Aerie: Flexible File-System Interfaces to Storage-Class Memory

Haris Volos[†], Sanketh Nalli[*], Sankarlingam Panneerselvam[*],
Venkatanathan Varadarajan[*], Prashant Saxena[*] and Michael M. Swift[*]

[†]*HP Labs, Palo Alto and* [*]*University of Wisconsin–Madison*

haris.volos@hp.com, {sankey, sankarp, venkatv, prashant, swift}@cs.wisc.edu

Abstract

Storage-class memory technologies such as phase-change memory and memristors present a radically different interface to storage than existing block devices. As a result, they provide a unique opportunity to re-examine storage architectures. We find that the existing kernel-based stack of components, well suited for disks, unnecessarily limits the design and implementation of file systems for this new technology.

We present Aerie, a flexible file-system architecture that exposes storage-class memory to user-mode programs so they can access files without kernel interaction. Aerie can implement a generic POSIX-like file system with performance similar to or better than a kernel implementation. The main benefit of Aerie, though, comes from enabling applications to optimize the file system interface. We demonstrate a specialized file system that reduces a hierarchical file system abstraction to a key/value store with fewer consistency guarantees but 20-109% higher performance than a kernel file system.

1. Introduction

New device technologies such as phase-change-memory (PCM), spin-transfer-torque RAM (STT-RAM), flash-backed DRAM and memristors provide persistent storage near the speed of DRAM. These technologies collectively are termed *storage-class memory* (SCM) [20] as data can be accessed directly through ordinary load/store instructions rather than through I/O requests. Direct access reduces software layering overheads found in deep storage stacks of existing operating systems, which can be a major contributor of overhead to access latency for fast SCM.

Recent work has explored high-performance storage-system designs targeted for SCM. Proposed kernel-mode file-systems reduce access latency to SCM and provide faster persistence through less buffering and by removing the block driver layer [3, 14, 55]. Other designs improve data access performance by enabling user-mode access to file data [10, 18]. All this work improves file-system performance considerably while maintaining the traditional POSIX file-system interface and its benefits, such as legacy-support, naming, and protected sharing. Unfortunately, we show that this interface precludes applications from enjoying the raw fast persistence speed of SCM. Other work proposed exposing SCM directly to programmers by providing a persistent memory interface to SCM [13, 52]. Applications can tap into the performance of SCM but they also lose important file-system features such as sharing semantics and global naming.

We propose that SCM no longer requires the OS kernel to provide the file-system interface to storage. The standard load/store interface of SCM deprecates the need for abstracting details of the specific storage device through a kernel driver, and the kernel no longer needs to mediate every access to storage for protection as virtual memory hardware can protect access to individual data pages. Instead, we suggest that user-mode libraries should implement and provide the file-system interface and functionality. This can provide two key benefits: (i) low-latency access to data by removing layers of code, and (ii) flexibility by enabling applications to define their own file-system interface semantics and operations regarding metadata. For example, a mail message store that operates on many small files can have a get/put interface rather than open/read/write/close to reduce the setup cost when accessing files. Nevertheless, there are several challenges that such an approach needs to address, such as ensuring that a malicious or buggy user-mode client does not corrupt the file system, supporting sharing and concurrency between multiple untrusted clients, and enabling fine-grained permissions not possible with hardware alone.

Based on this idea, we designed the Aerie architecture to expose file-system data stored in SCM directly to user-mode programs. Applications link to a file-system library that provides local access to data and communicates with a service for coordination. The OS kernel provides only coarse-grained allocation and protection, and most functionality is

Eurosys 2014, April 13–16, 2014, Amsterdam, Netherlands.

http://dx.doi.org/10.1145/2592798.2592810

distributed to client programs. For read-only workloads, applications can access data and metadata through memory with calls to the file-system service only for coarse-grained synchronization. When writing data, applications can modify data directly but must contact the file-system service to update metadata.

We implement two file-system interfaces on the same layout with Aerie: (i) PXFS, a POSIX-style file system optimized for sequential sharing of files, and (ii) FlatFS, a specialized file system optimized for small-file access through a put/get interface. Our experiments show that PXFS performs between only 22% slower than RamFS, which lacks consistency guarantees, and 17% faster on average than ext4. Thus, a standard file-system interface based on Aerie performs sufficiently well to replace a kernel file system. This is important as it enables the adoption of user-mode library file systems and any further interface optimizations enabled through them. The specialized FlatFS interface performs up to 45% faster than the fast kernel file system without crash consistency and 109% faster than a kernel file system with crash consistency. Thus, distributing file system functionality to client processes allows flexible implementations that dramatically improve performance.

2. Storage-Class Memory

Emerging storage-class memory (SCM) technologies promise to change many assumptions about storage. They have the persistence of storage but the fine-grained access of memory, and can be attached to the memory bus and accessed through load and store instructions [20]. Four recent technologies provide SCM capabilities: phase-change memory (PCM) [36], spin-transfer-torque MRAM [31], flash-backed DRAM [49], and memristors [47]. While the performance and reliability details differ, they all provide byte-granularity access and the ability to store data persistently across reboots without battery backing. SCMs are currently commercially available in modest sizes of up to 8GB [49].

The unique properties of SCM enable direct access from user mode. As memory, SCM can be protected by existing memory-translation hardware. Furthermore, it has much less need for scheduling to optimize latency, as there are no long seek or rotation delays. Because SCM provides speeds near DRAM, shared caching may not be as necessary. Finally, SCM does not require a driver for data access as it can implement a standard load/store or protected DMA interface [10]. Previous work on kernel-mode file-systems, including BPFS [14], SCMFS [55], and PMFS [3], has proposed removing some of the kernel layers, such as scheduling and drivers, to reduce access latency to SCM and provide faster persistence. Quill [18] and Moneta-D [10] further improve data access performance by enabling user-mode access to file data through memory mapping files and user-mode DMA respectively. While these works improve file-system performance considerably, the fixed and inefficient POSIX interface can limit the benefits (Section 3).

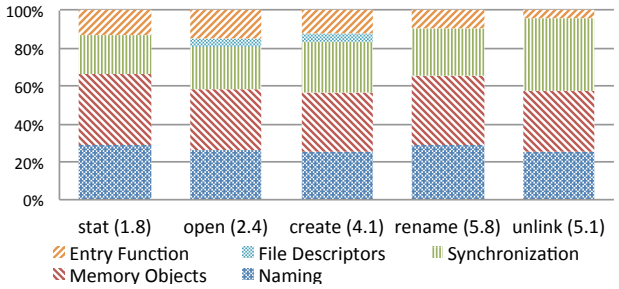

Figure 1: Breakdown of the time spent in the Linux Virtual File System (VFS) layer. *Open and create include a close operation. Numbers in parentheses represent the average execution time in microseconds.*

3. The Abstraction Cost of a File

The Unix file-system architecture and POSIX file-system interface introduce two sources of overhead. First, there is a cost of changing modes and cache pollution from entering the kernel [46]. Second, there is a cost due to the generality of the file-system interface that abstracts every system resource as a file. Generality offers programming convenience but comes at the cost of implementing interoperability between resources, such as disk files and network sockets, by associating all resources with common generic data structures, namely reference-counted file descriptors, in-memory *inodes* and *dentry* objects. For slow disks, abstraction comes at a relatively low cost. However, for fast SCM, similarly to fast networking [27], abstraction becomes expensive as the I/O cost is substantially lower.

