). In that case, LibFS first finds the 4 KB block of the file by walking the extent tree at each layer from the inode. It finds the block in the SSD. To read it, it requests a new NVMe namespace for the block, which the kernel creates on the fly. This allows LibFS to read the block bypassing the kernel. LibFS allocates a file data cache entry (at the head of LRU list), reads the block into the cache entry, patches it with the update from the update log. LibFS can now return the complete block from the file data cache to the user.

Close. The application closes the file. At this point, LibFS relinquishes the lease to the KernelFS (if it still has it).

Digest. At a later point, the kernel digests the update log contents. It reads the same 4KB block from the SSD, patches the block with the 1 KB update from the log, and writes the complete block to a new location in NVM (the block was recently used). Next, it updates the extent tree nodes to point to the new location by first reading them from the appropriate layers and then writing them to NVM. Finally, it updates the inode containing the extent tree root pointer in NVM. The digest is done and LibFS garbage collects the update log entry.

4 IMPLEMENTATION

We have implemented Strata from scratch, using Intel's Storage Performance Development Kit (SPDK) [30] for fast access to NVMe SSDs bypassing the Linux kernel and Intel's libpmem [10] to persist data to emulated NVM using non-temporal writes to avoid polluting the processor cache and the appropriate sequence of store fence and cache flush to guarantee persistence [54] (our testbed does not support the optimized `CLWB` instruction, so libpmem uses `CLFLUSH` to flush the cache to NVM). We also use the extent tree implementation of the EXT4 [35] file system and modified it for log-structured update.

Our prototype of Strata is implemented in 21,255 lines of C code. Shared data structures, such as lists, trees, and hash tables, account for 4,201 lines. LibFS has 10,131 and KernelFS has 6,923 lines of code. The main functionality in LibFS is writing to the update log. In KernelFS it is the extent tree update code and code for data migration. On top of Strata's low-level API, we implement a POSIX system call interposition layer. To do so, we modify glibc to intercept each storage-related system call at user-level and invoke the corresponding LibFS version of the call. The interposition layer is implemented in 1,821 lines of C code.

Our prototype is able to execute a wide range of applications. Strata successfully completes all 201 unit tests of the LevelDB key-value store test suite, as well as all tests in Filebench.

4.1 Limitations

Our current prototype has a few limitations, which we describe here. None of them impact our evaluation.

Kernel. Instead of loading our kernel module into the kernel's address space, we have placed it in a separate process and use the sockets interface to communicate "system calls" between LibFS and KernelFS. This results in higher overhead for system calls in Strata due to the required context switches. However, we believe the impact to be small, as a design goal of Strata is to minimize kernel-level system calls.

Leases. Leases are not fully implemented. We have evaluated their overhead, especially worst case performance (§3.4), but the prototype does not implement directory consistency, for example. Our benchmarks do not stress fine-grained concurrent sharing that would make lease performance relevant.

Memory mapped files. We did not implement memory mapped files because they are not used by our target applications. Memory mapped files increase write amplification for applications with small random writes. The hardware memory translation system is responsible for tracking updates to memory mapped files via dirty bits that are available only at a page granularity. A page is thus the smallest write unit. This is a general problem for memory mapped files, in particular as page sizes grow.

The common case of read-only mappings or writable private mappings are easy to accommodate in Strata. NVM pages can be mapped into a process' address space just as current OSes map page cache pages. The difficulty with shared writable mappings is their requirement that writes into memory are visible to other processes mapping the file. If writes must be immediately visible, Strata cannot do any user-level buffering and logging, but if writes can be delayed, Strata can buffer (and log) updates. On `msync`, LibFS writes updates (pages on which the dirty bit is set) to the log, and they are visible to other processes after digesting.

Fault tolerance. Strata currently does not contain any redundancy to compensate for storage device failures. Because it stores data across several devices, its mean time to data loss (MTTDL) will be the minimum of all devices. It remains future work to apply distributed reliability techniques to improve MTTDL in Strata [20, 29]. With Strata it is also not safe to remove individual storage devices from a powered down machine, without advance warning.

5 EVALUATION

We evaluate the performance and isolation properties of Strata. To put the performance of Strata into context, we compare it to a variety of purpose-built file systems for each storage layer. For NVM, we compare with the Linux EXT4-DAX [1] file system in its default ordered data mode, as well as to PMFS [25] and NOVA [52]. On the SSD, we compare to

F2FS [32]. On the HDD, we compare to EXT4 [35], also in ordered data mode. Ordered mode is the Linux default for EXT4 because it provides the best tradeoff between performance and crash consistency.

To evaluate the data management and migration capabilities of Strata, we compare it to a user-space framework that migrates files among layers without being integrated into the file system, as well as to a block-level two-layer cache provided by Linux's logical volume manager (LVM) [8]. The user-space management framework uses the NOVA, F2FS, and EXT4 file systems for the NVM, SSD, and HDD layers, respectively.The LVM cache uses the NVM and SSD layers, with a single F2FS file system formatted on top.

We seek to answer the following questions using our experimental evaluation.

