Aerie: Distributing File System Functionality for Direct Access to Storage-Class Memory

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 read and write data without kernel interaction but maintain the sharing and protection features of file systems. 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 for their specific needs. An application may access storage through an interface or data layout suited to its workload. As a concrete example, we demonstrate a specialized file system that reduces a hierarchical file system abstraction to a key/value store with fewer consistency guarantees but lower access latency. We find that a POSIX-style file system on Aerie performs 35% better than ext3 over SCM and only 15% worse than a kernel file system without crash consistency. A specialized design enabled by Aerie improves performance by 30-86% over a kernel file system for its workload.

1. Introduction

Emerging device technologies such as phase-change-memory (PCM), spin-torque transfer RAM (STT-RAM) and memristors provide persistent storage near the speed of DRAM. These technologies collectively are termed *storage-class memory* (SCM) [22] as data can be accessed through ordinary load/store instructions rather than through I/O requests. As a result, user-mode code can access data directly, so there is no need for the OS to mediate every access. Furthermore, existing virtual-memory hardware can protect access to individual data pages.

These two features of storage-class memory, high speed and direct access from user mode, are a fundamental shift from existing storage architectures. In most operating systems, the kernel has to mediate access to storage for protection and to abstract details of the specific storage device through a driver. The existing OS structure of file systems as a kernel-level service may no longer be necessary with SCM.

Recent work has explored high-performance file-system designs targeted for SCM. The BPFS file system leverages SCM's properties to provide strong reliability guarantees and better performance than existing file systems [16]. The Moneta-D system bypasses the kernel for data access but metadata operations critical to small-file performance still require the kernel [13]. Mnemosyne and NV-heaps [15, 57] provide direct access by memory mapping SCM into a process, but do not support full file-system functionality. We seek a system where almost all file system operations are handled at user mode in a client process, including metadata operations. This can benefit workloads that operate on many small files rather than a few large files.

We propose distributing the storage stack to create a flexible, high-performance storage architecture by mapping SCM into client processes. This provides 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 and implementation without extending the kernel. With direct access, a program reading a file can locate the file contents and read the data directly, without calling the kernel. In addition, an application with fixed-size files can pre-allocate storage in contiguous extents to lower the cost of locating file contents. Prior efforts at user-mode file systems, such as FUSE [1], provide flexibility at a heavy performance cost from context switching and copying data.

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 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: a POSIX-style system and one optimized for small-file access through a put/get interface. Through experiments, we show that the POSIX-style system performs much better than existing user-mode file systems and between only 15% slower than a kernel file system without consistency guarantees and 35% faster on average than ext3. The specialized key-value file-system interface performs up to 86% faster than the fast kernel file system. Thus, distributing file system functionality to client processes allows flexible implementations that dramatically improve performance.

2. Motivation

Despite rapid advancements in storage technology, the fundamental architecture of storage in operating systems has remained stable: applications invoke the kernel to store and retrieve data, which invokes a file system and then a block driver. Microkernels and user-mode file systems move the file system logic to user-mode, but leave the remaining layers intact [41, 1]. Databases may bypass the file system, but still call through the kernel and block driver.

2.1 What Changed?

Four features of past storage technologies require this layered design in the kernel:

1. **Protection:** Disks and other block storage devices do not implement a protection mechanism to limit access by a user or process. In addition, they use DMA to read/write data, which does not respect memory protection. Thus, the OS relies on the file system to decide which processes have access to which blocks on disk.
2. **Scheduling:** Disks have variable latency from seek and rotational delays and benefit significantly from scheduling to reorder requests.
3. **Caching:** Slow disks benefit from shared caches that allow processes to re-use data fetched by another process.

4. **Drivers:** Disks implement a variety of interfaces and therefore require a driver to present a standard block interface.

Due to the slow speed of disks and the high benefits of scheduling and caching, a kernel implementation of file systems provide many performance benefits at little additional cost.

Storage-class memory 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 [22]. Three recent technologies provide SCM capabilities: phase-change memory (PCM) [38], spin-torque-transfer RAM (STT-RAM) [31], and memristors [54]. 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. While SCMs are currently only available in small sizes and scaling projections have flattened in recent years, recent work indicates that even current SCM devices offer great potential for improved storage performance [12].

SCM has none of the features that require a kernel implementation of file systems. As memory, it 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 provide speeds near DRAM, caching data may be unnecessary. Finally, SCM does not require a driver for data access as it can implement a standard load/store or protected DMA interface [13].

2.2 What Benefits?

With these features in mind, we seek to redesign how operating systems support file systems. We seek two benefits from this redesign: flexibility and performance.

Flexibility. Major performance improvements can come from matching application needs to storage-system design. For example, Google's GFS optimizes for web data [24] and Facebook's Haystack optimizes for images [8]. These systems provide better performance by specializing for a narrow range of application workloads (*e.g.*, whole file access).

Implementing an application-specific file system today entails three unattractive choices (i) modify the kernel, which is difficult and requires long-term maintenance as the kernel evolves; (ii) layer above an existing file system, as in Microsoft Office file formats that implement a file system within a file [28] and add additional latency to file access; or (iii) use a user-mode framework such as FUSE [1], which decreases performance through extra context switches to a file-system process.

Direct access to storage from user level allows an application to implement a file system customized to its needs without modifying the kernel and without any performance penalty from executing outside the kernel. There are several degrees of freedom for an application to optimize a file system. The interface of a file system can be optimized based on the application needs, such as get/put for a mail message store rather than open/read/write/close. The interface can also provide concurrency semantics that better target an application's concurrency model, such as byte-range locking to enable concurrent access to the same file. Moreover, an application may optimize the file system's caching scheme and data structures to the access patterns of the workload. Also, an application can choose a layout suited to its workload, such as contiguous allocation for fixed-size files.

Performance. Direct access to file data from user-level can be lower latency and higher bandwidth than going through the kernel, as it avoids the cost of changing modes and cache pollution from entering the kernel [16, 53]. For example, PCM read latencies can be as low as 70 nanoseconds, while the time for a `stat` system call in Linux on a modern processor is approximately 800 nanoseconds. The Moneta-D system provides some of this benefit by bypassing the kernel for data access [13], but metadata operations critical to small-file performance, such as for IMAP servers that use many small files [20], still call into the kernel.