We quantify the file abstraction cost by measuring the cost of common file-system operations. Our focus is on namespace operations, which are metadata intensive. We perform each set of measurements over 1 million files organized in a 3-level-deep hierarchy and stored in an `ext4` file system on a RAM-disk. Our experiments start with cold inode and dentry caches to capture the cost of creating such in-memory objects. We use the `perf` profiling utility to obtain a breakdown of the time spent in the virtual file system (VFS) layer of x86_64 Linux 3.2.2 running on 2.4GHz Intel Xeon E5645 six-core machine.

Figure 1 shows this breakdown organized in five categories: (i) *entry function* for the time spent in the main routine of each VFS operation including the system call overhead, (ii) *file descriptors* for the cost of managing file-descriptors. (iii) *synchronization* for the cost of executing synchronization operations such as read-copy-update (RCU) and lock operations, (iv) *memory objects* for the cost of managing in-memory inodes and dentries, and (v) *naming* for the cost of hierarchical names.

We make two main observations. First, we observe that even a simple `stat` system call takes 1.8μs, which is about

25x slower than reading from PCM with a read latency of 70ns. Second, on average 87% of the VFS time is spent in supporting generic semantics through generic synchronization, generic hierarchical names, and in-memory objects. Specifically, synchronization needed to support concurrent operations and concurrency semantics accounts for 26% of VFS time on average, even though our measurements *use a single thread of execution*. Thus, this cost represents the uncontended best case and we expect this to rise with a larger number of threads. Generic hierarchical naming consumes on average 27% of VFS time. The majority of the time is spent in looking up and resolving each path-name component, including access control. This cost is proportional to the path components that need to be resolved and we expect this overhead to become worse with deep naming hierarchies that often arise in modern file systems [5]. Finally, the remaining 34% of VFS time is associated with managing in-memory inodes and dentries, which includes the cost for allocating, initializing, accessing, reference counting, and destroying such objects.

We believe the above observations have interesting implications for the design of file-system interfaces for fast SCM. While in-memory objects (*e.g.*, inodes and dentries) help improve performance of slow storage at a relative low cost, they add substantial cost with fast SCM and eradicating them may reduce latency. However, current file-system semantics often *require* in-memory objects: for example, POSIX sharing semantics allows files to be unlinked while open, which requires data structures to track users of open files. To the question *"was the 'unlink but leave around for anyone with it open' a motivation for inode's in Unix or just a lucky/unlucky consequence?"*, Ken Thompson's concise response *"it was deliberate"* reiterates that the use of in-memory inodes to support this feature has not been accidental [1]. When originally proposed, the performance cost of these semantics was negligible and they provide obvious utility. However with fast storage, their performance cost is high for applications that do not need them. We would therefore like to have applications pay this overhead only when needed.

Similarly, to remove unnecessary naming overheads, we would like to be able to support other naming structures in addition to hierarchical names, such as flat tag-based names or object namespaces. This would help applications that today layer their own naming solution on top of a file-system namespace, such as photo stores [9] and IMAP mail servers [16], avoid paying the extra overhead for the underlying file system namespace.

In summary, the file-system interface provides useful features, such as organizing data under a global logical namespace for easy access and protecting data for secure sharing between applications. Moving forward, we believe that such features will remain relevant. However, limiting ourselves to accessing these features via a single generic interface and semantics (as we do today with the POSIX interface) may unnecessarily hurt performance.

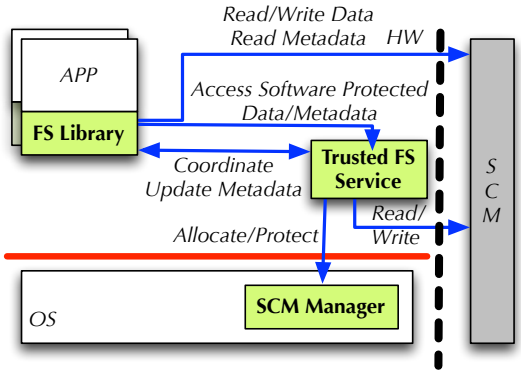

Figure 2: Decentralized Architecture. *Functionality is split between a user-mode library, a trusted service, and the kernel.*

4. Design

We designed Aerie with the main goal of providing applications with flexibility in defining their own file-system interface and implementation for high-performance access to SCM.

4.1 Direct Access

The main enabling mechanism for Aerie is direct access through memory to file-system data and metadata from user-mode file-system libraries. Direct access to *metadata* enables flexibility as the user-mode library implementation can optimize interface semantics and operations regarding metadata to the specific needs of the class of applications it targets. Direct access also avoids the costs of calling into the kernel [46].

With direct access though, a malicious program can corrupt and violate file-system invariants, such as inserting two files with the same name in a directory. Fundamentally, addressing this concern requires a trusted entity [33, 45] or that all clients validate metadata [40]. One design approach would be to resort to a purely centralized service that mediates every access to the file system. However, such a design would require frequent invocations to the trusted service [38] and eliminate the performance benefits of direct access. Aerie relies on a combination of hardware and software to efficiently address this challenge as we describe next.

4.2 Decentralized Architecture

Figure 2 illustrates Aerie's decentralized architecture. Aerie distributes the file system implementation between an untrusted user-mode library (libFS) and a trusted file-system service (TFS). The library provides the file-system interface, including naming and data access. The service supports cooperation between mutually distrustful programs by enforcing metadata integrity and synchronization. Our design optimizes for the common case of sequential sharing rather than concurrent [4] to reduce communication with the service.

Programs with concurrent access may be better served by a centralized file system. Aerie reduces the kernel's role to just multiplexing physical memory via the SCM manager.

In order to support multiple file systems, an OS may run multiple implementations of the library and the service, one for each file system layout. The kernel code, though, is common to all file systems. In addition, a single file-system layout may have multiple libFS implementations optimized for different workloads. In such a case, a single TFS may contain interface-specific logic to manage metadata and synchronization for each interface it supports.

libFS Client Library. Applications link against a libFS library for each file-system interface they use. Virtual-memory protection hardware enforces access control over file system data and read-only access to metadata that the client is allowed to access. This allows the client library to service most file-system operations directly from SCM without contacting a trusted service. Specifically, the library provides functionality to find and access data: lookup to map file names to file metadata, and indexing to translate a file offset into a byte in memory. For example, when an application opens a file, the library accesses directory contents in SCM to locate the file and can then read file data directly from SCM as well. The client also implements logic to invoke the TFS for operation on file-system state that cannot be enforced by hardware, such as enforcing file-system invariants over metadata. Clients can batch metadata updates to further reduce communication with the TFS.

Trusted File System Service (TFS). Functionality that requires a trusted third party but *not* privileged hardware access, such as integrity for metadata updates and concurrency control between processes, executes in the TFS. For concurrency control, the TFS provides a distributed lock service that issues leases to clients. The service also provides complete file system functionality on data for which memory protection is too coarse, as in the case of write-only files that typical memory protection hardware cannot support. TFS runs as a user-mode process accessed via RPC.

SCM Manager. Operations requiring hardware privileges, such as modifying memory permissions or virtual address mappings, must execute in the kernel. The SCM manager provides a storage allocation mechanism that is independent of high-level file system organization. Specifically, it provides an interface to allocate large chunks of SCM to a file-system volume, mount a file system by mapping it into a client process, and apply protections for client processes.

4.3 Putting it All Together: The Life of a Shared File

To demonstrate how everything fits together, we present an example where a process creates a file, writes some data, changes permissions, and then another process reads the file and finally deletes it. To create the file, the first process invokes the lock service via RPC to acquire locks on the directory that will contain the new file and on the file itself. To allocate space for storing file metadata and data, the process calls the TFS, which in turn calls into the kernel SCM manager to allocate and map SCM pages to the file system. The process can then write data directly without further invoking the service but logs any metadata modifications required for creating a directory entry, changing permissions, and resizing the file. When another process opens and reads the file, that process invokes the lock service to acquire necessary directory and file locks. The service then revokes locks held by the first process, at which point the process sends outstanding metadata updates to the service. The service applies the metadata updates before it hands the locks to the read process. The reading process can then read file data directly. When done, it deletes the file by logging metadata modifications to remove the file's directory entry and marking allocated space as free.