- How efficient is Strata when logging to NVM and digesting to a storage layer? How does it compare to file systems designed for and operating on a single layer?
- How do common applications perform using Strata? How does performance compare on other file systems?
- How well does Strata perform when managing data across layers, compared to solutions above (at user-level) and below the file system (at the block layer)?
- What is the multicore scalability of Strata? How does it compare to other file systems?
- How isolated are multiple tenants when sharing Strata, compared to other file systems?

Testbed. Our experimental testbed consists of 2 × Intel Xeon E5-2640 CPU, 64 GB DDR3 RAM, 400 GB Intel 750 PCIe-SSD, 1 TB Seagate hard-disk, and a 40 GbE Mellanox MT27500 Infiniband network card. All experiments are performed on Ubuntu 16.04 LTS and Linux kernel 4.8.12. We reserve 36 GB of DRAM to emulate NVM and leave the remaining 28 GB as DRAM. The other devices are used to capacity. Strata reserves 1 GB of write-only log area for each running application within NVM, the rest is dedicated to the shared area. To benefit from overlapping operations, LibFS starts a digest when its update log is 30% full. This value provided a good balance between digest overlap and log coalescing opportunities in a sensitivity study we conducted. All experiments involving network communication bypass the kernel using the rsockets [13] library.

NVM emulation. To emulate the performance characteristics of non-volatile memory, we have implemented a software layer that uses DRAM but delays memory accesses and limits memory bandwidth to that of NVM. The emulation implements an NVM performance model according to a recent study [54] (we could not obtain the PMEP hardware emulator used in the study).

The study predicts that NVM read latencies will be higher than DRAM. As done in NOVA and other studies, our model emulates this latency on all NVM reads by adding the latency differential to a DRAM read. In reality, read latency would be incurred only on a cache miss, but (like other studies) we do not emulate this behavior (making our model conservative). Due to write-back caching, writes do not have a direct latency cost as they reside in the cache. The study investigates the cost of a *write barrier* (e.g., Intel's PCOMMIT instruction) which ensures that flushed data does not remain in volatile buffers within the memory controller. Intel deprecated this write barrier from the x86 architecture [44], instead requiring NVM controllers to be part of the system's power-fail safe persistence domain. Data flushed from the cache are guaranteed to be made persistent on a power fail due to on-chip capacitors. Thus, our model does not require write barriers and their attendant (non-trivial) latency. Strata uses the mandatory fences and cache flush to enforce ordering, incurring that cost. Finally, NVM is bandwidth-limited compared to DRAM by an estimated ratio of $\frac{1}{8}$. Our performance model tracks NVM bandwidth use and if a workload hits the device's bandwidth limit, the model applies a bandwidth-modeling delay $B = \frac{\sigma \times (1 - \mathrm{NVM}_b/\mathrm{DRAM}_b)}{\mathrm{NVM}_b}$, with σ the size of the write IO in bytes, NVM_b the NVM bandwidth, and DRAM_b the DRAM bandwidth. The emulator resets the bandwidth limit every 10ms, which provided stable performance in a sensitivity study. With $\frac{\mathrm{NVM}_b}{\mathrm{DRAM}_b} = \frac{1}{8}$, we measure stable peak NVM bandwidth of 7.8GB/s as shown in Table 1.

5.1 Microbenchmarks

Hardware IO performance. To ensure that no other resource in our testbed system is a bottleneck, we first measure the achievable IO latency and throughput for each memory technology contained in our testbed server using sequential IO. The measured hardware IO performance matches the hardware specifications of the corresponding device (see Table 1). We measure DRAM using a popular memory bandwidth measuring tool [5]. The reported NVM performance is in line with our NVM performance model.

File system write efficiency. Write amplification is a major factor in a file system's common case performance. Most file systems amplify writes by writing meta-data in addition to user data, lowering their *write efficiency* (defined as the inverse of write amplification). For example, if a program writes and syncs 2 KB of data and the file system updates and writes a 4 KB data block and a 4 KB metadata block, then the write amplification is 4 and the write efficiency is 25%.

Figure 2 shows write efficiency for Zipfian ($s = 1$) random writes until a total of 1 MB has been written. We can see that for small writes ($\leq$ 1 KB), write efficiency suffers substantially for most file systems. Strata achieves the highest write efficiency among all file systems regardless of write

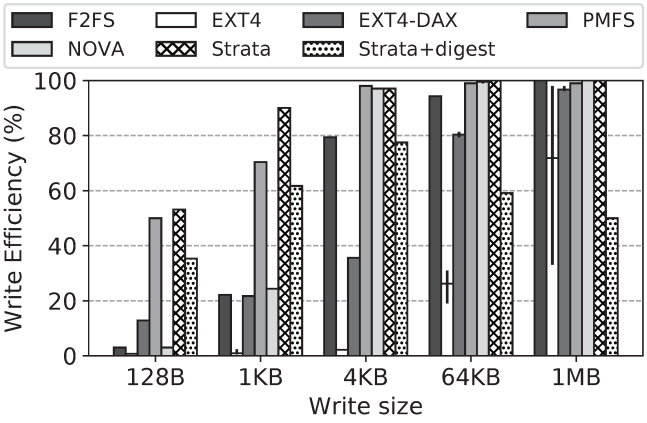


Figure 2: File system Zipf IO write efficiency. Error bars show minimum and maximum measurement.

