Scaling a file system to many cores using an operation log

Srivatsa S. Bhat,[†] Rasha Eqbal,[‡] Austin T. Clements,[§]
M. Frans Kaashoek, Nickolai Zeldovich
MIT CSAIL

ABSTRACT

It is challenging to simultaneously achieve multicore scalability and high disk throughput in a file system. For example, even for commutative operations like creating different files in the same directory, current file systems introduce cache-line conflicts when updating an in-memory copy of the on-disk directory block, which limits scalability.

ScaleFS is a novel file system design that decouples the in-memory file system from the on-disk file system using per-core operation logs. This design facilitates the use of highly concurrent data structures for the in-memory representation, which allows commutative operations to proceed without cache conflicts and hence scale perfectly. ScaleFS logs operations in a per-core log so that it can delay propagating updates to the disk representation (and the cache-line conflicts involved in doing so) until an `fsync`. The `fsync` call merges the per-core logs and applies the operations to disk. ScaleFS uses several techniques to perform the merge correctly while achieving good performance: timestamped linearization points to order updates without introducing cache-line conflicts, absorption of logged operations, and dependency tracking across operations.

Experiments with a prototype of ScaleFS show that its implementation has no cache conflicts for 99% of test cases of commutative operations generated by Commuter, scales well on an 80-core machine, and provides on-disk performance that is comparable to that of Linux ext4.

1 INTRODUCTION

Many of today's file systems do not scale well on multicore machines, and much effort is spent on improving them to allow file-system-intensive applications to scale better [4, 10, 13, 23, 26, 31]. This paper contributes a clean-slate file system design that allows for good multicore scalability by separating the in-memory file system from the on-disk file system, and describes a prototype file system, ScaleFS, that implements this design.

The main goal achieved by ScaleFS is **multicore scalability**. ScaleFS scales well for a number of workloads on an 80-core machine, but even more importantly, the ScaleFS implementation is conflict-free for almost all commutative operations [10]. Conflict freedom allows ScaleFS to take advantage of disjoint-access parallelism [1, 20] and suggests that ScaleFS will continue to scale even for workloads or machines we have not yet measured.

In addition to scalability, ScaleFS must also satisfy two standard file system constraints: **crash safety** (meaning that ScaleFS recovers from a crash at any point and provides clear guarantees for `fsync`) and **good disk throughput** (meaning that the amount of data written to the disk is commensurate with the changes that an application made to the file system, and that data is written efficiently).

These goals are difficult to achieve together. Consider directory operations in Linux's ext4 file system [29]. Directories are represented in memory using a concurrent hash table, but when Linux updates a directory, it also propagates these changes to the in-memory ext4 physical log, which is later flushed to disk. The physical log is essential to ensure crash safety, but can cause two commutative directory operations to contend for the same disk block in the in-memory log. For example, consider `create(f1)` and `create(f2)` under the same parent directory, where `f1` and `f2` are distinct. According to the Scalable Commutativity Rule [10], because these two `creates` commute, a conflict-free and thus scalable implementation is possible. However, in Linux, they may update the same disk block, causing cache conflicts and limiting scalability despite commuting.

Multicore scalability is important even for a file system on a relatively slow disk,[1] because many workloads operate in memory without flushing every change to disk. For example, an application may process a large amount of data by creating and deleting temporary files, and flush changes to

[†] Now at VMware. [‡] Now at Apple. [§] Now at Google.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SOSP'17, October 28–31, 2017, Shanghai, China.
© 2017 Copyright is held by the owner/author(s).
ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132779

[1] In this paper, we use the term "disk" loosely to refer to persistent storage, including rotational disks, flash-based SSDs, etc.

disk only after producing a final output file. It is important that the application not be bottlenecked by the file system when it is not flushing data to disk. Similarly, a file system may be hosting multiple applications, such as a text editor (which requires strong durability) and a parallel software build (which does not require immediate durability but needs the file system to scale well).