If a client fails with outstanding updates, TFS revokes locks held by the client. This implicitly discards the client's outstanding metadata updates that it did not ship to the service. This enforces metadata integrity but allows client newly written data to be lost if it was not linked to a file. Clients can avoid losing new data by forcing shipping metadata via the `libfs_sync` interface, which is the library equivalent of `fsync`.

4.4 Limitations

Aerie relies on the memory controller or memory device to implement necessary reliability mechanisms to address wear and that such mechanisms are robust to malicious attacks that aim to overwhelm the anti-wearout mechanism [13, 42].

5. Implementation

We implemented Aerie on Linux 3.2.2 for x86-64 processors. In this section we present the implementation of the kernel SCM manager and the functionality of user-mode components TFS and libFS for key file system features. We defer the discussion of two file system interfaces to Section 6. The SCM manager comprises 650 lines of C code and the user-mode components comprise 16,930 lines of C++ code.

5.1 Infrastructure Services

Aerie is relies on a set of infrastructure mechanisms for interprocess communication, consistently updating data in SCM, and distributed concurrency control.

Interprocess Communication. We use remote-procedure call (RPC) implemented using sockets on the loopback interface for communication between clients and the server. The server is multithreaded and can handle multiple RPC requests concurrently. Batching of metadata operations at a client (Section 5.3.5) helps take RPC off the critical path for most operations. An RPC implementation based on a design compatible with recent operating system redesigns for

many-core processors could further help reduce the cost of communication [8].

Persistence Primitives. We borrow the persistence primitives from Mnemosyne [52] to support consistently updating file system structures in SCM in the presence of failures. We implement them through regular x86 instructions and provide three basic operations: (1) *wlflush* uses x86 `clflush` to write and flush a cache line out of the processor cache into SCM for persistence, (2) *bflush* uses x86 `mfence` to flush the processor write-combining (WC) buffers into SCM for persistence, and (3) *fence* uses x86 `mfence` to order writes to SCM. We assume that the memory controller guarantees atomicity of 64-bit updates [14]. We use these primitives to implement our higher-level consistency mechanisms and a persistent log for redo logging. Writes to the log are done using x86 streaming instructions (which buffer writes in WC buffers and enable high bandwidth for sequential writes). Flush of the log writes to SCM is done through `bflush`. Hardware support for committing transactions [12, 24] would reduce the overhead of consistency.

Distributed Lock Service. We implement distributed concurrency control with a centralized lock service executing in the TFS service. The lock service provides multiple-reader, single-writer locks identified by a 64-bit identifier. Our implementation derives from prior lock services for storage systems [25, 26, 32, 50]. However, because our lock service is intended for a single machine, we do not replicate the service for fault tolerance. Aerie does not use Linux's futexes [17] because it must be able to revoke locks.

Clients access the lock service via a local clerk in libFS. When a client thread requests a lock, the clerk invokes the lock service to acquire a global lock that synchronizes the client with other processes. The clerk then issues a local lightweight mutex that client threads use to synchronize within the process. When another process requests conflicting access to the lock, the service calls the clerk back to revoke the lock. The clerk may hold the lock after a thread releases the local mutex. It releases the global lock when it has not been used recently or when the lock service calls back to revoke the lock. If the lock is in use when a callback arrives, the clerk prevents additional threads from acquiring the local mutex and releases the global lock when the local mutex is released. Clients of the lock service are responsible for preventing deadlocks by ordering or preempting locks.

An unresponsive client can deny service to the file system due to bugs or malicious behavior. This occurs in any system with mandatory file locks, such as Windows. Aerie addresses denial-of-service by attaching a *lease* to each lock that must be renewed by the clerk [25]. A client that does not renew its lease implicitly releases the lock and allows other processes to proceed.

5.2 SCM Manager

The SCM manager is a kernel component, whose responsibility is allocation, mapping, and protection of SCM.

Allocation. The SCM manager is designed for allocating a small number of large static memory partitions. It allocates contiguous regions of physical memory using first-fit. The SCM manager stores a table listing each partition and an access control list indicating who can modify or access the partition, typically the TFS. As with all data structures in Aerie, the SCM manager stores the partition table in SCM and uses persistence primitives to assure consistent updates.

Mapping. Once allocated, a partition can be mapped into any process with the `scm_mount_partition` API. In order to reduce the overhead of page tables, the SCM manager uses a linear mapping of physical addresses that can be computed from a single virtual base address (similar to Direct Segments [7]), and maps SCM at the same virtual address in all processes. Thus, mounting a partition does not populate the page table but instead leaves that to ensuing page faults. This effectively treats the page table as a giant software TLB, similar to Mach's `pmap` structure [43]. As a result, page tables are dynamic structures that need not be stored in SCM.

The SCM manager further reduces the space overhead of the page table by aggressively sharing page tables between processes. All processes with the same access to files—those with the same user and group IDs—share the entire page table branch mapping SCM.

Protection. The unit of protection in Aerie is the *extent*, which is a range of memory within a partition associated with protection rights assigned by higher-level software. The SCM manager stores extents in a radix tree corresponding to the page-table layout. Each extent consists of a starting address, length, and a 32-bit ACL identifier. The 30 high bits represent a group identifier (GID) and the lowest 2 bits represent the memory protection rights (read, write). The `scm_create_extent` API takes a starting address and length in pages and creates an extent structure. The `scm_mprotect_extent` changes the protection on an extent. Only processes with write access to a partition can manipulate extents. At run time each process inherits and maintains the user's group memberships in a hash table. On a fault, the manager uses the GID of the extent as a key in the hash table to quickly decide if the process has access to the extent.

Changing permissions on an extent is more expensive than changing permission on a file because permissions must be changed for all pages in the file and for all clients of the file system. To avoid synchronously modifying many page tables, the SCM manager instead invalidates portions of the page table mapping the affected extents (if they were valid), and allows them to be faulted back in later. Thus, clients implicitly communicate with the kernel to reload mappings when protection changes.

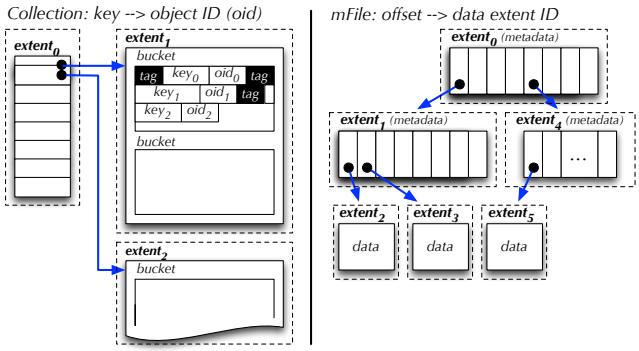

Figure 3: Collection and mFile objects. *Link pointers connecting extents store the virtual address of an extent. Dashed rectangles around extents show memory protection. All extents comprising an object share the same ACL.*

We borrow a technique from single address space operating systems to handle page faults [11]. When a page fault occurs, the SCM manager computes a new page table entry from the linear mapping and the permissions stored in the extent tree. On processor architectures with support for separating protection from addressing [35], only a single page table would be needed.

5.3 File System Features

libFS and the TFS cooperate together to provide the functionality for key file system features.