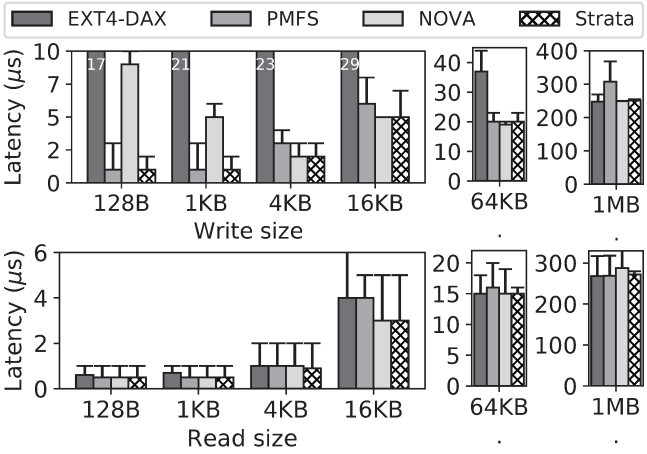


Figure 3: Average IO latency to NVM. Error bars show 99th percentile.

sizes because Strata performs the minimal amount of IO for persisting data and meta-data changes with the operation log. The Strata + digest result includes additional writes to digest data into the shared area in the background. Depending on IO properties, Strata can greatly improve write efficiency by coalescing the digest (§5.2). In this case, write efficiency declines as write size approaches total size and coalescing opportunities diminish.

Latency. We compare the read and synchronous (fsync) write latency of Strata and other NVM-optimized file systems using various IO sizes on an otherwise idle system. This experiment emulates the small, bursty writes often exhibited by cloud application back ends, such as key-value stores. Latency and tail-latency are of primary importance for these applications as it determines the processing latency of a user's web request.

In this experiment, the burst size is 1 GB. For Strata, this case is ideal as it fits into the update log. Hence, no digest occurs during the experiment. Before the read phase, Strata digests the file, hence all reads are served directly from the shared extent tree area. We assume this case to be common, as most key-value stores cache data in DRAM and so it is likely that recently written data can be served from DRAM.

	1 KB	4 KB	64 KB	1 MB	4 MB
EXT4-DAX	35	44	98	812	2947
PMFS	7	10	53	656	2408
NOVA	13	17	54	563	2061
Strata	5	8	49	569	2074
No persist	4	6	30	302	1157

Table 3: Latency (μs) of (non-)persistent RPC.

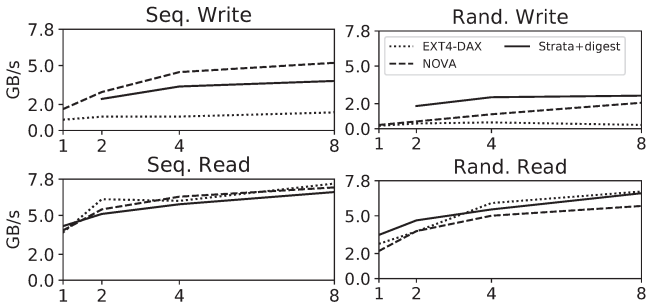


Figure 4: NVM throughput scalability (4 KB IO). X-axis = number of threads; Top Y-axis = NVM bandwidth.

Figure 3 shows read and write latencies. We can see that Strata achieves equivalent latency to the best-performing alternative file system, regardless of IO size. In the 99th-percentile tail, Strata achieves up to 33% lower latency for small writes and up to 12% for reads compared to the best-performing alternative file system.

Strata's performance comes from writing an operation log using kernel-bypass. In the other file systems, IO either involves copying data from user to kernel buffers (EXT4-DAX), or various persistent file system structures are modified in-place (PMFS), or they copy-on-write (NOVA), while Strata simply logs write operations using a single log write operation with no DRAM copying.

Persistent RPC. Many cloud services use remote procedure calls (RPCs) that must persist data before returning to the caller. Table 3 shows a microbenchmark, where one client sends an RPC to a server that logs it to stable storage. Strata can synchronously persist 1KB of data for each RPC only 1 μs (25%) later than an RPC that persists no data. Strata makes persistence cheap in terms of latency for the data sizes that are common in RPCs (≤4KB).