Our key insight is to *decouple* the in-memory file system from the on-disk file system, and incur the cost of writing to disk (including the cache-line conflicts to construct an in-memory copy of the on-disk data structures that will be written to disk) only when requested by the application. To enable decoupling, ScaleFS separates the in-memory file system from the on-disk file system using an operation log, based on oplog [5]. The operation log consists of per-core logs of file system operations (e.g., `link`, `unlink`, `rename`). When `fsync` or `sync` are invoked, ScaleFS sorts the operations in the operation log by timestamp, and applies them to the on-disk file system. For example, ScaleFS implements directories in such a way that if two cores update different entries in a shared directory, then no interaction is necessary between the two cores. When an application calls `fsync` on the directory, ScaleFS merges the per-core operation logs into an ordered log, prepares the on-disk representation, adds the updated disk blocks to a physical log, and finally flushes the physical log to disk.

Although existing file systems decouple representations for reads, the operation log allows ScaleFS to take this approach to its logical extreme even for updates. For example, Linux has an in-memory cache that represents directory entries differently than on disk. However, system calls that modify directories update both the in-memory cache and an in-memory copy of the on-disk representation. Keeping an updated copy of the on-disk representation in memory means that when it is eventually written to disk (e.g., when an application invokes `fsync`), the on-disk state will correctly reflect the order in which the application's system calls executed.

ScaleFS's log enables two independent file systems: an in-memory file system tailored to achieve scalability, and an on-disk file system tailored for high disk throughput. The in-memory file system can choose data structures that allow for good concurrency, choose inode numbers that can be allocated concurrently without coordination, etc. (as in sv6 [10]) and be completely unaware of the on-disk data structures. On the other hand, the on-disk file system can choose data structures that allow for good disk throughput, and can even reuse an existing on-disk format. The operation log connects the two file systems, so that changes that took place in the in-memory file system can be applied consistently to the on-disk file system on `fsync`.

We have implemented ScaleFS by extending sv6 [10], which does not provide crash safety, with the on-disk file system from xv6 [12] using the decoupling approach. The implementation does not support all of the file system operations that, say, Linux supports (such as `sendfile`), but ScaleFS does implement many operations required from a file system, and supports complex operations such as `rename` across directories. Furthermore, we believe that extending ScaleFS to support additional features can be done without impacting scalability of commutative operations.

Experiments with ScaleFS on Commuter [10] demonstrate that ScaleFS maintains sv6's high multicore scalability for commutative operations while providing crash safety. This demonstrates that ScaleFS's decoupling approach is effective at combining crash safety and scalability. Experimental results also indicate that ScaleFS achieves better scalability than the Linux ext4 file system for in-memory workloads, while providing similar performance when accessing disk. ScaleFS is conflict-free in 99.2% of the commutative test cases Commuter generates, while Linux is conflict-free for only 65% of them. Furthermore, experiments demonstrate that ScaleFS achieves good disk performance.

The main contributions of the paper are as follows.

- A new design approach for multicore file systems that decouples the in-memory file system from the on-disk file system using an operation log.
- Techniques based on timestamping linearization points that ensure crash safety and high disk performance.
- An implementation of the above design and techniques in a ScaleFS prototype.
- An evaluation of our ScaleFS prototype that confirms that ScaleFS achieves good scalability and performance.

The rest of the paper is organized as follows. §2 describes related work, §3 describes the semantics that ScaleFS aims to provide, §4 provides an overview of ScaleFS, §5 describes the design of ScaleFS, §6 summarizes ScaleFS's implementation, §7 presents experimental results, and §8 concludes.

2 RELATED WORK

The main contribution of ScaleFS is the split design that allows the in-memory file system to be designed for multicore scalability and the on-disk file system for durability and disk performance. The rest of this section relates ScaleFS's separation to previous designs.

File system scalability. ScaleFS adopts its in-memory file system from sv6 [10]. sv6 uses sophisticated parallel-programming techniques to make commutative file system operations conflict-free so that they scale well on today's multicore processors. Due to these techniques, sv6 scales better than Linux for many in-memory operations. sv6's, however, is only an in-memory file system; it does not write data to durable storage. ScaleFS's primary contribution over sv6's in-memory file system lies in combining multicore scalability with durability. To do so, ScaleFS extends the in-memory file system using oplog [5] to track linearization points, and adds an on-disk file system using an operation log sorted by the timestamps of linearization points.