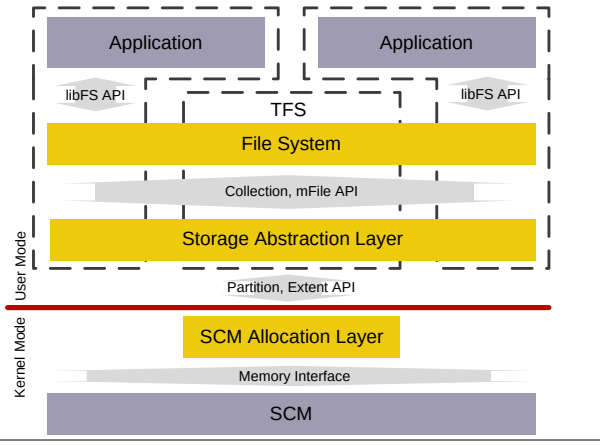

Figure 1. Aerie design. Functionality is distributed between application processes, a trusted service, and the kernel.

3. Design

Aerie is a local storage service that enables programs to store and share data through a user-mode file system. Our goals for Aerie are:

1. *Flexibility* for applications to customize the file system interface, policies, and layout.
2. *High performance* to reap the benefits of low-latency storage memory.
3. *Protected sharing* between processes to enable flexible application structuring and composition.

The main enabling mechanism for our goals is direct access through memory to file-system data and metadata from user mode. Although direct access enables our first two goals, it is at odds with our third goal of protected sharing and raises several other challenges. Protected sharing requires mediating each file system access to enforce file system permissions. For performance, we use hardware memory protection when possible, which poses the challenge of translating file system permissions to memory protection.

Designing a file system for flexibility and direct access raises many challenges, of which we focus on four. First, how should file systems be structured to provide maximum flexibility to applications while still providing access to shared data? Existing file systems have a single implementation per system, but direct access enables each program to have a different implementation. Second, if clients have direct access to data, how can we ensure a malicious or buggy application does not corrupt file system structures? Third, how do we provide concurrency control between applications efficiently? Kernel file systems use locks in shared memory and provide only limited locking capabilities to user-mode code. Finally, how can we minimize communication for coordination between clients to keep performance high?

3.1 Assumptions

We designed Aerie based on assumptions about the features of future SCM products and application behavior. First, we assume that hardware provides user-mode code with low-latency, protected access to SCM. This can be achieved by placing SCM directly on the memory bus, which allows access through normal load/store instructions protected by virtual memory hardware [16, 15, 44]. Alternatively, protected user-mode DMA that restricts the set of

memory and SCM addresses can also be used [13].

Second, our failure model assumes that any state stored in volatile memory such as the hardware cache or main memory may be lost but any state written to SCM persists. We therefore depend on hardware to provide an atomic update of at least 64 bits [16, 46]. Similar to past systems [15, 57], we also rely on the 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 [47]. Without such hardware support, the system must be limited to clients that are trusted not to inflict excessive wear.

Third, we assume that most sharing is sequential rather than concurrent. Programs with concurrent access may be better served by a centralized file system. When concurrent sharing occurs, Aerie provides correct behavior but with reduced performance.

Finally, we assume programs may exhibit malicious or buggy behavior such as stray writes. We therefore do not trust the client library to correctly modify file-system metadata. However, we assume that trusted code in the kernel or service is as reliable as existing file system code (which can also corrupt data in memory [62]).

3.2 Architecture

Aerie distributes file system functionality for direct and protected access to storage memory. Figure 1 shows the Aerie organization. The SCM manager in the kernel allocates SCM to file systems and constructs page tables that map SCM into processes with the necessary protection. Programs link to the libFS library that provides the file system API. The trusted file system (TFS) service provides integrity for metadata updates and concurrency control between processes.

Aerie relies on hardware protection to enforce access control over file system data. This allows the client library to service most file-system operations directly from SCM without contacting the service or the kernel. 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.

Thus, Aerie provides two paths to storage: (1) a fast, untrusted code path, and (2) a slow, trusted code path. The untrusted code path executes within any user process and relies on virtual memory hardware to limit access to data. Thus, reading files can be provided completely by the code in libFS. The trusted code path invokes TFS to support any operation on file-system state that cannot be enforced in hardware. For example, enforcing integrity constraints, such as reference counts, cannot be left to clients.

In order to support multiple file systems, a system may have multiple implementations of the library and the service, one for each file system implementation. The kernel code, though, is common to all file systems. In addition, a single file system may have multiple libFS implementations optimized for different workloads.

3.2.1 Abstractions and Interfaces

To encourage flexibility, we architect Aerie in three layers, each providing a specific service. Like ZFS layers (ZPL, DMU, and SPA) [10], these layers allow lower-level services to be reused by multiple higher-level file systems or file-system interfaces. Table 1 lists the layers of Aerie and the main operations of the layer.

Storage-class-memory allocation layer. At the bottom level, the SCM allocation layer does the minimum work required for protection: it records and enforces resource usage. To achieve this, it exposes storage in the form of *memory partitions* and *memory extents*, which are close to hardware and provide primitive operations upon which higher-level interfaces are built.

A memory partition is a contiguous region of virtual addresses mapped to SCM, and is used for coarse-grain allocation of physical SCM to a file system volume. Partitions are named by their starting address, and the allocation layer exposes a list of partitions and their owners similar to a mount table.

A memory extent is a range of memory within a partition associated with protection rights and higher-level software assigns their location and size. Memory extents are similar to standard file-system extents with the addition of protection and are used by higher layers to store data within an object, such as file contents. While each extent is contiguous virtually, higher-level objects constructed out of extents, such as files, may not be as they can combine non-adjacent extents through an indirection structure. Extents are named by a descriptor that encodes their partition and offset within the partition. In addition to methods to create and delete extents and partitions, the SCM allocation layer provides a method to change the protection of an extent with the `mprotect` operation.

The SCM allocation layer provides a simple ACL representation for higher levels to specify permission on file-system data. These ACLs are stored for each extent.

Storage abstraction layer. This layer adds structure and synchronization to the raw memory exposed by the allocation layer. It provides low-level methods that can be used to implement functionality commonly found in file systems, thus allowing higher-level flexibility.

The storage abstraction layer implements two basic abstractions of memory: the *memory file* (*mFile*), which maps offsets to extents, and the *collection*, which groups objects (mFiles or other collections). Both mFiles and collections provide the same access to all members (extents for mFiles and elements for a collection). These abstractions are the building blocks for files, directories, and other file-system metadata structures.

The storage abstraction layer also provides *distributed concurrency control* through a hierarchical lock service to synchronize access to shared data across processes. The lock service lets clients request a lock for shared or exclusive access to an object. It also offers *hierarchical intent locks* [27, 60], which can make locking more efficient when resources can be organized in a hierarchy such as a directory of files. We describe this more in later sections.