5.3.1 Naming

Aerie implements mechanisms for low- and high-level naming. For low-level naming, each file-system storage object (described below) is identified by a 64-bit *storage object ID* (OID). The six least-significant bits encode the type of the object and the remaining 58 bits encode the virtual memory address where the object is stored, which corresponds to the virtual address of the head extent of the object. This encoding enforces a minimum object size of 64-bytes and provides 64 different types. As a result, locating an object given its OID requires no lookup of its address, but it cannot be relocated in memory. We did not find the lack of relocation to be an issue in the file systems we implemented. Objects can grow arbitrarily large without having to be relocated because they are not linear regions of virtual memory but are a structure composed of multiple extents.

Aerie provides the *collection* object type for building higher-level naming structures, such as directories. The collection object provides an associative interface for storing key-value pairs. We implement collections as a linear hash table that is packed into extents as shown in Figure 3. The hash table is organized into fixed-size buckets, which store key-value pairs. A key is an array of bytes of arbitrary length and the value field stores a 64-bit storage object ID. A tag preceding the key records the key length. When the hash table fills, we attach additional extents and rehash some existing elements into the new extents. We perform consistent updates using shadow updates, so new extents are allocated and populated and then linked into the hash table with a single 64-bit atomic write to a pointer. We delete items by marking them using a tombstone key. When the number of tombstones rises above a configurable threshold, we rehash the live key-value pairs into a new table and then update the collection's header to point to the new table with a single 64-bit atomic write.

The untrusted library can safely read collection contents directly without communicating with the service. Thus, directory operations such as traverse, read, and open are handled by locating collections and reading them directly.

5.3.2 Indexing

Aerie provides the *memory file* (*mFile*) object type for mapping offsets to extents. mFiles provide access to a range of bytes starting at a specified offset and can be used to implement data files. We implement the mFile as a radix tree of indirect blocks that point to fixed-size extents. Larger extents are broken into pieces when added to the tree. Similarly to collections, clients can read mFiles without invoking the service. Thus, clients can locate and read/write file extents directly.

5.3.3 Protection

Aerie requires a client to assign protection to storage objects through the service to ensure that all objects' extents receive the same protection rights. The service uses the `scm_mprotect_extent` API to propagate protection down to the object's extents.

Although hardware-enforced protection enables us to efficiently enforce file-system level permissions, it may be too rigid to directly represent the whole spectrum of permissions. Memory typically grants read or read/write access, while files may have write-only access. In addition, metadata may have semantically richer permissions, such as directory *list* and *traverse*. Enabling, for example, both list and traverse can be enforced in hardware through read access to allow reading directory contents. However, enabling only one the two, for example traverse-only, requires hardware to enforce no-access to prevent listing directory contents, which prevents both.

Thus, the library can directly access any storage memory allowed by memory protection. Since protection is stricter than permissions, the library calls into the TFS for any operations allowed by file system level permissions but prevented by memory protection, as in the case of write-only files or traverse-only directories.

5.3.4 Concurrency

Aerie performs concurrency control over file-system objects using the distributed lock service (Section 5.1). We assign a unique global lock to every object. A client requests a lock that covers an object, and can use that until it is done or the

lock is revoked. The TFS requires that clients hold a lock in write mode covering an object when sending metadata updates to the TFS. Clients, though, can treat locks as advisory because data is already available through memory. A file system interface is free to provide applications any degree of consistency and concurrency semantics. Thus, a client library can choose when to acquire locks and how long to hold them to implement different consistency levels [19, 26]. For example, Windows file systems provide mandatory locking, while POSIX file systems do not.

File system implementations may organize locks hierarchically to reduce calls into the service. When acquiring a hierarchical lock, clients can access descendants of an object without additional calls. We extend the basic lock manager to provide three modes for each lock: *explicit*, meaning the lock covers only a single object; *hierarchical*, meaning it covers the object and its descendants, and *intent*, meaning that the object is not locked, but a descendant may be. The clerk in libFS implements the hierarchical locking logic. If it holds a hierarchical lock, the clerk answers requests for locks on descendant objects locally and issues local mutexes. For example, a client can lock a directory of files using a global lock and then acquire local mutexes on individual files. The clerk de-escalates in response to revocations [32]. When another thread requests conflicting access to a resource protected by a hierarchical lock, the clerk will request locks lower in the hierarchy and release the high-level lock.

When an object is a member of multiple collections, such as a file hard linked to multiple directories, hierarchical locking no longer works. The classic solution would be to lock each collection from which the file is accessible [26]. However, this approach requires finding those collections, which introduces complex bookkeeping. Instead, we follow a novel locking protocol where clients do not need to lock each collection but instead explicitly lock just the object. Each object has a membership count that clients use to detect when explicit locking is needed. The service updates the membership count when it adds or removes an object from a collection. The transition from hierarchical locks to explicit is safe because it requires an exclusive lock on both collections, which prevents concurrent reads.

5.3.5 Integrity

For metadata integrity, Aerie requires clients apply their metadata updates through the TFS. Before the TFS performs updates, it validates that these maintain file system invariants, such as ensuring that rename operations do not cause cycles in the namespace or that files map only allocated extents.

Naïvely having a client synchronously call into the TFS for each metadata update can negatively impact performance for metadata-intensive applications. We optimize by observing that a client can delay communicating metadata updates to the TFS until such updates need to be visible to other clients. Clients buffer their updates locally in a log that they send to a server periodically (similar to delayed writes) or when they must release a global lock. Locks ensure that clients do not journal and batch conflicting operations.

Each log entry has a header identifying the operation, the identifiers of the objects it modifies, and fields the operation updates. The ability to log operations also enables the TFS to benefit from work done at the client. For example, when the client performs a file append that requires allocating new storage, it logs each extent it pre-allocates and writes the data to the new extents. The server then only has to verify each allocation and attach each extent to the file rather than having to allocate storage, write the data, and attach the extents to the file.

Validating a client's updates involves two steps: (i) validate that messages have a valid structure, correspond to known operations, and that the operation maintains invariants, and (ii) verify that the client holds necessary locks and permissions. Recent work has shown that file-system integrity constraints can be enforced using fast local checks on the data being modified [21].

5.3.6 Crash Recovery

The TFS uses write-ahead logging implemented using a redo log to atomically perform multiple metadata updates. The server first logs each metadata update, flushes the log, and issues a fence to ensure following writes are ordered after the log writes. It then writes and flushes metadata using `wlflush`. In case of a crash, the TFS can recover by replaying the log of metadata updates. The server does not need to reacquire locks as updates were written and ordered in the log with locks held. The server addresses client failures by revoking locks held by the client. Outstanding metadata updates a failed client has not yet shipped to the service are implicitly discarded by the server. This guarantees metadata invariants but allows client data to be lost.

5.3.7 Free Space Management

The TFS implements a buddy storage allocator to create extents out of a partition. Clients do not allocate storage directly through the buddy allocator. Instead, libFS pre-allocates a pool of 1000 collections, 1000 mFiles, and 1000 extents to avoid contacting the service for create or append operations. Similarly to WAFL [28], the service maintains special files that track pre-allocated objects owned by each client to prevent memory leaks.