NOVA [42] and iJournaling [32] take an approach similar to ScaleFS by maintaining per-inode logs. ScaleFS generalizes the per-inode log approach by allowing the use of different on-disk file systems, whereas NOVA and iJournaling dictate the on-disk layout by requiring the per-inode log to be directly stored in non-volatile memory or on disk. ScaleFS also allows directory operations to be more scalable than NOVA or iJournaling, because ScaleFS maintains per-core logs, which avoid cache-line contention even when modifying the same directory from different cores. Finally, ScaleFS can absorb multiple operations to the same directory to reduce the amount of data written to disk.

Many existing file systems, including Linux ext4, suffer from multicore scalability bottlenecks [4, 10, 31], and file system developers are actively improving scalability in practice. However, most practical file systems are taking an incremental approach to improving scalability, as demonstrated by NetApp's Waffinity design [13]. This improves scalability for particular workloads and hardware configurations, but fails to achieve ScaleFS's goal of conflict-free implementations for all commutative operations, which is needed to scale on as-of-yet unknown workloads or hardware platforms.

Separation using logging. File systems use different data structures for in-memory and on-disk operations. For example, directories are often represented in memory differently than on disk to allow for higher performance and parallelism. Linux's dcache [11, 30], which uses a concurrent hash table, is a good example. Similarly, ext4's delayed allocation allows ext4 to defer accessing disk bitmaps until necessary. However, no file system completely decouples the in-memory file system from the on-disk file system. In particular, in every other scalable file system, operations that perform modifications to in-memory directories also manipulate a copy of the on-disk data structures in memory. File systems update an in-memory copy of the on-disk data structures so that when these data structures are eventually written to disk, they are consistent with the order in which the applications' system calls executed in memory. This lack of decoupling between the in-memory and on-disk representations causes unnecessary cache-line movement and limits scalability.

ReconFS [28] pushes the separation further than traditional file systems do. For example, it decouples the volatile and the persistent directory tree maintenance and emulates hierarchical namespace access on the volatile copy. In the event of system failures, the persistent tree can be reconstructed using embedded connectivity and metadata persistence logging. ReconFS, however, is specialized to non-volatile memory, while ScaleFS's design does not require non-volatile memory.

The decoupling is more commonly used in distributed systems. For example, the BFT library separates the BFT protocol from NFS operations using a log [6]. However, these designs do not use logs designed for multicore scalability.

ScaleFS implements the log that separates the in-memory file system from the on-disk file system using an oplog [5]. Oplog allows cores to append to per-core logs without any interactions with other cores. ScaleFS extends oplog's design to sort operations by timestamps of linearization points of file system operations to ensure crash safety.

Applying distributed techniques to multi-core file systems. Hare is a scalable in-memory file system for multicore machines without cache coherence [17]. It does not provide persistence and poses this as an open research problem. ScaleFS's decoupling approach is likely to be a good match for Hare too.

SpanFS [22] extends Hare's approach by implementing persistence. This requires each core to participate in a two-phase commit protocol for operations such as `rename` that can span multiple cores. Much like Hare, SpanFS shards files and directories across cores at a coarse granularity, and cannot re-balance the assignment at runtime. While SpanFS can alleviate some file system scalability bottlenecks, its rigid sharding requires the application developer to carefully distribute files and directories across cores, and to not access the same file or directory from multiple cores.

In contrast to SpanFS and Hare, ScaleFS does not require the application developer to partition files or directories. ScaleFS provides scalability for commutative operations, even if they happen to modify the same file or directory. As we show in §7, ScaleFS achieves good scalability even when delivering messages to shared mailboxes on 80 cores; we do not expect SpanFS or Hare would scale under this workload.

On-disk file systems. ScaleFS's on-disk file system uses a simple design based on xv6 [12]. It runs file system updates inside of a transaction as many previous file systems have done [8, 18], has a physical log for crash recovery of metadata file system operations (but less sophisticated than, say, ext4's design [29]), and implements file system data structures on disk in the same style as the early versions of Unix [37].