Log size sensitivity. Log size is configurable on a per-log basis. LibFS should configure log size according to expected burst behavior of the application. We conduct a sensitivity study to find how performance is affected by log size, analyzing the sequential write microbenchmark and the *varmail* workload from the Filebench suite [47]. We vary the log size

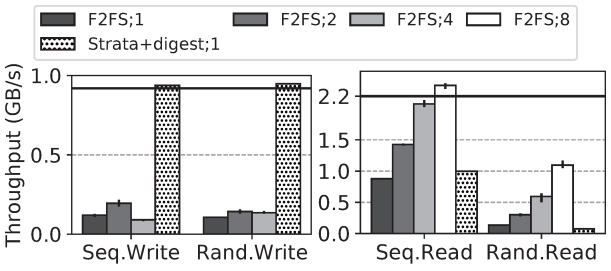


Figure 5: Average SSD throughput (4 KB IO) over number of threads. Horizontal lines = SSD bandwidth.

between 100 MB and 4 GB. We find that with a log size of 500 MB and larger, both workloads reach maximum performance. A 100 MB log degrades performance by up to 6.4%. This result confirms that Strata's background digest and garbage collection work efficiently even for small log sizes.

Throughput scalability. We compare sustained IO throughput using 4 KB IO size. When reading, update logs are clean. This benchmark emulates common data streaming workloads with sustained busy periods. Strata always logs data to NVM and digests it to the evaluated layer and our results include both operations. The other file systems operate directly on their respective layer, but F2FS and EXT4 use a buffer cache in DRAM. For Strata, we count LibFS (logging and reading) and KernelFS (digesting) threads.

Figure 4 shows the result for NVM, varying the number of threads from 1 to 8. Each benchmark run conducts 30 GB of IO, partitioned over the number threads and using a private file for each thread to avoid application-level locking. PMFS crashes when using multiple threads. With 4 threads, Strata approaches NVM bandwidth. Since both workloads have no locality, write efficiency is 50% for Strata, resulting in an application-level throughput that is half the NVM bandwidth. Strata is up to 26% slower than NOVA for sequential writes. This is the worst case for Strata since KernelFS cannot improve write efficiency via digest optimizations. However, for random writes, Strata achieves 28% higher throughput than NOVA with 8 threads and 3× higher throughput with 2 threads. This is because LibFS blindly writes small, unaligned data to the log, and KernelFS can merge them into block writes when digesting the log, while NOVA has to read and modify blocks. EXT4-DAX is up to 10% faster than Strata on reads, but for a single-threaded workload Strata is 36% faster than EXT4-DAX for random reads.

Figure 5 shows SSD throughput of Strata and F2FS. We mount F2FS with the synchronous option to provide the same guarantee as Strata. For writes, Strata achieves 7.8× better throughput than F2FS by aggregating writes in the update log and batching them to SSD on digest. For sequential reads, both Strata and F2FS read ahead to achieve similar performance (our Strata prototype implementation currently supports only single-threaded SSD access). When accessing the

[μs]	EXT4-DAX		PMFS		NOVA		Strata	
Mean	44.97	45.14	4.43	3.72	2.34	1.82	1.58	1.44
99th	74	74	7	6	4	3	2	2
99.9th	84	84	54	45	5	4	3	3
99.99th	95	95	72	50	10	6	6	5

Table 4: (Tail-)latencies with two clients (4 KB IO).

HDD, Strata achieves ~10% better throughput than EXT4 for all operations (full device bandwidth for sequential IO, 10 MB/s for random write and 3 MB/s for random read), but with synchronous write semantics and without any of EXT4's complexity. For example, Strata does not require a journal, delayed block allocation, or locality groups.

Using a portion of NVM as a persistent update log allows Strata to perform similarly or better than file systems purpose-built for each storage layer, while providing synchronous and in-order write semantics.

Data migration. To show Strata's read/write performance on multiple layers, we run a Zipfian ($s = 1$) benchmark with an 80:20 read/write ratio on a 120 GB file that fits in NVM and SSD and compare to F2FS (4 KB IO size). Strata achieves a throughput of 4.4 GB/s, while F2FS attains 1.3 GB/s. The locality of the workload causes most IO to be served from NVM, a benefit for Strata because of NVM's higher capacity compared with the DRAM buffer cache used by F2FS.

Isolation. Clients want the write performance allotted to them regardless of the activities of other clients. We run two processes that write and fsync (and digest using a kernel thread for each process in Strata) a burst of 4KB operations concurrently and observe write (tail-)latency to evaluate how well Strata isolates multiple clients, compared to other file systems. Table 4 shows latency experienced by two competing clients. We can see that Strata provides equivalent latencies to the single client case, while other file systems do not isolate clients as well. EXT4-DAX provides equality, but slows down under concurrent access. NOVA and PMFS do not provide equal performance to both clients. Strata allocates fully isolated per-client write logs that can be allocated on client-local NUMA nodes, while other file systems use shared data structures for write operations that cause performance crosstalk because of locks and memory system effects.