File-system interface layer. The top layer implements a file system API. Similar to Pilot [49], this level provides a name space above lower-level collections and mFiles and can expose familiar interfaces to create, delete, or rename files. While the lower level provide the mechanisms for storing, retrieving, and protecting data, the interface layer provides the policies of how and when to access data. The FS interface layer assigns protection both to storage objects that clients can read directly and to metadata limited to the TFS. The interface can either be a standard POSIX interface or tailored to application needs, such as a tag-based lookup mechanism [52].

3.2.2 Components

While Aerie seeks to provide as much functionality within a client process as possible, some file-system features require a third party. Cooperation between mutually distrustful programs require a trusted entity to enforce synchronization and integrity [51]. Thus, Aerie distributes the file system among three components.

libFS Client Library. Applications link against a libFS library for each file-system interface they use. The library provides functionality from both the FS interface and storage-abstraction layers needed 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. It also implements logic to invoke a trusted service.

Trusted File System (TFS) Service. Functionality that requires a trusted third party (*i.e.*, integrity and concurrency control) but *not* privileged hardware access execute in the TFS service, which

Abstraction	Function	Operations
SCM Allocation		
Partitions	Contiguous virtual memory holding a FS volume	`allocate, free, mount`
Extent	Contiguous memory with same access rights	`create, delete, mprotect`
Storage Abstraction		
mFiles	Mapping of offsets to memory locations	`create, delete, addExtent, removeExtent, protect`
Collections	Grouping of storage objects	`create, delete, add, remove, protect`
Locks	Concurrency control	`request, release, revoke`
File System		
Files, Directories	Standard POSIX file systems	`create, unlink, read, write, ...`

Table 1. Aerie layers and their supported abstractions and operations.

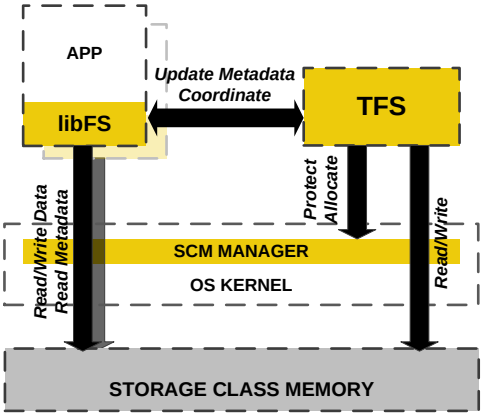

Figure 2. Aerie components (solid colored). Arrows show communication between components.

includes code both from the storage abstraction and FS interface layers. The service also provides complete file system functionality on data for which memory protection is too coarse. The service executes as a user-mode process accessed via RPCs.

SCM Manager. Operations requiring hardware privileges, such as modifying memory permissions or virtual address mappings, must execute in the kernel. The SCM manager provides the storage allocation layer, which contains only low-level mechanisms that are independent of high-level file system organization.

3.3 File System Design

In previous sections we described the layered architecture and basic components of Aerie. Here, we discuss how to build a complete file system with this architecture.

Naming. Aerie implements a namespace by mapping each directory to a collection. The collection maps a name to an identifier of an mFile (for a file) or a collection (for a subdirectory). While namespace operations are metadata heavy, operations that read metadata can be serviced directly and safely by the library to avoid communication with the TFS service on every file access. Thus, directory operations such as traverse, read, and open are handled by locating collections and reading them directly.

Protection. We use hardware page-level protection to implement file-system permissions. We precompute the permissions for each piece of file data and then construct a page table for a process based on the permissions it receives. In contrast, a normal file system checks an ACL on every open, whereas we compute the effective ACL for the user and put those permissions in the page table.

Although hardware-enforced protection enables us to efficiently enforce permissions, it may be too rigid to directly represent the whole spectrum of file-system permissions. For example, while read permission on a file is equivalent to read permission on a page containing the file data, this does not hold true for directory contents: *list* and *traverse* both require reading the directory contents, but only *list* lets the client see all the names. Thus, clients with traverse only should not be able to access the page containing directory contents.

A file-system implementation can therefore choose which memory to protect with hardware and which to protect in software. Data with simple read/write permissions corresponding to memory protection can be made available to libFS through memory. If not, then libFS must call into the TFS service to access the object.

Integrity. In order to prevent corruptions by untrusted code, only the TFS service applies modifications to file system objects. Clients are only trusted to modify file contents but not file system objects, such as mFiles and collections. Instead, they create a log of their operations, similar to a file system journal, and ship the journal to the TFS. The TFS then validates the entries in the journal are legal, protected by locks, and preserve invariants, such as link counts and free/allocated status. Finally, the TFS applies them as a transaction by forcing the log to SCM and then updating data structures.

Similarly to a kernel-mode file system though, the service is vulnerable to corrupting itself. The file system and storage abstractions can use existing techniques such as checksums, replication, sealing pages through memory protection, or periodic checks of metadata.

Concurrency. Aerie performs concurrency control over storage objects using the distributed hierarchical lock service. File systems use hierarchical locks to grant a client access to a subtree of the name space. A client can read all the files within a subtree without communicating with the server, which improves performance. 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 covering an object when sending a metadata update to the TFS. Clients, though, can treat locks as advisory, because the data is already available through memory. In addition, a client may change the granularity of locks internally, such as by assigning byte-range locks to different threads while the process holds a lock on the entire file.

The file system library is free to provide applications any degree of consistency and concurrency semantics. For example, Windows file systems provide mandatory locking, while POSIX file systems do not, and GFS provides atomic append operations [24]. A client library can choose when to acquire locks and how long to hold them to implement different consistency levels [27].

Clients can batch updates together while holding a lock. This amortizes the cost of communicating with the server. Internally, a client should structure operations as short-lived because a lock may be revoked on short notice. When a lock is revoked, the client must ship to the service any buffered metadata updates covered by the lock before releasing the lock. This is required because TFS does not accept metadata updates that are not covered by a lock in order to preserve invariants during concurrent updates. If a client does not revoke the lock when requested, TFS forcefully releases the lock, which clients treat similarly to media removal.

Durability. The TFS guarantees atomicity and durability of metadata operations, but the libFS clients must ensure durability of file

Subsystem	# Lines	Subsystem	# Lines
Infrastructure services	3800	Storage objects	10220
SCM manager	650	PXFS	2660
Distributed Lock Service	2910	KVFS	340
Total	20580		

Table 2. Implementation size.

data by forcing it out to SCM. To provide atomicity when an operation modifies multiple metadata objects, the TFS provides atomicity using write-ahead redo logging. For example, writing new data to a file may require the file system to allocate and insert multiple extents into the file's storage object, all of which go to a persistent journal first.