6. File Systems Interfaces on Aerie

A major goal of Aerie is to provide a substrate for flexible file-system design. To demonstrate this capability, we built two file system interfaces. The first, PXFS, shows how that Aerie has the functionality to implement a POSIX-style interface for compatibility with existing code. The second, FlatFS, shows how to optimize the interface for a specific workload. Each interface provides its own library but both

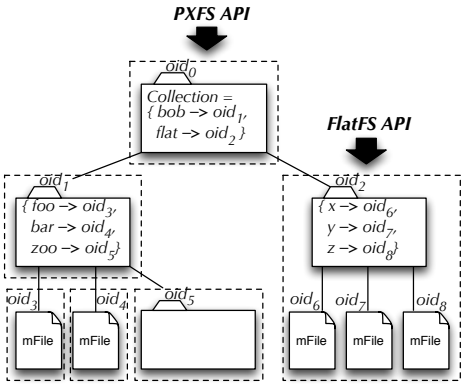

Figure 4: PXFS and FlatFS object layout. *Dashed rectangles around objects show memory protection.*

interfaces share the same TFS. Moreover, each interface has its own code routines in the TFS to manage interface-specific metadata and synchronization. Such code builds on top of a common substrate that provides common functionality for metadata management and synchronization.

6.1 Functionality: PXFS File System

PXFS provides most POSIX semantics for files and directories, including moving files across directories, retaining access to open files after its permissions change or it is unlinked, and permission checks on the entire path to a file. It does not provide asynchronous update of timestamps or predictable file-descriptor numbers. PXFS comprises 2,660 lines of C++ code.

Storage Objects. We implement POSIX storage objects directly with mFiles and collections as shown in Figure 4. Files are mFiles with page-sized extents and directories are collections mapping file names to the object IDs of files and directories. A root collection holds the root directory. PXFS creates a volatile shadow object in the client when opening a file for write to buffer metadata writes before sending them to the TFS.

Naming. We implement a hierarchical namespace by organizing the directory collections into a tree. To create a file within a directory, a client creates an mFile, acquires a write lock on the directory's collection, and then inserts the name and mFile's object ID. To atomically rename a file between directories, PXFS acquires write locks on old and new directory collections, inserts a name-file ID pair in the new destination and removes the name-file ID from the old collection. Acquiring multiple locks may lead to deadlock, so PXFS acquires both two locks before the rename operation and releases them if the TFS revokes a lock. Since acquiring write locks on collections forces other clients that hold locks on the collection to send their modifications to the service, directory updates that happen near the root directory may be slow.

PXFS supports both absolute and relative path resolution. Absolute paths are resolved starting at the root by recursively acquiring a read-only lock on each directory collection until the name is resolved. Relative paths are resolved starting at the working directory by recursively acquiring locks up or down the directory hierarchy to prevent concurrent renames of directories higher up the tree.

File sharing. PXFS supports concurrent file access. When a client opens a file, it acquires a lock on the file's mFile, which it holds until it closes the file. To allow files to be unlinked while open, the PXFS TFS maintains a table of open files for which at least one client had its lock revoked. Specifically, when another client requests the lock on an open file, clients with the file open notify the service that the file is open when releasing the lock. The service then adds the file to a collection of currently open files. The client can still obtain explicit locks on the mFile to read or write data, and when the client terminates or notifies the service that it has closed the file, the service reclaims the file's memory. If a client fails to notify the TFS of an open file, it will be unable to obtain subsequent locks on the file and thus any metadata updates will be rejected.

Permission changes are handled similarly: memory protection is updated synchronously when the permissions change, but processes with the file open notify the service. They can then access the file through the service over RPC. This approach to sharing is similar to Sprite's support for consistent read/write sharing [41], which reverts to sending requests to the server when there are conflicting concurrent accesses to a file. As POSIX specifies that permissions are enforced along the path to a file (Windows by default does not), PXFS updates the protection on all objects underneath a directory when its permissions change.

Caching. The major optimization in PXFS is a per-client in-memory cache of path names to speed name resolution. The cache is organized as a hash table that maps absolute path names to storage objects, and benefits applications that suffer from slow lookups due to long absolute paths. Accesses using relative path names do not use the cache, as the path names tend to be shorter. A directory path maps to the directory's collection, and a file path maps to the file's mFile object. Hash table entries are created on demand after resolving each path component. Entries are removed when a directory or a file is unlinked or when the cache grows too large. Concurrent accesses to the cache by threads within the same client process are synchronized using a regular process-private spin lock. The cache is optimized for workloads with infrequent sharing between processes: we provide cache consistency between processes by flushing the entire cache whenever the client releases a lock or the TFS revokes a lock (a simple optimization would be to drop only the paths covered by the lock).

Discussion. The PXFS design illustrates several optimization opportunities provided by Aerie. For example, read-only access to files only communicates with the TFS to ac-

quire locks, and if there are no conflicting accesses, a coarse-grained lock high in the file system tree suffices. Similarly, the client can write to file data locally, including writing new data to files, and only communicates with the service for metadata changes such as creating or appending to a file. Batching metadata changes further reduces communication. The cache design is optimized for non-shared access using absolute path names.

We found that supporting POSIX semantics increases the complexity of the implementation. For example, to retain open files that have been unlinked, clients must communicate with the server to indicate when files are opened or closed. While we chose to provide this feature to support applications that depend on it, the performance cost when files do not need to be cached may be excessive. For example, many network file systems, such as NFS [44] and AFS [30], relax consistency semantics.

6.2 Optimization: FlatFS

To demonstrate how targeting a specific application can further improve performance by changing the file-system interface and semantics, we designed FlatFS to provide (i) a simple storage model and (ii) a key-value store interface targeting applications that store many small files in a single directory, such as an email client or wiki software. Clients have a shared consistent view to files through a flat key-based namespace and access files through a simple put/get/erase interface. In addition, all files have the same permissions. In contrast to PXFS, FlatFS does not support POSIX semantics, such as hierarchical namespace, unlinking or renaming open files, and multiple names for a file. FlatFS comprises 340 lines of C++ code.

Storage Objects. FlatFS files are implemented with mFiles containing a single extent holding the entire file contents, which optimizes for small files with a known maximum size. The mFiles store no other metadata, such as permissions or access time, optimizing for the common case where all files in a directory have the same permissions. We further optimize for workloads that do not use human-readable names by replacing the hierarchical namespace with a single collection that maps file names to mFiles, as shown on the right of Figure 4. No name caching is needed because of the flat namespace. Adding or removing files, though, are metadata modifications that eventually go to the TFS.

File sharing. Like PXFS, FlatFS optimizes for scalable concurrent access to the large, flat key-based namespace through hierarchical locks. A single lock covers the whole collection and multiple locks under the single lock cover the extents that comprise the hash table of the collection (PXFS uses a single lock for a directory). Each extent's lock also covers the files linked from the key-value pairs stored in the extent. Operations acquire the single collection lock in intent mode, and then acquire the lock covering the extent where the key-value pair is stored. Insert and delete operations acquire a write lock while lookups acquire a read lock. When an insert or delete causes a rehash of the table, the rehash operation acquires the single lock covering the whole collection in write mode.

Discussion. FlatFS and PXFS use the *same memory layout* and differ in the policies the interface layer uses to allocate and synchronize data. An application can use either interface to access files. To PXFS, the underlying collection appears as a single global directory, and individual files can be accessed using the standard open/read/write/close operations with the usual semantics. However, when an application accesses files through the FlatFS interface, it benefits from important performance optimizations. First, the get/put interface opens a file and returns its data in a single operation, which removes the need to maintain state about open files in memory. Second, fixed-size files simplify storage allocation and reduce the amount of metadata. Third, the flat key-based namespace removes the cost of complex hierarchical path name resolution. Finally, hierarchical locks enable scalable concurrent access to the flat namespace. An alternative model to FlatFS would be to implement a key-value store as a single large file. FlatFS, in contrast, enables mutually distrustful programs to concurrently access and update files, such as for indexing or backup/restore.

7. Evaluation

We evaluate two main questions about Aerie.

1. *Generic POSIX performance:* Does a standard file-system interface based on Aerie perform fast enough to replace kernel file systems?
2. *Optimizations:* Can Aerie provide new opportunities for optimizations exceeding kernel FS performance?