5.2 Filebench: Mail and Fileserver

Mail servers access and create/delete many small files and are thus a good measure of Strata's meta-data management. File servers are similar, but operate on larger files and have a higher ratio of file IO compared with the number of directory operations. To evaluate these workloads, we use the *Varmail* and *Fileserver* profiles of the Filebench file system benchmark suite that is designed to mimic common mail and file servers. Both benchmarks operate on a working set of 10,000 files. Files are created with average sizes of 32 KB and

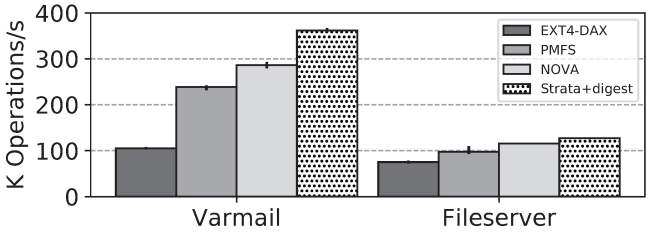


Figure 6: Varmail and Fileserver throughput.

128 KB for Varmail and Fileserver, respectively, though files grow via 16 KB appends in both benchmarks. Both workloads read and write data at 1 MB granularity. Varmail and Fileserver have write to read ratios of 1:1 and 2:1, respectively. In this configuration, all workloads fit into NVM and thus we compare performance to NVM file systems.

Varmail uses a write-ahead log for crash consistency (**sync** signifies Varmail waiting for persistence). Its application level crash consistency protocol involves creating and writing the write-ahead log in a separate file, **sync**, appending a mail to the mailbox file, **sync**, and then garbage collecting the now redundant write-ahead log by deleting the separate file. The Fileserver workload is similar to the SPECsfs [14] benchmark. It randomly performs file creates, deletes, appends, reads, writes, and attribute operations on a directory tree.

Figure 6 shows the result. Varmail achieves 26% higher write throughput on Strata versus NOVA: 57% of the time is spent in Varmail; 18% is spent in reading application data from NVM and 14% in writing application data to the log area. Another 10% is spent searching directories. Less than 1% of the time is spent in other Strata activities, such as searching extent trees. Fileserver throughput improvements are smaller (7% versus NOVA). This is expected; Fileserver has a larger average write size and no crash consistency protocol.

Strata's log compaction strategy is a good fit for Varmail, which creates and deletes many temporary files and performs many temporary durable writes. Strata digestion skips 86% of the log records because those updates have been superseded by subsequent workload updates: 50% are data writes, 24% are directory updates, and 12% are file creates. For example, if the workload creates a temporary file, it can write and read the file, but if it deletes the file before digestion Strata does not need to copy the file's data and metadata to the shared area. This optimization avoids 14 GB of data copying.

5.3 Data Migration

To show performance when Strata uses multiple storage devices, we configure the Fileserver workload to do 1 MB appends with 1000 files. In this case, the working set starts out operating in NVM, but then grows to incorporate the SSD and HDD. Fileserver's workload is uniform random. It has no locality. Thus, all evaluated systems only migrate data down towards slower layers.

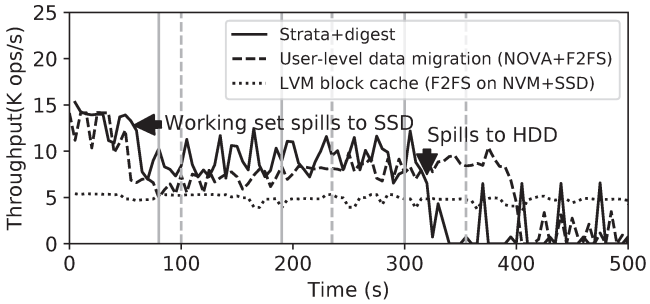


Figure 7: Fileserver throughput on multi-layer storage over time. Vertical lines every 1 million operations.

We compare Strata to a user-space data migration (UDM) framework using NOVA, F2FS, and EXT4 file systems on the appropriate layers with default mount options. UDM migrates files instead of blocks, which is a best case for UDM because the Fileserver workload performs mostly complete file I/O. We also compare to Linux's logical volume manager (LVM) cache target [8]. LVM caches blocks of a fixed *cache chunk* size underneath the file system. We cache SSD blocks in NVM, running F2FS on top (outperforming EXT4) in synchronous mode (-o sync) for a fair comparison. We cannot use EXT4-DAX or NOVA since LVM requires a block abstraction. To keep crash-consistent block-to-device mappings, it persists cache meta-data to a separate partition. We configure LVM to provide write-back caching with 64 KB cache chunk, reserving 300 MB of NVM space to store the cache meta-data.

Figure 7 shows the result. Both Strata and UDM start with full throughput on NVM, but UDM demonstrates more jitter. This is because of log garbage collection in NOVA. After 80 seconds, the working set sizes are large enough that data starts migrating to SSD, and it quickly becomes the bottleneck. In this period, digesting is slower than logging so the application stalls on a full log (between spikes in Strata). UDM's throughput drops below that of Strata causing UDM to fall behind (vertical lines). UDM lags because it migrates entire files, rather than individual blocks. Strata's workload grows to include HDD after 310 seconds (90 seconds later for UDM due to lower throughput) and throughput drops significantly.