Failures. Aerie must handle four types of failure. First, media failure may happen when SCM hardware is physically damaged. Such failures can be addressed using existing techniques such as checksums and replication. Second, client failures may happen when a client crashes due to a software error. Aerie addresses client failures by revoking locks held by the client and discarding any outstanding metadata updates a failed client has not yet shipped to the service. This guarantees metadata invariants but allows client data to be lost. Third, the whole system may fail due to a software or hardware error, or power failure. Upon system restart, the service restarts and can perform recovery from the log. Third, the service may fail similarly to clients. We treat this similar to a system failure, and require that the TFS and all clients restart.

3.4 Summary

Aerie provides flexibility through simple abstractions that support a wide variety of file system interfaces. It is structured in three layers, which are distributed through the kernel, client libraries, and a trusted file-system service. It prevents a faulty client from corrupting data by centralizing metadata updates at the TFS, which ensures clients can only corrupt file contents (similar to current systems). Aerie uses a distributed hierarchical lock service to synchronize access to files, which allows many operations to be performed without communication and many updates to be batched together.

An alternative design to direct access to SCM is to cache an entire file system in DRAM and use SCM only to log updates. We see two challenges to this approach: (1) recovery may be slow, as it requires rereading data from another medium (flash or disk) and reapplying a long log, and (2) SCM device scalability may be better than DRAM and lower power [38].

The Aerie design shares strong similarities with many distributed file systems that allow clients to access a storage service directly. However, it relies on hardware for access, which only provides read and write operation, rather than software, which can provide a much richer interface [43, 25]. The trust model also differs from past systems relying on a shared block device, because clients cannot be trusted to make correct changes to metadata [55].

4. Implementation

We implemented an Aerie prototype on Linux 3.2.2 for x86-64 processors. In this section we present the implementation of the two lower-level layers of Aerie, the SCM allocation layer and storage abstraction layer. We defer the discussion of two file system interfaces to Section 5. Table 2 summarizes the amount of code comprising Aerie's major components. Kernel-mode code is written in C, while user-mode code is written in C++.

4.1 Infrastructure Services

Aerie relies on low-level mechanisms for inter-process communication and for consistently updating data in SCM.

Interprocess Communication. We use remote-procedure call (RPC) implemented using sockets on 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 4.4) 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 [7].

Persistence Primitives. We borrow the persistence primitives from Mnemosyne [57] 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 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.

4.2 SCM Manager

The SCM manager is a kernel component that provides the storage-class memory allocation layer. Its sole 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. The SCM manager does *not* allocate extents and only records their location and protection.

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, and maps SCM at the same virtual address in all processes. Thus, mounting a partition does not actually map it into an address space but instead ensures that ensuing page faults will lazily create the page table. This effectively treats the page table as a giant software TLB, similar to Mach's pmap structure [48]. 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.

Protection. The unit of protection in Aerie is the extent. We store 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 higher 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 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.

We borrow a technique from single address space operating systems to handle page faults [14]. 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 a processor architecture with support for separating protection from addressing [37], only a single page table would be needed.

4.3 Distributed Lock Service

The storage abstraction layer contains two services: distributed concurrency control and storage objects. We discuss concurrency first and storage objects in the next section.

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 [55, 32, 60, 26, 27, 40]. However, because our lock service is intended for a single machine, we do not replicate the service for fault tolerance. Aerie does use Linux's futexes [21] because it must be able to revoke locks.

Clients access the lock service via a local clerk. 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.

Hierarchical locking. Aerie assigns a unique global lock to every object, such as a collection or mFile. Like a futex, the ID of an object can be used as the name of a lock [21]. The lock manager provides 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. When acquiring a hierarchical lock clients can access members of a collection without additional locks.

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.

Protection. 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 [26]. A client that does not renew its lease implicitly releases the lock and allows other processes to proceed. Furthermore, Aerie can limit the number of locks a process may hold to reduce contention [11].

4.4 Storage Objects

The storage abstraction layer provides objects upon which file systems can build. The implementation is shared between the TFS server, which is responsible for writing to objects, and libFS, which provides read access to objects and write access to object contents.

4.4.1 Data Structures

We provide simple implementations of mFile and collections that is sufficient to show the use of the abstractions. We use C++ template polymorphism to decouple the implementation of the interface from the implementation of layout and consistent updates. Thus, other implementations are possible with the same interface and could provide better performance or less space overhead.

Each storage object is identified by a 64-bit integer (a *storage object ID*). 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. This encoding enforces a minimum object size of 64-bytes and provides 64 different types. As a result, accessing an object 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 a structure composed of multiple extents. Storage objects expose a buffer to store metadata from the file system interface layer. They are allocated from extents with other objects sharing the same protection.

Collections. 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. The hash table stores key-value pairs in which a key is an array of bytes of arbitrary length and the value field stores a 64-bit storage object ID. 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 drop below 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.

mFile. The mFile object provides access to a range of bytes starting at a specified offset. We implement the mFiles as radix tree of indirect blocks that point to fixed-size extents. Larger extents are broken into pieces when added to the tree.

4.4.2 Protection

The file-system layer assigns protection to a storage object and the storage layer must propagate that protection to the memory containing the file objects using the `scm_mprotect_extent` API.

However, memory and file systems do not have perfectly compatible protection models: 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 search. The file system interface maps each granted permission to the protection it enables and each denied permission to the protection it disables. The granted permissions remaining after removing denied permissions are then mapped to memory. This process ensures that permissions requiring conflicting protection

are properly enforced. For example, granting write-only access to a file allows the write permission, which is enabled by read/write protection, but disallows the read permission, which prevents read-only and read/write protection. Thus, the file data would be set to no-access protection.

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

4.4.3 Concurrency

The storage layer associates each storage object with a global lock. Clients acquire the lock in read-mode when they read objects directly, and in write mode for metadata updates performed by the TFS. The service verifies that clients hold the appropriate lock on the object before it performs any updates.

File systems use hierarchical locks by organizing storage objects in a tree hierarchy through collections. When a client acquires a hierarchical lock on a collection, it implicitly locks any other storage object accessible through the collection. When using an implicit lock to perform an update, it must prove to the TFS that its lock covers the object by providing the list of collections between the locked and mutated object. In addition, the client must obtain intent locks above the locks it holds to detect conflicts.

When a storage 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 [27]. 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.

4.4.4 Integrity

File systems interfaces and the storage layer cooperate to guarantee the integrity of metadata. The file system interface enforces high-level invariants, such as ensuring that rename operations do not cause cycles in the namespace. The storage layer enforces storage-object invariants, such as ensuring that mFiles map only allocated extents.