7.1 Methodology

We performed our experiments on a 2.4GHz Intel Xeon E5645 six-core (twelve thread) machine equipped with 40GB of DRAM running x86-64 Linux kernel 3.2.2.

Storage class memory. We emulate SCM using DRAM. Separately from our main experiments, we study the sensitivity of Aerie to slow SCM performance using a simple performance model that slows down accesses through software-created delays (Section 7.4). We use a 24GB memory partition for all test configurations, as the test workloads do not require more.

Workloads. We compare Aerie against three Linux kernel-mode file systems: RamFS, ext3, and ext4. RamFS uses the VFS page cache and dentry cache as an in-memory file system. RamFS does not provide any consistency guarantees against crashes; thus it serves as a best-performing kernel-mode file system. To compare against file systems that provide crash consistency, we use ext3 and ext4. Both ext3 and ext4 are mature, production-quality file systems optimized for disk block devices and provide crash consistency using journaling. We use two mature file systems to better position

Benchmark	Latency (μs)			
	PXFS	RamFS	ext3	ext4
Sequential read	0.65	0.58	0.65	0.57
Sequential write	1.2	1.2	1.5	1.2
Random read	1.2	1.1	4.2	4.2
Random write	1.1	1.4	3.1	2.5
Open	1.2	1.3	1.6	1.6
Create	5.5	3.0	65.6	81.2
Delete	3.6	2.3	10.5	17.4
Append	3.4	1.1	5.6	3.5

Table 1: Latency of common file system operations. *All read-/write operations use a 4096-byte buffer.*

Aerie's performance. We mount ext3 and ext4 file systems on a Linux's RAM disk (`brd` device driver) that we modified to perform block writes using streaming writes and flush blocks using `blflush`.

We wrote our own microbenchmarks that stress specific file operations. For application-level workloads, we use a modified version of FileBench [2] that calls through libFS rather than system calls. Unless otherwise specified, workloads are single threaded. For all our experiments we report averages of at least five runs.

7.2 Generic POSIX Performance

A prime motivation for Aerie is that direct access to storage can enable optimizations that make user-mode file systems faster than ones in the kernel. However, for programs that do not benefit from optimized interfaces, performance cannot suffer compared to standard file-system designs. We show that despite higher communication and synchronization costs due to the TFS, Aerie can deliver POSIX performance comparable to kernel-mode file systems.

7.2.1 Microbenchmark Performance

We evaluate the latency of common POSIX file-system operations. The sequential tests operate on a 1GB file in 4KB blocks, and the random workloads randomly access 100MB out of a 1GB file in 4KB blocks. Open/create/delete are measured by opening/creating/removing 1024 4KB files. Because Aerie batches updates, we report average latency. The kernel-mode file systems operate with warm inode and dentry caches.

Table 1 shows the latency of common file system operations on PXFS, RamFS, ext3, and ext4. As expected RamFS performs consistently better than ext3 and ext4. Compared with ext4 in the kernel, PXFS is between 2% to 90% faster (average 47% faster) for all operations except sequential reads. ext4 benefits from variable length extents for storing file data that improve sequential access performance. Open is faster for PXFS because ext4 has to bring the file into the inode cache. PXFS benefits by not calling into the kernel, which helps random reads, and by batching metadata updates for create, delete, and append. PXFS compares similarly to ext3.

Benchmark	Latency (μs)				
	PXFS	PXFS-NNC	RamFS	ext3	ext4
Fileserver	16.8 (17.9)	24.3	13.1	30.3	18.7
Webserver	3.0 (3.3)	5.5	3.2	3.3	3.3
Webproxy	3.5 (3.5)	4.0	3.1	4.9	4.5

Table 2: Average latency per workload operation. *PXFS-NNC is PXFS with no name caching. Numbers in parentheses show the 95-percentile latency.*

PXFS performs close to RamFS, which runs in the kernel but provides no crash consistency guarantees. RamFS is between 8% to 67% faster (average 23% faster) for all operations except open and random writes which are 8% and 27% slower. PXFS is faster for open as it benefits from accessing metadata directly without invoking the kernel. However, PXFS provides flexibility of interface implementations, while RamFS provides only the POSIX interface. In contrast, existing solutions supporting flexible implementations such as FUSE [48], are more than 10x slower than kernel file systems.

We separately measure the cost of changing file permissions. The TFS asks the SCM manager to change memory protection on pages storing the file with the `scm_mprotect_extent`. If a page has been referenced and is in a page table, the SCM manager shoots down the page from the TLB and invalidates its page table entries. Changing protection takes 3.3μs per page that has been referenced, most of which is TLB shootdown time. For large files, it may be faster to flush the entire TLB.

7.2.2 Application Workload Performance

We evaluate application-level performance with three FileBench profiles (*Fileserver*, *Webserver*, and *Webproxy*) to exercise different aspects of the file system. The Fileserver workload emulates file-server activity and performs sequences of creates, deletes, appends, reads, and writes. The Webserver workload performs sequences of open/read-/close operations on multiple files and appends to a log file. Webproxy performs sequences of create/write/close, open/read/close, and delete operations on multiple files in a single directory plus appends to a log file. Each workload is broken up into individual iterations, and we report the latency of an iteration. The Fileserver and Webserver workloads use 10,000 files, mean directory width of 20, and a 1MB I/O size. The mean file size was 128KB for the Fileserver and 16KB for the Webserver. The Webproxy benchmark was run with 1000 files, mean directory width of 1500, mean file size of 16KB, and 1MB I/O size.

Table 2 shows the average latency to complete one workload operation. The table includes results for both PXFS with name caching and PXFS with no name caching (PXFS-NNC). As we further discuss in Section 7.3.1, name caching improves PXFS performance considerably. Compared to RamFS, PXFS with name caching is only 18% (on average) slower for Fileserver and Webproxy, but matches

the performance of Webserver *despite communication with the TFS*. The microbenchmark results in Table 1 explains much of these results. The Fileserver workload has a large fraction of file creates, deletes, and appends, which are metadata-update intensive and are therefore faster on RamFS because of no crash consistency. Additionally, the Fileserver workload uses larger writes (128KB) than the microbenchmarks (4KB), which amortize the cost of entering the kernel and lead to 20% better performance than PXFS. Webproxy on PXFS is slower than on RamFS for similar reasons. Metadata-update operations are also abundant in the Webproxy workload but their cost on PXFS is offset by the performance benefit of direct access to small files (16KB). Webserver is a read-mostly workload (opens/read-/close) to small files. PXFS performs slightly better (6% better) than RamFS because of the lower latency to open a file (8% lower) due to direct access to metadata.

Compared to ext3 and ext4, which in contrast to RamFS provide crash consistency, PXFS is between 10% and 91% faster than ext3 and between 10% and 29% faster than ext4. Specifically, PXFS improves latency by 79% and 40% for for the Fileserver and Webproxy workloads respectively running on ext3. Both workloads have a large fraction of file creates and deletes, writes, and random access, for which PXFS is substantially faster than ext3. For Fileserver, the large performance improvement comes from PXFS's 102% better performance for writes and 191% faster deletes. The Webserver workload sees less benefit on PXFS as it is almost entirely sequential data reads, which perform similarly on PXFS and ext3, and offset the performance benefit of PXFS's faster file opens (22% faster).

Compared to ext4, PXFS achieves 28% lower latency for the Webproxy workload, 12% lower latency for the Fileserver workload, and 10% lower latency for the Webserver workload. The benefit is lower for Fileserver as ext4's extent file layout improves file write performance by 55% over ext3, which helps bridge the performance gap between PXFS and ext3. Since PXFS uses a radix tree of blocks, which is similar to ext3's tree of indirect blocks, we expect that an extent file layout could similarly improve performance of PXFS.