Both Strata and UDM start out attaining 2.8× better throughput than F2FS on top of LVM. Once the working set spills to SSD, Strata is 2.0× faster than LVM. Note that LVM does not use the HDD, which is why it maintains its throughput throughout the duration of the experiment. LVM's working set never grows beyond the size of the SSD. Strata's approach to managing multi-layer storage offers higher throughput compared to multi-layer caching at the block layer. Strata can leverage the low latency and byte addressability of NVM directly by providing a user-level update log, while LVM requires a block-level interface in the kernel. LVM also adds additional cache meta-data IO to every file system IO.

	EXT4-DAX	PMFS	NOVA	Strata
Write sync.	**49.2**	**18.9**	**35.2**	**17.1**
Write seq.	8.7	5.4	15.0	4.9
Write rand.	19.5	15.2	25.0	11.1
Overwrite	28.0	20.9	37.7	17.3
Delete rand.	5.6	3.6	12.3	3.3
Read seq.	1.2	1.1	1.1	1.1
Read rand.	6.3	5.8	6.7	5.8
Read hot	1.6	1.5	1.5	1.5

Table 5: Latency [μs] for LevelDB benchmarks.

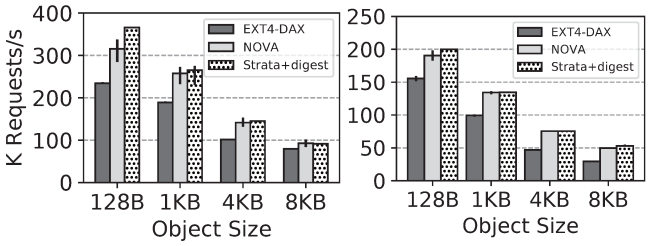
Strata's cross-layer approach also performs well compared to a solution operating above the file system, treating each layer as a black box. Strata benefits from combining application access pattern knowledge and cross-layer block allocation. Hence, Strata can maintain more meta-data in faster layers to speed up file system data structure traversal.

5.4 Key-value Store: LevelDB

Modern cloud applications use key-value stores like LevelDB [23]. LevelDB exports an API allowing applications to process and query structured data, but uses the file system for data storage. We run a number of LevelDB benchmarks, including sequential, random, and random IO with 1% of hot key-value pairs with a key size of 16 bytes and a value size of 1 KB on a working set of 300,000 objects. We measure the average operation latency. This workload fits into NVM and we compare against NVM file systems. To achieve higher performance, LevelDB does not wait for persistence after each write operation, but Strata guarantees persistence due to its synchronous semantics. We introduce a synchronous random write category (bold font in Table 5) to show a case where persistence is requested upon every operation by LevelDB.

Table 5 shows the result. We can see that LevelDB achieves lower latency on Strata than on any of the other NVM file systems, regardless of workload. In particular for random writes and overwrites, Strata performs 27% and 17% better than PMFS, respectively, while providing synchronous write semantics. NOVA does not perform well on this workload, as it uses a copy-on-write approach, which has high latency overhead, while EXT4-DAX incurs kernel overhead.

Our experiment demonstrates that a file system with a simple, synchronous IO interface can provide low latency IO if the underlying storage device is fast. Modern applications struggle to make logically consistent updates that are crash recoverable [42] and Strata helps such systems by providing simple recovery semantics. For example, SQLite must call `fsync` repeatedly to persist data to its log, to persist the log's entry in its parent directory, and to persist its data file so it can reclaim the log. All of these `fsync`s are unnecessary when file system operations become persistent synchronously.



(a) Standalone **(b) Replicated**

Figure 8: Redis SET throughput.

5.5 Redis

Redis [12] is an example of a key-value store that is typically used in a replicated, distributed scenario. Redis either logs operations to an append-only-file (AOF) or uses an asynchronous snapshot mechanism. Only the AOF provides persistence guarantees for all operations, as snapshots are only persisted at larger time intervals.

Standalone. We start by benchmarking a single Redis instance. We configure it to use AOF mode and to persist data synchronously, before acknowledging a user's request. Figure 8a shows the throughput of SET operations using 12 byte keys and with various value sizes. Redis achieves up to 22% higher throughput on Strata, compared to NOVA for small values, which is the common case for key-value stores [18].

Replication. Redis supports replication for fault tolerance. Figure 8b shows the throughput of Redis with a single replica, which must persist its record before the master can acknowledge a request. Redis throughput drops by about half relative to the non-replicated case due to the extra network round-trip (see Table 3 for round-trip and persistence latencies). Strata improves throughput by up to 29% relative to EXT4-DAX and retains a 5% improvement over NOVA. Persistence in Strata is fast enough for use in high-performance network servers, and Strata's cross-layer management provides more capacity than an NVM-only file system.

6 RELATED WORK

Logging and coherence in file systems. WAFL [28] and ZFS [51] both have the capability to use logging to a low-latency medium (non-volatile RAM in WAFL's case) to reduce the latency of file system writes. Similarly, Frangipani [48] uses logging to support synchronous persistence of file metadata, but not data, in a distributed file system. xFS [17] uses an invalidation-based write-back cache coherence protocol to minimize communication overhead among applications and a centralized network server. Strata extends these ideas to provide low-latency kernel-bypass for applications on the local machine, synchronous persistence for data and metadata, and a lease mechanism to provide coherence among applications managing data at user-level. McoreFS [19]

uses per-core logs and operation commutativity properties to improve multicore file system scalability. Strata can leverage these same techniques to improve scalability if needed.