The TFS server performs all modifications to file system objects in order to prevent clients from violating these invariants. To avoid frequent synchronous calls to the service for every metadata update, 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.

The log is implemented by the storage abstraction layer and contains operations from both storage objects and the file system interface. Each entry has a header identifying the operation, the identifiers of the objects it modifies, and fields the operation updates. For example, the entry for creating a file has the directory object, the object ID of the file, and the name of the file. The ability to log operations from both storage objects and the file system enable the TFS server to benefit from work done at the client. For example, when the client logs a file write that requires allocating new storage, it logs each individual storage extent it pre-allocates in addition to logging the high-level file operation. 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 then attach the extents to the file.

The server validates a client's updates before applying them. First, it validates that messages have a valid structure, correspond to known operations, and that the operation maintains invariants. Second, it verifies that the client holds necessary locks and permissions. Finally, it performs the operations using redo logging to survive crashes. Recent work has shown that file-system integrity constraints can be enforced using fast local checks on the data being modified [23].

4.4.5 Crash Recovery

The TFS server 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 server 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.

4.4.6 Free Space Management

The TFS service implements a buddy storage allocator [35, 36] 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. The service maintains a collection that tracks the pre-allocated objects owned by each client to prevent memory leaks.

5. 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 implemented two file system interfaces. The first, PXFS, shows how to use the storage abstractions to implement a POSIX-style file system interface for compatibility with existing code. The second one, KVFS, shows how to optimize the interface for a specific workload.

5.1 PXFS: POSIX-style 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.

Storage Objects. We implement POSIX storage objects directly with mFiles and collections. Files are mFiles with page-size extents and directories are a collection mapping file names to the object IDs of files and directories. A root collection holds the root directory. Because the storage layer provides most of the mechanisms to access file data, the file and directory implementation is small. 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 server.

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 read/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 read/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. Since acquiring multiple locks, may lead to deadlock, we acquire all the locks in advance and release all locks if the TFS revokes a lock. Since acquiring read/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 service maintains a table of open files that are not locked. If 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 into 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. This design guarantees the client can directly access the file even if other clients unlink or rename it.

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 [45], 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.

Discussion. With this design, read-only access to files only communicates with the TFS service to acquire locks, and if there are no conflicting accesses, a coarse grained lock high in the file system tree suffices. The client can write to file data locally, including writing new data to files, but must communicate with the service for metadata changes such as creating or appending to a file.

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 [50] and AFS [30], relax consistency semantics.

5.2 KVFS: Key-Value File System

In order to demonstrate how an application can use Aerie's facilities to improve performance, we designed KVFS to provide a (i) 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, KVFS does not support POSIX semantics, such as hierarchical namespace, unlinking or renaming open files, and multiple names for a file.

KVFS files are implemented with mFiles containing a single extent holding the entire file contents. The mFiles store no other metadata, such as permissions or access time. The file system does not have a hierarchical namespace, so all files are stored in a single collection that maps file names to mFiles. Thus, KVFS and PXFS use the *same memory layout* and differ in the policies the interface layer uses to allocate and synchronize data.

We enable scalable concurrent access to the 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. 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 read/write lock while lookups acquire a read lock. When insert or delete cause a rehash of the table, the rehash operation acquires the single lock covering the whole collection in read/write mode.

With this design, a client can operate almost entirely without communication. It can batch requests to pre-create file objects and allocate extents, and then commit groups of files at once. Furthermore, 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. An alternative model to KVFS would be to implement a key-value store as a single large file, KVFS, in contrast, enables mutually distrustful programs to concurrently access and update files, such as for indexing or backup/restore.

6. Evaluation

The goal of Aerie is flexibility and performance. We evaluate performance with a mix of micro- and macrobenchmarks and compare against traditional and user-mode file systems. In addition, we evaluate the benefits of specializing file system design to a workload.

While we have limited experience building file systems for Aerie, the PXFS and KVFS file systems demonstrate the value of its layered architecture. Table 2 gives the size of each file system. For comparison, the ext3 file system in the Linux kernel is 11,663 lines, and the user-mode implementation for Fuse is 18,916 lines. KVFS, with reduced functionality, is only 340 additional lines of code yet provides synchronized access to files. While neither KVFS nor PXFS are as full-featured as ext3, their small size demonstrates the benefit of Aerie's storage abstractions to flexible file-system design.

6.1 Methodology

We performed our experiments on a at 2.4GHz Intel Xeon E5645 six-core (twelve thread) machine equipped with 48GB of DRAM running x86-64 Linux 3.2.2 kernel.

Storage class memory. It is not yet possible to purchase any form of storage class memory that can be plugged into a commodity server and accessed from user mode. Instead, we emulate SCM using DRAM by adding delays to model SCM's performance, which has been used by past projects [56, 61, 57].

As phase-change memory (PCM) is the nearest to commercial availability, we base our model on its performance. There is a wide variety of projections for PCM's performance, and the specific design of the memory system can have a great impact on performance [38]. We limit our model to the most important aspect of performance: slow writes. Our model accounts for PCM's slower writes relative to DRAM by introducing a delay after each write. All writes to SCM in Aerie use a macro that allows us to insert a delay. Our model does not account for additional latency on loads or the effects of memory-system architecture, including cache evictions and read-after-write bank conflicts, where reads may be queued behind long writes to the same bank.

We estimated write bandwidth based on projections provided by Numonyx [19]. All tests add 150ns of extra latency as demonstrated previously by PCM prototype devices [9, 4] and limit write bandwidth to 4GB/s. For all our experiments we report averages of at least five runs.

Workloads. We compare Aerie against three Linux file systems: RamFS, ext3 and ext3 with FUSE. RamFS uses the VFS page cache and dentry cache as an in-memory file system. We modified RamFS

Benchmark	Latency (μs)			
	RamFS	ext3	ext3-FUSE	PXFS
Sequential read	0.77	0.83	3.5	0.7
Sequential write	2.1	1.9	18.4	1.5
Random read	1.2	4.5	25.5	1.2
Random write	1.4	3.8	20.8	1.6
Open	2.0	4.7	127.0	3.7
Create	9.2	113.4	2044.5	13.4
Delete	3.1	11.6	227	2.7
Append	5.4	7.5	25.3	4.0

Table 3. Latency of common file system operations. All read/write operations use a 4096-byte buffer.