Because of the complete decoupling of the in-memory file system and on-disk file system, ScaleFS could be modified to use ext4's disk format, and adopt many of its techniques to support better disk performance. Similarly, ScaleFS's on-disk file system could be changed to use other ordering techniques than transactions; for example, it could use soft updates [16], a patch-based approach [15], or backpointer-based consistency [7].

ScaleFS's on-disk file system provides concurrent I/O using standard striping techniques. It could be further extended using more sophisticated techniques from file systems such as LinuxLFS [21], BTRFS [38], TABLEFS [36], NoFS [7], and XFS [40] to increase disk performance.

ScaleFS's on-disk file system can commit multiple transactions concurrently using multiple physical journals. Prior work [27, 32, 35] has also shown how concurrent `fsync` calls can run in parallel. Using CCFS [35] or iJournaling [32] instead of our prototype on-disk file system may provide better disk performance; the contribution of this paper lies in the

decoupling of the in-memory and on-disk file systems, and the multi-core scalability that this allows ScaleFS to achieve.

ScaleFS's on-disk file system is agnostic about the medium that stores the file system, but in principle should be able to benefit from recent work using non-volatile memory, such as UBJ [25], byte-granularity logging [14, 42], or ReconFS [28] to minimize intensive metadata writeback and scattered small updates.

3 DURABILITY SEMANTICS

To achieve high performance, a file system must defer writing to persistent storage for as long as possible, allowing two crucial optimizations: batching (flushing many changes together is more efficient) and absorption (later changes may supersede earlier ones). However, a file system cannot defer forever, and may need to flush to persistent storage for two reasons. First, it may run out of space in memory. Second, an application may explicitly call `fsync`, `sync`, etc. Thus, for performance, it is critical to write as little as possible in the second case (`fsync`). To do this, it is important to agree on the semantics of `fsync`.

A complication in agreeing on the semantics of `fsync` comes from the application developer, whose job it is to build crash-safe applications on top of the file system interface. If the interface is too complex to use (e.g., the semantics of `fsync` are subtle), some application developers are likely to make mistakes [34]. On the other hand, an interface that provides cleaner crash safety properties may give up performance that sophisticated application programmers want. In this paper, ScaleFS aims to provide high performance, targeting programmers who carefully manage the calls to `sync` and `fsync` in their applications, with a view to obtain the best performance. This approach is in line with other recent work on high-performance file systems [32, 35]. Additional interfaces can help application developers write crash-safe applications, such as CCFS streams [35], Featherstitch [15], and failure-atomic msync [33], although the ScaleFS prototype does not support them.

ScaleFS's semantics for `fsync` are defined by four properties. The first property is that *`fsync`'s effects are local*: an `fsync` of a file or directory ensures that all changes to that file or directory, at the time `fsync` was called, are flushed to disk. For instance, calling `fsync` on one file will ensure that this file's data and metadata are durably stored on disk, but will not flush the data or metadata of other files, or even of the directory containing this file. If the application wants to ensure the directory entry is durably stored on disk, it must invoke `fsync` on the parent directory as well [34, 43].

The local property allows for high performance, but some operations—specifically, `rename`—cannot be entirely local. For instance, suppose there are two directories `d1` and `d2`, and a file `d1/a`, and all of them are durably stored on disk (there are no outstanding changes). What should happen if the application calls `rename(d1/a, d2/a)` followed by `fsync(d1)`? Naïvely following the local semantics, a file system might flush the new version of `d1` (without `a`), but avoid flushing `d2` (since the application did not call `fsync(d2)`). However, if the system were to now crash, the file would be lost, since it is neither in `d1` nor in `d2`. This purely local specification makes it hard for an application to safely use `rename` across directories, as articulated in our second property, as follows.

The second property for ScaleFS's `fsync` semantics is that *the file system can initiate any `fsync` operations on its own.* This is crucial because the file system needs to flush changes to disk in the background when it runs out of memory. However, this property means that if the user types `mv d1/a d2/a`, a file system implementing purely local semantics can now invoke `fsync(d1)` to free up memory, and as in the above example, lose this file after a crash. This behavior would be highly surprising to users and application developers, who do not expect that `rename` can cause a file to be lost, if the file was already persistently stored on disk before the `rename`.