Multi-layer block stores. Various efforts have studied the use of caching among different storage technologies. Strata leverages similar ideas, in the context of a read-write file system. Operating with a file system API allows us to support, and requires us to handle, a broader class of application access patterns. For example, RIPQ [46] is a novel caching layer that minimizes write amplification when using local SSD as a read-only cache for remote photo storage. FlashStore [24] is a key-value store designed to use SSD as a fast cache between DRAM and HDD, similarly minimizing the number of reads/writes done to SSD. Nitro [33] is an SSD caching system that uses data deduplication and compression to increase capacity. Dropbox built a general-purpose file system that uses Amazon S3 for data blocks, but keeps metadata in SSD/DRAM [36]; technical details on its operation are not public. RAMcloud [40] uses disk as a back up for data in replicated DRAM. It applies log structure to both DRAM and disk [45], achieving higher DRAM utilization.

NVM/SSD optimized block storage/file systems. Much recent work proposes specialized storage solutions for emerging non-volatile memory technologies. BPFS [22] is a file system for non-volatile memory that uses an optimized shadow-paging technique for crash consistency. PMFS [25] explores how to best exploit existing memory hardware to make efficient use of persistent byte-addressable memory. EXT4-DAX [1] extends the Linux EXT4 file system to allow direct mapping of NVM, bypassing the buffer cache. Aerie [50] is an NVM file system that also provides direct access for file data IO, using a user-level lease for NVM updates. Unlike Strata, none of these file systems provide synchronous persistence semantics, as they require system calls for metadata operations. Only NOVA [52] goes one step further and uses a novel per-inode log-structured file system to provide synchronous file system semantics on NVM, but requires system calls for every operation. F2FS [32] is a SSD-optimized log-structured file system that sorts data to reduce file system write amplification; lacking NVM, it does not provide efficient synchronous semantics. Decibel [37] is a block-level virtualization layer that isolates tenants accessing shared SSDs by observing and controlling their device request queues. Strata generalizes these ideas to provide direct and performance-isolated access to NVM for both meta-data and data IO using a per-application update log, along with providing efficient support for much larger SSD and HDD storage regions. Strata also coalesces logs to minimize write amplification, which is new compared to these existing systems.

Managed storage designs. All storage hardware technologies require a certain level of software management to achieve good performance. Classic examples include elevator scheduling [6] and log-structured file systems [16]. Modern examples include log-structured merge trees [39] (LSM-trees) and B^{ϵ}-trees, used by various storage systems [34, 43, 53]. All of these systems rely on a particular layout of the stored data to optimize read or write performance or (in the case of LSM-trees) both. Unlike all of these systems, Strata specializes its data representation to different storage layers, changing the correctness and performance properties on a per-device basis.

Strong consistency. A number of approaches propose to redesign the file system interface to provide stronger consistency guarantees for slow devices. Rethink the sync [38] proposes the concept of *external synchrony*, whereby all file system operations are internally (to the application) asynchronous. The OS tracks when file system operations become externally visible (to the user) and synchronizes operations at this point, allowing it to batch them. Optimistic crash consistency [21] introduces a new API to separate ordering of file system operations from their persistence, enabling file system crash consistency with asynchronous operations. Strata instead leverages fast persistence in NVM to provide ordered and atomic operations.

7 CONCLUSION

Trends in storage hardware encourage a multi-layer storage topology spanning multiple orders of magnitude in cost and performance. File systems should manage these multiple storage layers to provide advanced functionality like efficient small writes, synchronous semantics, and strong QoS guarantees.

Acknowledgments. For their insights and valuable comments, we thank the anonymous reviewers and our shepherd Ashvin Goel. We acknowledge funding from NSF grants NSF-1518702 and CNS-1618563.

REFERENCES

[1] 2014. Supporting filesystems in persistent memory. https://lwn.net/Articles/610174/. (Sept. 2014).
[2] 2015. Linux control group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt. (Oct. 2015).
[3] 2017. Amazon S3. (Aug. 2017). https://aws.amazon.com/s3/.
[4] 2017. Apache ZooKeeper. https://zookeeper.apache.org. (Aug. 2017).
[5] 2017. Bandwidth: a memory bandwidth benchmark. (Aug. 2017). http://zsmith.co/bandwidth.html.
[6] 2017. Elevator algorithm. https://en.wikipedia.org/wiki/Elevator_algorithm. (Aug. 2017).
[7] 2017. Intel Optane Memory. (Aug. 2017). http://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html.
[8] 2017. lvmcache – LVM caching. http://man7.org/linux/man-pages/man7/lvmcache.7.html. (Aug. 2017).
[9] 2017. NVM Express 1.2.1. http://www.nvmexpress.org/wp-content/uploads/NVM_Express_1_2_1_Gold_20160603.pdf. (Aug. 2017).
[10] 2017. Persistent Memory Programming. (Aug. 2017). http://pmem.io/.