A large benefit for PXFS comes from batching, which is not possible in ext3 or ext4 because the kernel releases locks before returning to usermode. We found the average optimum batch size for our workloads to be 8MB of metadata. Despite batching, the latency results in Table 2, show that latency is consistently low with 95-percentile latency only slightly higher than average latency.

7.2.3 Client and Server Scalability

The preceding results evaluated single-threaded performance. First, we evaluate the effect of having multiple threads in the client. Figure 5 shows throughput (workload operations per second) for our three workloads as we vary the number of threads in a single client process. For Fileserver, PXFS achieves better scalability than ext3 and nearly doubles throughput when going from 1 to 4 threads. However, we find that PXFS does not scale beyond 4 threads due to contention in the storage allocator. . For Webserver, PXFS achieves almost linear scalability similarly to the kernel-mode file systems. For Webproxy, PXFS throughput does not increase because of single-lock bottlenecks. Specifically, we see contention (22% of total runtime) for the lock covering the single directory. With extra effort, both locks can be split into multiple fine-grained locks to remove this contention.

Benchmark	Throughput (ops/s)			
	1	2	4	6
Fileserver (FS)	59,440	90,594	170,560	213,941
FS+Webproxy (WP)	N/A	273,414	412,647	599,447
FS+WP (FlatFS)	N/A	349,011	660,549	921,816

Table 3: Throughput performance of a multiprogrammed workload with increasing client processes.

We evaluate the scalability of the TFS by running two multiprogrammed workloads: (1) multiple single-threaded Fileserver instances, and (2) Fileserver+Webproxy both using PXFS, and (3) Fileserver using PXFS + Webproxy using FlatFS. The last two comprise multiple number of Fileserver and Webproxy instances. We do not consider Webserver; as a read-mostly workload it does not put much pressure on the server. We configured each client to operate in a different directory to avoid contention between clients due to locking.

Table 3 shows the aggregate throughput of both tests and suggests that both workloads can scale well (about 3x speedup for 4 clients). This is because multiple threads in the TFS can perform metadata updates concurrently under different parts of the namespace due to hierarchical locks. The TFS CPU utilization increases from 20% for 1 client to 83% for 4 clients.

7.3 Optimization Benefits

We finally evaluate the key premise of Aerie: a library file-system implementation enables new performance optimizations.

7.3.1 Path-Name Caching

The major optimization in PXFS is a per-client in-memory cache of path names to speed name resolution. This cache is similar to the dentry cache in the kernel, but replicated in each process. This cache benefits workloads with little sharing, as the cache is flushed when multiple processes open the same files.

By comparing the performance of PXFS with name caching and PXFS with no name caching (PXFS-NNC), which is shown in Table 2 and Figure 5, we see that the path-name cache is quite effective for improving PXFS performance. Specifically, name caching helped improve performance by up to 44% for Fileserver, 121% for Webserver, and 190% for Webproxy. The benefit is higher for Webserver

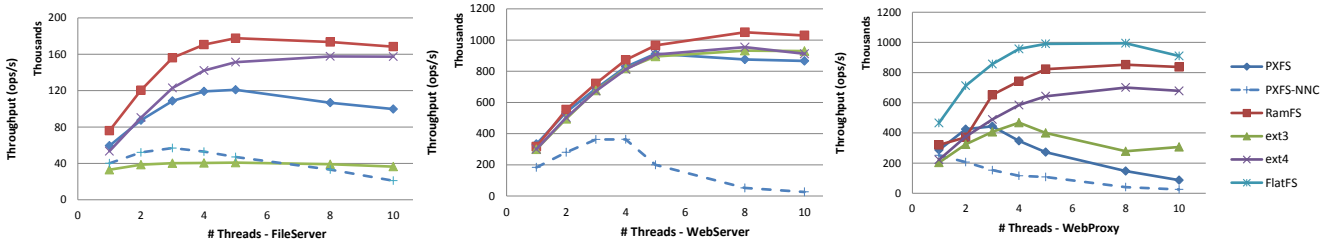

Figure 5: Throughput (operations per second) for 1-10 threads. *PXFS-NNC is PXFS with no name caching.*

and Webproxy as files are small, and therefore name resolution when opening a file is relatively more expensive than it is for larger files in Fileserver.

7.3.2 Workload-Specific Performance

A key motivator for Aerie is the ability to create workload-specific file system interfaces. We measure the performance of FlatFS with a get/put interface in a single directory with the Webproxy workload, whose usage fits the FlatFS interface. We modified the Webproxy workload by converting the create-write-close file sequence to a put operation, open-read-close file to a get operation and delete to an erase operation. We convert the append to a get/modify/put sequence.

The right side of Figure 5 shows the performance of FlatFS for the Webproxy workload. For a single thread, FlatFS is 45% faster than RamFS and 62% faster than PXFS. With five threads, it is 20% faster than RamFS and 263% faster than PXFS. Compared to the kernel-mode file systems that provide crash consistency (*i.e.*, ext3 and ext4), PXFS is between 53% and 109% faster. With a single thread, the biggest benefit comes from using a get/put interface instead of open/read/write/close. With the standard interface, PXFS must create a temporary in-memory object representing an open file and record the file offset on every read. With get/put, FlatFS can locate the file in memory and copy it directly to an application buffer. With multiple threads, the performance benefit comes from using multiple locks within a single directory, which alleviates the scalability limitations of PXFS. In addition, data access is faster with FlatFS because it stores files in a single extent rather than using a radix tree of multiple extents. Thus, getting or putting data is a single `memcpy` operation.

7.4 Sensitivity to Memory Performance

As discussed in Section 2 there are several competing SCM technologies each with different performance characteristics. As the specic design of the memory system can have a great impact on performance [36], and CPU prefetching and caching can hide much of the cost of accessing SCM [14], we perform a series of experiments to study sensitivity to the most important aspect of performance: slow writes.

We emulate slow SCM using DRAM by introducing an extra configurable delay when writing to SCM. For the library file systems, we introduce the delays through the routines that write to SCM both at the client and server. For the kernel file systems, we introduce delays through the RAM disk driver when writing a block. We create delays using a software spin loop that uses the x86 RDTSCP instruction to read the processor timestamp counter and spins until the counter reaches the intended delay.

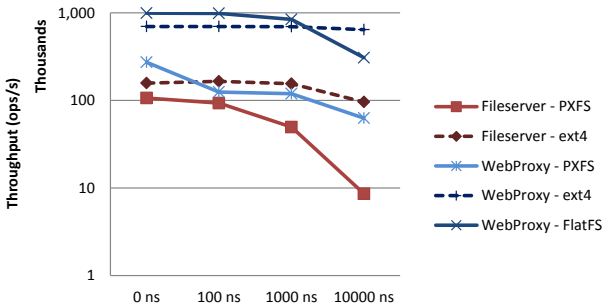

Figure 6: Throughput performance for different memory write latencies. *Horizontal axis shows latency over existing DRAM latency.*

Figure 6 presents throughput performance for different memory write latencies for 8 threads. For brevity, we omit the Webserver workload, which is read-intensive and thus remains largely insensitive to write latency. We make two main observations. First, the performance gap between the library file system PXFS and kernel-mode ext4 grows as write latency to SCM increases, thus suggesting that longer write latencies favor coarse-grain block access over fine-grain byte addressable access. Second, interface specialization through FlatFS is less beneficial with longer write latencies as the storage access cost becomes relatively more significant than the software cost.

8. Related Work

Our work touches and benefits from a wide scope of previous work. Below we discuss and draw connections to classes of previous work we feel are most closely related.