The third property aims to resolve this anomaly, by requiring that *`rename` will not cause a file or directory to be lost*, although it is acceptable if a crash during `rename` causes a file to appear twice in the file system (both under the old and new path names). As mentioned in the previous paragraph, this property is not strictly necessary based on first principles, but we believe a file system would be difficult to use without this guarantee.

The fourth property requires that *on-disk data structures must be crash-safe*, meaning the file system will not be corrupted after it recovers from a crash. An example of what falls under this property is what ext4 provides in `data=ordered` mode, which prevents uninitialized data from being exposed after a crash.

Taking all of ScaleFS's properties together, the final semantics of `fsync` are that it flushes changes to the file or directory being `fsync`ed, and, in the case of `fsync` on a directory, it also flushes changes to other directories where files may have been renamed to, both to avoid losing files and to maintain internal consistency.

4 OVERVIEW

ScaleFS consists of two file systems: an in-memory file system called MemFS, and an on-disk file system called DiskFS. The in-memory file system uses concurrent data structures that allow commutative in-memory operations to scale, while the on-disk file system deals with constructing disk blocks and writing them to disk. This approach allows most commutative file system operations to execute conflict-free. The two file systems are coupled by an operation log [5], which logs directory changes such as `link`, `unlink`, `rename`, etc. Although an operation log is designed for update-heavy workloads, it is a good fit for ScaleFS's design because MemFS handles all of the reads, and the log entries are collected when flushing changes to DiskFS.

To allow MemFS to operate independently of DiskFS, MemFS uses different inode numbers from DiskFS; to make this explicit, we use the term *mnode* to refer to inode-like

objects used by MemFS, and we reserve the term inode for DiskFS. This allows MemFS to allocate mnode numbers in a scalable way (e.g., by incrementing per-core counters), but still preserves the ability of DiskFS to map inode numbers to physical disk locations. As in traditional Unix file systems, inode numbers are permanent; however, the application-visible mnode number (which is returned by the `stat` system call) can change after a reboot. To the best of our knowledge, the only application that may rely on stable inode numbers across reboots is qmail [2].

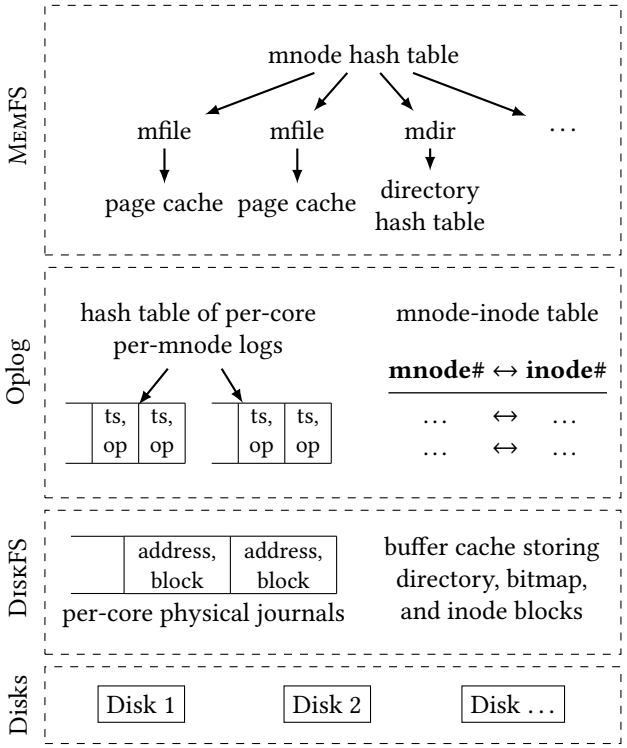

Figure 1: An overview of ScaleFS.