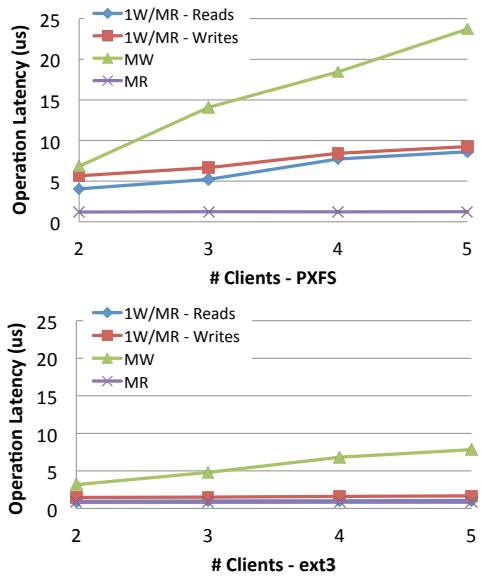

Figure 3. Performance of concurrent clients reading or writing a 4KB page of the same file. 1W - one writer, MW - multiple writers, and MR - multiple readers.

to introduce delays when writing data and metadata in the caches to account for the extra delay of PCM. 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 constructed an emulator, *SCM-disk*, for a PCM-based block device. Based on Linux's RAM disk (`brd` device driver), SCM-disk introduces delays when writing a block to limit the effective bandwidth. We mount an ext3 file system on SCM-disk. In addition, we also configured a user-mode version of ext3 using FUSE [1] that writes data to SCM-disk. We use 16GB memory partition for all four configurations.

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.

6.2 Microbenchmark Performance

Individual operations. A prime motivation for Aerie is that direct access to storage can make user-mode file systems as fast as ones in the kernel. We evaluate the latency of common 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.

Table 3 shows the latency of common file system operations on PXFS, RamFS, and both ext3 versions. As expected RamFS performs consistently better than ext3 except for sequential write. Writes in RamFS are performed directly to SCM whereas in ext3 they are staged in RAM. PXFS performs close to RamFS for all operations but `create` and `open`, where PXFS latency is 45% and 85% higher respectively. Opening a file takes longer for PXFS because pathname resolution walks the persistent directory structure for each path component, and creates a shadow object to buffer metadata updates. Of the 3.7μs to complete an open call, 0.85μs is spent in lookup and 1.5μs in creating a shadow. In constrast, RamFS already has the objects in memory so it only pays the overhead of looking up directory entries in the dentry cache, which is highly optimized for lookups. Adding a path-name cache to PXFS or deferring shadow object creation until the first write can reclaim much of the performance difference.

Benchmark	Latency (μs)			
	RamFS	ext3	ext3-FUSE	PXFS
Fileserver	198 (218)	424 (764)	5324	270 (292)
Webserver	143 (132)	158 (148)	2728	189 (195)
Webproxy	90 (66)	140 (130)	234	86 (64)

Table 4. Average latency and 95-percentile latency (in parentheses) to complete one workload iteration.

Compared with ext3 in the kernel, PXFS is between 15% to 90% faster (average 35%) for all operations. Open is faster for PXFS because ext3 has to bring the file into the inode cache. PXFS benefits by not calling into the kernel, which helps all writes and random reads, and by batching metadata updates for create, delete and append.

In comparison to Aerie's performance, user-mode performance with FUSE was between 5 to 21 times slower for read/write operations and between 10 to 150 times slower for metadata operations, largely due to the extra context switches and data copies necessary to invoke the file-system service. Aerie provides a flexible means of implementing file systems with high performance.

We separately measure the cost of changing file permissions. The TFS server 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 shootdowns time. For large files, it may be faster to flush the entire TLB.

Sharing. Due to the lack of workloads that exhibit sharing between applications, we measure instead the impact of sharing on our system through a stress microbenchmark. Figure 3 shows the latency of reading or writing a 4KB page of the same file shared between multiple concurrent clients in PXFS. As expected, multiple writers exhibit poor performance due to heavy lock contention, but a single writer with multiple readers degrades read and write performance only slightly. Running the same microbenchmark on ext3 shows similar behaviour but with better absolute performance than PXFS due to in-kernel shared memory locking rather than distributed locking.

6.3 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 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 File-

Benchmark	Throughput (workload iterations/s)			
	1	2	4	6
Fileserver	3974	8532	12131	18190
Fileserver+Webproxy	N/A	6217	11784	16200

Table 5. Throughput performance of a multiprogrammed workload with increasing client processes.

server and Webserver benchmark 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 4 shows the average latency to complete one workload iteration. Compared to RamFS, PXFS is 35% slower for Fileserver and Webserver, but matches the performance of Webproxy. The microbenchmark results in Table 3 explains much of these results. Additionally, the Fileserver workload uses larger writes (128KB) than the microbenchmarks (4KB), which ammortize the cost of entering the kernel and lead to 25% better performance than PXFS.

Webserver is a read-mostly workload (opens/read/close) to small files. RamFS performs better because of the lower latency to open a file (60% lower) due to in-memory caching of directory entries. In contrast, PXFS has no caching and must re-walk persistent structures on every access. PXFS matches the performance of RamFS on Webproxy because there is only a single directory lookup.

Compared to ext3, which in constrast to RamFS provides crash consistency, PXFS achieves 36% and 39% lower latency for the Fileserver and Webproxy workloads. 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 better write performance for writes (103μs vs 220μs) and 70% faster deletes (19μs vs 62μs). The Webserver workload, though, is almost entirely sequential data reads and performs worse on PXFS for the same reasons that PXFS performs worse than RamFS.

A large benefit for PXFS comes from batching, which is not possible in ext3 because the kernel releases locks before returning to usermode. We found the average optimum batch size for our workloads to be 8MB. As shown in Table 4, batching for PXFS affects latency variance only slightly, with 95-percentile latency only slightly higher than average latency. The exception is Webproxy where the 95-percentile is lower due to a single outlier pulling the average up. Note that the other file systems show similar outlier effects, even without batching.

6.4 Client and Server Scalability

The preceding results evaluated single-threaded performance. First we evaluate the effect of having multiple threads in the client. Figure 4 shows throughput (workload iterations per second) for our three workloaads as we vary the number of threads in a single client process. For Fileserver, PXFS achieves better scalability than ext3 and doubles throughput when going from 1 to 6 threads. For the other two workloads, Webserver and Webproxy, PXFS throughput does not increase because of single-lock bottlenecks. (1) For Webserver, we see increased contention (20% of total runtime) for an internal lock in the storage object implementation. (2) For Webproxy, 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.

RamFS presents near-linear scalability for all three workloads primarily because it does not have to write to a journal, which becomes a scalability bottleneck for ext3 (5% for 1 thread vs. 30% for 4 threads of total run time is spent in journaling). RamFS also benefits from scalable RCU synchronization for directory lookups.