[11] 2017. Product Brief: Intel Optane SSD DC P4800X Series. (Aug. 2017). http://www.intel.com/content/www/us/en/solid-state-drives/optane-ssd-dc-p4800x-brief.html.

[12] 2017. Redis. http://redis.io. (Aug. 2017).

[13] 2017. rsockets library. (Aug. 2017). https://github.com/ofiwg/librdmacm.

[14] 2017. SPECsfs2014. (Aug. 2017). https://www.spec.org/sfs2014/.

[15] 2017. SQLite. https://sqlite.org. (Aug. 2017).

[16] 2017. The Sprite Operating System. https://www2.eecs.berkeley.edu/Research/Projects/CS/sprite/sprite.html. (Aug. 2017).

[17] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. 1995. Serverless Network File Systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 109–126. https://doi.org/10.1145/224056.224066

[18] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. London, England, UK, 53–64. https://doi.org/10.1145/2254756.2254766

[19] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[20] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. 1994. RAID: High-performance, Reliable Secondary Storage. *ACM Comput. Surv.* 26, 2 (June 1994).

[21] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 228–243. https://doi.org/10.1145/2517349.2522726

[22] 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)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[23] J. Dean and S. Ghemawat. 2011. LevelDB: A Fast Persistent Key-Value Store. https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html. (2011).

[24] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High Throughput Persistent Key-value Store. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 1414–1425. https://doi.org/10.14778/1920841.1921015

[25] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

[26] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2011. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 71–83. https://doi.org/10.1145/2043556.2043564

[27] T. Haynes and D. Noveck. 2015. Network File System (NFS) Version 4 Protocol. (March 2015). https://tools.ietf.org/html/rfc7530.

[28] Dave Hitz, James Lau, and Michael Malcolm. 1994. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC'94)*. USENIX Association, Berkeley, CA, USA, 19–19. http://dl.acm.org/citation.cfm?id=1267074.1267093

[29] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*.

[30] Intel Corporation. 2017. Storage Performance Development Kit. (Aug. 2017). http://www.spdk.io.

[31] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. 2010. DFS: A File System for Virtualized Flash Storage. *Trans. Storage* 6, 3, Article 14 (Sept. 2010), 25 pages. https://doi.org/10.1145/1837915.1837922

[32] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, USA, 273–286. http://dl.acm.org/citation.cfm?id=2750482.2750503

[33] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. 2014. Nitro: A Capacity-optimized SSD Cache for Primary Storage. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 501–512. http://dl.acm.org/citation.cfm?id=2643634.2643686

[34] Mike Mammarella, Shant Hovsepian, and Eddie Kohler. 2009. Modular Data Storage with Anvil. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 147–160. https://doi.org/10.1145/1629575.1629590

[35] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, Vol. 2. Ottawa, ON, Canada.

[36] Cade Metz. 2016. The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire. (March 2016). https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/.

[37] Mihir Nanavati, Jake Wires, and Andrew Warfield. 2017. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 17–33. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/nanavati

[38] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 1–14. http://dl.acm.org/citation.cfm?id=1298455.1298457

[39] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). In *Acta Informatica*.

[40] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (Aug. 2015), 55 pages. https://doi.org/10.1145/2806887

[41] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 1–16. http://dl.acm.org/citation.cfm?id=2685048.2685050

[42] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. 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)*. USENIX Association, Berkeley, CA, USA, 433–448. http://dl.acm.org/citation.cfm?id=2685048.2685082

[43] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 145–156. http://dl.acm.org/citation.cfm?id=2535461.2535480

[44] Andy M Rudoff. 2016. Deprecating the PCOMMIT Instruction. (Sept. 2016). https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[45] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX Association, Berkeley, CA, USA, 1–16. http://dl.acm.org/citation.cfm?id=2591305.2591307

[46] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, USA, 373–386. http://dl.acm.org/citation.cfm?id=2750482.2750510

[47] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *USENIX ;login:* 41, 1 (2016).

[48] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. 1997. Frangipani: A Scalable Distributed File System. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, New York, NY, USA, 224–237. https://doi.org/10.1145/268998.266694

[49] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. 2015. How to Get More Value from Your File System Directory Cache. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 441–456. https://doi.org/10.1145/2815400.2815405

[50] 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)*. ACM, New York, NY, USA, Article 14, 14 pages. https://doi.org/10.1145/2592798.2592810

[51] Scott Watanabe. 2009. *Solaris 10 ZFS Essentials*. Prentice Hall.

[52] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*. USENIX Association, Berkeley, CA, USA, 323–338. http://dl.acm.org/citation.cfm?id=2930583.2930608

[53] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2016. Optimizing Every Operation in a Write-optimized File System. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 1–14. https://www.usenix.org/conference/fast16/technical-sessions/presentation/yuan

[54] Yiying Zhang and Steven Swanson. 2015. A study of application performance with non-volatile main memory. In *31st Symposium on Mass Storage Systems and Technologies (MSST)*.