Figure 1 presents an overview of ScaleFS's design. At the top of the figure, MemFS maintains an in-memory cache in a hash table indexed by mnode number. The hash table contains two types of mnodes: mfiles (for files) and mdirs (for directories). Each mfile has a page cache, storing the file's blocks in memory. This cache is implemented using a radix array [9], which allows for scalable access to file pages from many cores. The page cache can be sparsely populated, filling in pages on demand from disk. The page cache also keeps track of dirty bits for file pages.

Directory mnodes (mdirs) maintain an in-memory hash table mapping file names to their mnode numbers. The hash table index is lock-free, but each hash table entry has a per-entry lock. The use of a hash table allows for concurrent lookups and for modifications of different file names from different cores without cache line contention [10], and the per-entry lock serializes updates to individual names in a directory.

At the bottom, DiskFS implements a traditional on-disk file system. The file system uses a physical journal to apply changes atomically to disk. For scalability, DiskFS uses per-core physical journals. DiskFS maintains a buffer cache for on-disk metadata, such as allocation bitmaps and inodes, for efficiently performing partial block writes; for example, updating a single inode in an inode block does not require reading the block from disk in the common case. File data blocks, however, are stored only in the MemFS page caches and not in DiskFS's buffer cache.

Coupling MemFS and DiskFS is an oplog layer. The oplog layer consists of per-core per-mnode logs of modifications to that mnode on a particular core. These logs contain logical directory changes, such as creating a new directory entry, or removing one. The operations are not literally the system calls, but rather capture the effect of the system call on a specific directory; any results needed to complete the system call are produced by MemFS using the in-memory state. For example, the `link` system call creates a log entry that specifies the operation (i.e., `link`), and contains the mnode numbers of the parent directory and the file being linked, and the name of the link being created. Each log entry also contains a timestamp reflecting the order of the operation; as we describe later, we use synchronized CPU timestamp counters for this purpose [5]. When MemFS makes any change to an in-memory directory, it also appends a timestamped log entry to the oplog layer.

When MemFS decides to flush changes to disk (e.g., as a result of an `fsync` on a directory), oplog applies the logged changes to DiskFS, which in turn generates physical journal entries that are written to disk. The oplog layer allows concurrent modifications in MemFS to scale, as long as they do not contend in the in-memory hash table. As a result, concurrent modifications to different file names in the same directory can avoid cache-line conflicts, because these changes are added to different per-core logs. In contrast, in ext4, directory changes immediately update an in-memory copy of the on-disk directory representation, which can incur cache-line conflicts. When ScaleFS finally flushes changes to disk, it incurs cache-line conflicts in merging per-core oplogs (according to the timestamps of the log entries), but this is inevitable, because changes from other cores must be processed on the core executing the `fsync`. Furthermore, the cost of the merge is amortized across all of the operations collected by the `fsync`.

The oplog layer is crucial for ensuring integrity of internal file system data structures (directories and link counts) when flushing directory changes to disk. In the absence of oplog, MemFS would need to scan the in-memory hash table of a directory to determine what changes must be made to the on-disk directory state. To achieve scalability, we would like for `fsync` (and thus this scan) to run concurrently with other directory modifications. As a result, the scan might miss some files altogether that were renamed within a directory, even though the file was never deleted. Flushing these

changes might cause the file to be deleted on disk, and be lost after a crash. Cross-directory renames further complicate matters in the absence of oplog, because MemFS would need to obtain a consistent in-memory snapshot of multiple concurrent hash tables. With oplog, MemFS never needs to scan in-memory directory hash tables.

The above rationale also explains ScaleFS's choice of using oplog for directory changes but not file changes. If an application concurrently modifies a file while calling `fsync` on it, ScaleFS does not specify which of the concurrent changes will be flushed to disk, and might flush concurrent changes out of order. Most importantly, regardless of which concurrent file changes are flushed to disk, there is no risk of ending up with a corrupted on-disk file system.

To provide better intuition of how the pieces of ScaleFS fit together, the rest of this section gives several example system calls and walks through how those system calls are implemented in ScaleFS.