We evaluate the scalabilty of the TFS server by running two multiprogrammed workloads: (1) multiple single-threaded Fileserver instances, and (2) Fileserver+Webproxy, which runs an equal 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 5 shows the aggregrate throughput of both tests and suggests that both workloads can scale well (3x speedup for 4 clients). This is because multiple threads in the TFS server can perform metadata updates concurrently under different parts of the namespace due to hierarchical locks. The server CPU utilization increases from 24% for 1 client to 72% for 4 clients.

6.5 Workload-Specific Performance

A key motivator for Aerie is the ability to create workload-specific file system interfaces. We compare the performance of KVFS with a get/put interface in a single directory against PXFS for the Webproxy workload, whose usage fits the KVFS 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.

Figure 4 shows the performance of KVFS for the Webproxy workload. For a single thread, KVFS is 86% faster tham RamFS and 79% faster than PXFS. With six threads, it is 30% faster tham RamFS and 415% faster than PXFS. 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, KVFS 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 KVFS 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. 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. There have been several prior projects investigating the integration of storage-class memory into file systems. Initially cast as non-volatile RAM (NVRAM), these memories can be used as persistent write buffers to reduce the latency of writing data [29], or to hold frequently changing metadata and small files [42, 59]. However, the fundamental file system and storage architecture are left unchanged. More recently, the BPFS file system leverages SCM's properties to provide strong reliability guarantees and better performance over traditional file systems [16] while SCMFS utilizes the existing memory management module in the operating system to remove block management from the file system [61]. However, both designs provide no direct access to storage from user mode and no flexibility in file system interface and organization. The Moneta-D system makes a first step towards direct user-mode access by bypassing the kernel for data access. However, metadata operations critical to small-file access still require the kernel, and direct access is not exploited for file system flexibility.

Distributed file systems. Our architecture has been influenced by distributed file system designs and naturally bears many similarities

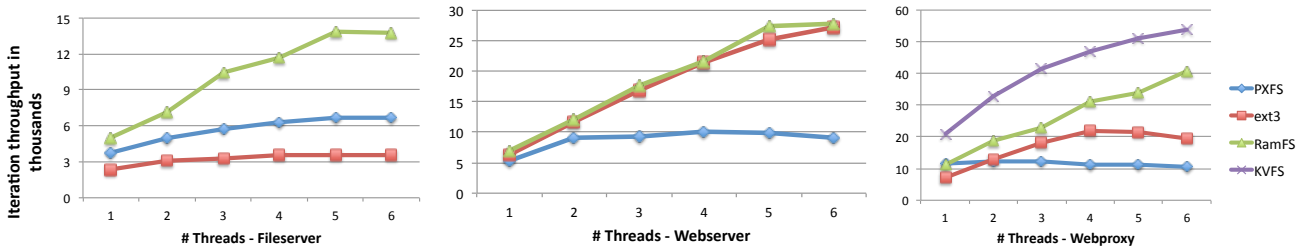

Figure 4. Throughtput performance (workload iterations per second) as a function of the number of threads in a single client.

to them. Coda [34], Farsite [3] and Ivy [43] 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 [5, 18, 25, 55, 39]. 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 KVFS, many distributed file systems relax file system consistency semantics for improved performance [30, 50].

File system structure. Several projects have explored building file systems on primitives higher than the block abstraction. ZFS [10], object-based storage devices (OSD) [25] and the logical disk [17] all expose groups of blocks and atomic operations. While ZFS collects block storage under a transactional object store, OSDs push the object functionality into the storage device itself instead. Boxwood [40] explored fault-tolerant data structures such as fault-tolerant B-tree as the fundamental storage infrastructure for building distributed file or other storage systems. Aerie similarly exposes primitive structures, mFiles and collections, to the file system interface layer.

User-mode access. Exokernel [33] and Nemesis [6] have explored exposing storage to user-mode for application performance and flexibility. However, they still maintained protection of the block device within the kernel, so disk access still required invoking a kernel-mode device driver. Other attempts to move the file-system code to user mode have retained the model of a shared file system that accesses storage through a shared device, such as a network or disk [41, 1]. Modern DBMS completely bypass the file system and perform direct I/O to the block device, which however prevents sharing the block device with other untrusted applications. Finally, previous work on user-mode networking [58] had recognized the need for direct protected access to fast network devices to avoid software-layering overheads.

8. Conclusion

New storage technologies promise high-speed access to storage directly from user mode. The existing file-system architecture, where a shared kernel component mediates all access to data, unnecessarily limits performance both by interposing on requests and complicating file system implementation. The Aerie architecture represents a new design targeting storage-class memory, and reduces the kernel role to just multiplexing physical memory. As a result, applications can achieve high performance by optimizing the file system interface for application needs without changes to complex kernel code.

References

[1] Fuse: File system in userspace. `http://fuse.sourceforge.net`.

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

[3] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *OSDI 5* (Dec. 2002).

[4] AHN, S., SONG, Y., JEONG, C., SHIN, J., FAI, Y., HWANG, Y., LEE, S., RYOO, K., LEE, S., PARK, J., HORII, H., HA, Y., YI, J., KUH, B., KOH, G., JEONG, G., JEONG, H., KIM, K., AND RYU, B. Highly manufacturable high density phase change memory of 64mb and beyond. In *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International* (dec. 2004), pp. 907 – 910.

[5] ANDERSON, D., CHASE, J., AND VAHDAT, A. Interposed request routing for scalable network storage. In *OSDI 4* (Oct. 2000).

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

[7] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP 22* (Oct. 2009).

[8] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., AND VAJGEL, P. Finding a needle in Haystack: Facebook's photo storage. In *OSDI 9* (Oct. 2010).

[9] BEDESCHI, F., RESTA, C., KHOURI, O., BUDA, E., COSTA, L., FERRARO, M., PELLIZZER, F., OTTOGALLI, F., PIROVANO, A., TOSI, M., BEZ, R., GASTALDI, R., AND CASAGRANDE, G. An 8Mb demonstrator for high-density 1.8V phase-change memories. In *VLSI Circuits, Symposium on* (June 2004).

[10] BONWICK, J., AHRENS, M., HENSON, V., MAYBEE, M., AND SHELLENBAUM, M. The zettabyte file system. Technical report, Sun Microsystems, 2002.

[11] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *OSDI 7* (Nov. 2006).

[12] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *MICRO 43* (Dec. 2010).

[13] CAULFIELD, A. M., MOLLOV, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing safe, user space access to fast, solid state disks. In *ASPLOS 17* (Mar. 2012).