File systems for SCM. Prior to the work discussed in Section 2, earlier projects investigated the integration of storage-class memory into file systems for use as persistent write buffers to reduce the latency of writing data [29], or to hold frequently changing metadata and small files [39, 54]. The fundamental file system and storage architecture are left unchanged.

Application performance and flexibility. Achieving performance improvements by matching application needs to storage-system design has been a recurring theme in the systems community. For example, Google's GFS optimizes for web data [22] and Facebook's Haystack optimizes for images [9]. Exokernel [33] and Nemesis [6] have explored exposing storage to user-mode for application performance and flexibility. However, they still maintain protection of the block device within the kernel, so storage access still requires invoking a kernel-mode device driver. Thus, accessing metadata still requires expensive calls into the kernel.

Distributed file systems. Our architecture has been influenced by distributed file system designs and naturally bears many similarities to them. Coda [34], Farsite [4] and Ivy [40] distribute file system functionality to untrusted clients, and reintegrate clients' changes to the file system by verifying and replaying operations previously written to a log. Previous work on distributed file systems provided direct access to block storage over the network to improve performance and scalability [15, 23, 37, 50]. Aerie applies similar techniques to a local setting and must work with the fixed memory interface rather than a flexible software interface to storage. Similar to FlatFS, many distributed file systems relax file system consistency semantics for improved performance [30, 44].

Fast networking. Previous work on user-mode networking [53] had recognized the need for direct protected access to fast network devices to avoid software-layering overheads. On-going work has proposed unifying the I/O stack for fast networking and fast SCM [51].

9. Conclusion

New storage technologies promise high-speed access to storage directly from user mode. We have presented a decentralized file-system architecture that represents a new design targeting SCM, and reduces the kernel role to just multiplexing physical memory. Applications can achieve high performance by optimizing the file-system interface for application needs without changes to complex kernel code. We showed that we can still implement POSIX with good performance and that further performance improvements can come from customizing the interface to the workload.

Acknowledgments

We thank David DeWitt for his support in pursuing this work. We thank Andrea Arpaci-Dusseau, Kim Keeton, Harumi Kuno, and Brad Morrey for their feedback on earlier drafts of this paper. We also thank our anonymous reviewers, and our shepherd, Steven Hand, who helped us improve this paper through reviews and suggestions. This work was supported in part by a grant from the Microsoft Jim Gray Systems Lab, and in part by the National Science Foundation under grants CNS-0915363, CNS-1218485, and CNS-0834473.

References

[1] Fans of the OS Plan 9 from Bell Labs: unlink/remove/whatever. `http://blog.gmane.org/gmane.os.plan9.general/month=20020801/page=14`, Aug. 2002.

[2] Filebench benchmark. `http://sourceforge.net/apps/mediawiki/filebench`, 2011.

[3] PMFS: A file system for persistent memory. `https://github.com/linux-pmfs/pmfs/blob/master/Documentation/filesystems/pmfs.txt`, 2013.

[4] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *OSDI 5*, Dec. 2002.

[5] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. *Trans. Storage*, 3(3), Oct. 2007.

[6] P. R. Barham. A fresh approach to file system quality of service. In *NOSSDAV*, May 1997.

[7] A. Basu, J. Gandhi, M. M. Swift, M. D. Hill, and J. Chang. Efficient virtual memory for big memory servers. In *ISCA 40*, June 2013.

[8] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP 22*, Oct. 2009.

[9] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *OSDI 9*, Oct. 2010.

[10] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *ASPLOS 17*, Mar. 2012.

[11] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(4):271–307, Nov. 1994.

[12] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *SOSP 24*, Nov. 2013.

[13] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS 16*, Mar. 2011.

[14] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP 22*, Oct. 2009.

[15] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The direct access file system. In *FAST 2*, Mar. 2003.

[16] Dovecot. Mailbox formats. `http://wiki.dovecot.org/MailboxFormat/`.

[17] U. Drepper. Futexes are tricky. `www.akkadia.org/drepper/futex.pdf`, 2005.

[18] L. A. Eisner, T. Mollov, and S. Swanson. Quill: Exploiting fast non-volatile memory by transparently bypassing the file system. Technical report, UCSD, 2013.

[19] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of ACM*, 19(11):624–633, Nov. 1976.

[20] R. F. Freitas and W. W. Wilcke. Storage-class memory: the next storage system technology. *IBM Journal of Research and Development*, 52(4):439–447, 2008.

[21] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown. Recon: Verifying file system consistency at runtime. In *FAST 10*, Feb. 2012.

[22] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP 19*, Oct. 2003.

[23] G. A. Gibson, D. Rochberg, J. Zelenka, D. F. . Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. G. off, C. Lee, B. Ozceri, and E. Riedel. File server scaling with network-attached secure disks. In *SIGMETRICS 1997*, June 1997.

[24] E. Giles, K. Doshi, and P. Varman. Bridging the programming gap between persistent and volatile memory using WrAP. In *CF'13*, May 2013.

[25] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP 12*, Dec. 1989.

[26] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Readings in database systems*, pages 94–121. Morgan Kaufmann Publishers Inc., 1988.

[27] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *OSDI 10*, Oct. 2012.

[28] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX ATC Winter*, Dec. 1994.

[29] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. Technical Report TR 3002, NetApp, 2005.

[30] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[31] Y. Huai. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS Bulletin*, 18(6):33–40, Dec. 2008.

[32] A. M. Joshi. Adaptive locking strategies in a multi-node data sharing environment. In *VLDB 17*, Sept. 1991.

[33] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *SOSP 16*, Oct. 1997.

[34] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10:3–25, Feb. 1992.

[35] E. J. Koldinger, J. S. Chase, and S. J. Eggers. Architecture support for single address space operating systems. In *ASPLOS 5*, Oct. 1992.

[36] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase-change memory as a scalable DRAM alternative. In *ISCA 36*, June 2007.

[37] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *ASPLOS 7*, Oct. 1996.

[38] D. Mazières. A toolkit for user-level file systems. In *USENIX ATC*, June 2001.

[39] E. Miller, S. Brandt, and D. Long. HeRMES: High-performance reliable MRAM-enabled storage. In *HotOS 8*, May 2001.

[40] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: a read/write peer-to-peer file system. In *OSDI 5*, Dec. 2002.

[41] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21:23–36, Feb. 1988.

[42] M. K. Qureshi, J. K. Michele, Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-Based. main memory with start-gap wear leveling. In *MICRO 42*, Dec. 2009.

[43] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *ASPLOS 2*, Oct. 1987.

[44] R. Sandberg. The Sun network file system: Design, implementation and experience. In *USENIX ATC*, June 1986.

[45] M. D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. PhD thesis, Massachusetts Institute of Technology, 1972.

[46] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *OSDI 9*, Oct. 2010.

[47] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453:80–83, 2008.

[48] M. Szeredi. Fuse: Filesystem in userspace. `http://fuse.sourceforge.net`, 2005.

[49] V. Technology. ArxCis-NV (TM): Non-Volatile DIMM. `http://www.vikingtechnology.com/arxcis-nv`, 2014.

[50] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *SOSP 16*, Oct. 1997.

[51] A. Trivedi, P. Stuedi, B. Metzler, R. Pletka, B. G. Fitch, and T. R. Gross. Unified high-performance I/O: one stack to rule them all. In *HotOS 14*, May 2013.

[52] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *ASPLOS 16*, Mar. 2011.

[53] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *SOSP 15*, Dec. 1995.

[54] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *USENIX ATC*, June 2002.

[55] X. Wu and A. L. N. Reddy. SCMFS: a file system for storage class memory. In *SC'11*, Nov. 2011.