File creation. When an application creates a file, MemFS allocates a fresh mnode number for that file, allocates an mfile structure for the file, adds the mfile to the mnode hash table, and adds an entry to the directory's hash table. MemFS also logs a logical operation to the directory's oplog, containing the new file name and the corresponding mnode number. Many cores can perform file creation concurrently without cache-line conflicts, even when creating files in the same directory (as long as the file names differ). This is because mnode numbers can be allocated with no cross-core communication; the mnode hash table and the directory hash tables can be updated concurrently; and the directory oplog consists of per-core logs that avoid cache line sharing.

`fsync`. When an application calls `fsync` on a directory, MemFS combines the log entries from all per-core logs for the directory's mnode. MemFS then sends the changes in timestamp-sorted order to DiskFS, which makes the changes durable using a physical on-disk journal. For any newly created mnodes that do not yet have inode counterparts, MemFS also allocates on-disk inodes and adds the corresponding tuple to the mnode-inode table.

When an application invokes `fsync` on a file, MemFS scans the file's page cache for any dirty pages, and writes these changes to DiskFS. MemFS also compares the in-memory and on-disk file length, and adjusts the on-disk file length if they differ. If there is no inode number corresponding to the mfile being `fsync`ed, MemFS allocates it first (as above), and then updates the newly allocated on-disk file. Note that this on-disk inode would have a zero link count, and would not appear in any directory, until the application calls `fsync` on some directory containing the file (or a background flush does the same).

Background flush. ScaleFS periodically flushes in-memory changes to disk by invoking `sync`. ScaleFS implements the `sync` system call by iterating over all dirty mfiles and all mnode oplogs, and flushing them to disk by invoking `fsync`. One optimization in this case is that ScaleFS combines the changes into a single physical transaction in DiskFS (to the extent allowed by the maximum size of the journal).

`readdir`. When an application lists the contents of a directory, MemFS simply enumerates the in-memory hash table. If the directory is not present in the mnode hash table, MemFS looks up the directory's inode number and reads the on-disk representation from DiskFS. MemFS then builds up an mdir representation of the directory, including the in-memory hash table. MemFS also translates the inode numbers of the files in that directory, which appear in the on-disk directory format, into mnode numbers, which will appear in the mdir format. MemFS performs this translation with the help of the mnode-inode table. Any inodes that do not have corresponding mnode numbers yet (i.e., have not been accessed since boot) get a fresh mnode number allocated to them; this is why mnode numbers can change across a reboot.

Reading a file. When an application reads a file, MemFS looks up the corresponding page in the mfile's page cache, and returns its contents. If the page is not present, MemFS looks up the corresponding inode number and asks DiskFS to read in the data for that inode number from disk.

Writing to a file. Similarly, when an application writes to a file, MemFS updates the page cache (possibly paging in the page from DiskFS if the application performs a partial page write on a missing page). MemFS also marks the page as dirty, so that it will be picked up by a subsequent `fsync`. Finally, if the write extended the length of the file, MemFS adjusts the mfile's length field in memory.

Crash recovery. After a crash, DiskFS recovers the on-disk file system state by replaying the on-disk journals, after sorting the journal entries by timestamps (since there are multiple journals on disk, corresponding to DiskFS's per-core journals). One subtle issue is orphan inodes: these can result from an application calling `fsync` on a file, and the system crashing before the corresponding directory change was flushed to disk (or, conversely, an application that unlinks a file but retains an open file descriptor to it). Our prototype DiskFS scans the inodes at boot and frees any orphan inodes (allocated inodes with a zero link count).

5 DESIGN

ScaleFS's design faces two challenges—performance and correctness—and the rest of this section describes how ScaleFS achieves both. We explicitly mark the aspects of the design related to either performance or correctness with **[P]** and **[C]** respectively.

5.1 Making operations orderable [P, C]

To ensure crash consistency, operations must be added to the log in the order that MemFS applied the operations. Achieving this property while maintaining conflict freedom is dif-

ficult, because MemFS uses lock-free reads, which in turn makes it difficult to determine the order in which different operations ran. For example, consider a strawman that logs each operation while holding all the locks that MemFS acquires during that operation, and then releases the locks after the operation has been logged. Now consider the scenario in Figure 2. Directory `dir1` contains three file names: `a`, `b` and `c`. `a` is a link to inode 1, `b` and `c` are both links to inode 2. Thread T1 issues the syscall `rename(b, c)` while thread T2 issues `rename(a, c)`.