[14] CHASE, J. S., LEVY, H. M., FEELEY, M. J., AND LAZOWSKA, E. D. Sharing and protection in a single-address-space operating system. *ACM TOCS 12*, 4 (Nov. 1994), 271–307.

[15] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ASPLOS 16* (Mar. 2011).

[16] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *SOSP 22* (Oct. 2009).

[17] DE JONGE, W., KAASHOEK, M. F., AND HSIEH, W. C. The logical disk: a new approach to improving file systems. In *SOSP 14* (Dec.

1993).

[18] DEBERGALIS, M., CORBETT, P., KLEIMAN, S., LENT, A., NOVECK, D., TALPEY, T., AND WITTLE, M. The direct access file system. In *FAST 2* (Mar. 2003).

[19] DOLLER, E. Phase change memory and its impacts on memory hierarchy. `http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf`, 2009.

[20] DOVECOT. Mailbox formats. `http://wiki.dovecot.org/MailboxFormat/`.

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

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

[23] FRYER, D., SUN, K., MAHMOOD, R., CHENG, T., BENJAMIN, S., GOEL, A., AND BROWN, A. D. Recon: Verifying file system consistency at runtime. In *FAST 10* (Feb. 2012).

[24] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *SOSP 19* (Oct. 2003).

[25] GIBSON, G. A., ROCHBERG, D., ZELENKA, J., NAGLE, D. F. ., AMIRI, K., CHANG, F. W., FEINBERG, E. M., OFF, H. G., LEE, C., OZCERI, B., AND RIEDEL, E. File server scaling with network-attached secure disks. In *SIGMETRICS 1997* (June 1997), pp. 272–284.

[26] GRAY, C., AND CHERITON, D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP 12* (Dec. 1989).

[27] GRAY, J. N., LORIE, R. A., PUTZOLU, G. R., AND TRAIGER, I. L. Granularity of locks and degrees of consistency in a shared data base. In *Readings in database systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988, pp. 94–121.

[28] HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *SOSP 23* (Oct. 2011), pp. 71–83.

[29] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an nfs file server appliance. Tech. Rep. TR 3002, NetApp, 2005.

[30] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51–81.

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

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

[33] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICEÑO, H. M., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on exokernel systems. In *SOSP 16* (Oct. 1997).

[34] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10* (Feb. 1992), 3–25.

[35] KNOWLTON, K. C. A fast storage allocator. *Communications of ACM 8*, 10 (Oct. 1965), 623–624.

[36] KNUTH, D. The art of computer programming volume 1: Fundamental algorithms. Addison-Wesley, Reading, MA.

[37] KOLDINGER, E. J., CHASE, J. S., AND EGGERS, S. J. Architecture support for single address space operating systems. In *ASPLOS 5* (Oct. 1992).

[38] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase-change memory as a scalable DRAM alternative. In *ISCA 36* (June 2007).

[39] LEE, E. K., AND THEKKATH, C. A. Petal: distributed virtual disks. In *ASPLOS 7* (Oct. 1996).

[40] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI 6* (Dec. 2004).

[41] MAZIÈRES, D. A toolkit for user-level file systems. In *USENIX ATC* (June 2001).

[42] MILLER, E., BRANDT, S., AND LONG, D. HeRMES: High-performance reliable MRAM-enabled storage. In *HotOS 8* (May 2001).

[43] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: a read/write peer-to-peer file system. In *OSDI 5* (Dec. 2002).

[44] NARAYANAN, D., AND HODSON, O. Whole-system persistence. In *ASPLOS 17* (Mar. 2012).

[45] OUSTERHOUT, J. K., CHERENSON, A. R., DOUGLIS, F., NELSON, M. N., AND WELCH, B. B. The Sprite network operating system. *IEEE Computer 21* (Feb. 1988), 23–36.

[46] OUYANG, X., NELLANS, D. W., WIPFEL, R., FLYNN, D., AND PANDA, D. K. Beyond block I/O: Rethinking traditional storage primitives. In *HPCA 17* (Feb. 2011), pp. 301–311.

[47] QURESHI, M. K., MICHELE, J. K., FRANCESCHINI, SRINIVASAN, V., LASTRAS, L., AND ABALI, B. Enhancing lifetime and security of PCM-Based. main memory with start-gap wear leveling. In *MICRO 42* (Dec. 2009).

[48] RASHID, R., TEVANIAN, A., YOUNG, M., GOLUB, D., BARON, R., BLACK, D., BOLOSKY, W., AND CHEW, J. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *ASPLOS 2* (Oct. 1987).

[49] REDELL, D. D., DALAL, Y. K., HORSLEY, T. R., LAUER, H. C., LYNCH, W. C., MCJONES, P. R., MURRAY, H. G., AND PURCELL, S. C. Pilot: an operating system for a personal computer. *Communications of ACM 23* (Feb. 1980), 81–92.

[50] SANDBERG, R. The Sun network file system: Design, implementation and experience. In *Proceedings of the Summer USENIX Conference* (June 1986).

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

[52] SELTZER, M., AND MURPHY, N. Hierarchical file systems are dead. In *HotOS 12* (May 2009).

[53] SOARES, L., AND STUMM, M. FlexSC: Flexible system call scheduling with exception-less system calls. In *OSDI 9* (Oct. 2010).

[54] STRUKOV, D. B., SNIDER, G. S., STEWART, D. R., AND WILLIAMS, R. S. The missing memristor found. *Nature 453* (2008), 80–83.

[55] THEKKATH, C. A., MANN, T., AND LEE, E. K. Frangipani: A scalable distributed file system. In *SOSP 16* (Oct. 1997).

[56] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST 9* (Feb. 2011).

[57] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: lightweight persistent memory. In *ASPLOS 16* (Mar. 2011).

[58] VON EICKEN, T., BASU, A., BUCH, V., AND VOGELS, W. U-Net: a user-level network interface for parallel and distributed computing. In *SOSP 15* (Dec. 1995).

[59] WANG, A.-I. A., REIHER, P., POPEK, G. J., AND KUENNING, G. H. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *USENIX ATC* (June 2002).

[60] WILLIAM E. SNAMAN, J., AND THIEL, D. W. The VAX/VMS Distributed Lock Manager. *Digital Technical Journal 1*, 5 (Sept. 1987), 29–44.

[61] WU, X., AND REDDY, A. L. N. SCMFS: a file system for storage class memory. In *SC2011* (Nov. 2011).

[62] ZHANG, Y., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. End-to-end data integrity for file systems: a zfs case study. In *FAST 8* (Feb. 2010).