As an aside, although this may seem like a corner case, our goal is to achieve perfect scalability without having to guess what operations might or might not matter to a given workload. We set this goal because sequences of system calls that appear to be a corner case sometimes show up in real applications and turn out to matter for overall application performance, and it is difficult to determine in advance whether a particular seemingly corner-case situation will or will not show up in practice.

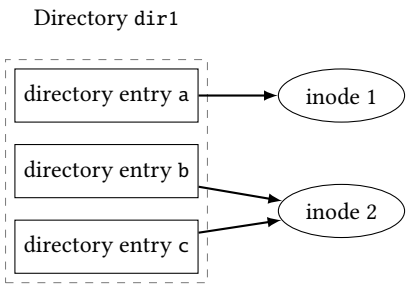

Figure 2: Data structures involved in a concurrent execution of `rename(a,c)` (by thread T2) and `rename(b,c)` (by thread T1). Arrows depict a directory entry referring to an inode. There are three directory entries in a single directory, two of which are hard links to the same inode.

Returning to the example, assume T1 goes first. While T1 and T2 do not commute, to ensure conflict freedom of operations that would commute with T1, T1 performs a lock-free read of `b` and `c` to determine that both are hardlinks to the same inode. Thus, all it needs to do is remove the directory entry for `b` and decrement the inode's reference count, which it can do without holding a lock by using a distributed reference counter like Refcache [9]. The only lock T1 acquires is on `b`, since it does not modify `c` at all. In this case, T1's `rename(b, c)` can now complete without having to modify any cache lines (such as a lock) for `c`.

T2 then acquires locks on `a` and `c` and performs its `rename`. Both of the threads now proceed to log the operations while holding their respective locks. Because the locks they hold are disjoint, T2 might end up getting logged before T1. Now when the log is applied to the disk on a subsequent `fsync`, the disk version becomes inconsistent with what the user believes the file system state is. Even though MemFS has `c` pointing to inode 1 (as a result of T1 executing before T2), DiskFS would have `c` pointing to inode 2 (T2 executing before T1). Worse yet, this inconsistency persists on disk until all file names in question are deleted by the application—it is not just an ephemeral inconsistency that arises at specific crash points. The reason behind this discrepancy is that reads are not guarded by locks. However if we were to acquire read locks, ScaleFS would sacrifice conflict freedom of in-memory commutative operations.

We address this problem by observing that modern processors provide synchronized timestamp counters across all cores. This observation is at the center of oplog's design [5: §6.1], which ScaleFS builds on. On x86 processors, timestamp counters can be accessed using the `RDTSCP` instruction (which ensures the timestamp read is not re-ordered by the processor).

Building on this observation, ScaleFS orders in-memory directory operations by requiring that all directory modifications in the in-memory file system be *linearizable* [19]. Moreover, ScaleFS makes the linearization order explicit by reading the timestamp counter at the appropriate linearization point, which records the order in which MemFS applied these operations. This subsequently allows `fsync` to order the operations by their timestamps, in the same order that they were applied to MemFS, without incurring any additional space overhead.

Timestamping lock-free reads [P, C]. In the above example, the linearization point of `rename(b, c)` must be before the linearization point of `rename(a, c)` because `rename(b, c)`'s lock-free read observes `c` before `rename(a, c)`'s write modifies `c`. There is no explicit synchronization between these operations, yet MemFS must correctly order their linearization points.

To order lock-free reads with writes, MemFS protects such read operations with a seqlock [24: §6], and ensures that such read operations happen before any writes in the same operation. With a seqlock, a writer maintains a sequence number associated with the shared data. Writers update this sequence number both before and after they modify the shared data. Readers read the sequence number before and after reading the shared data. If the sequence numbers are different, or the sequence number is odd (indicating a writer is in the process of modifying), then the reader assumes that a writer has changed the data while it was being read. In that case a reader simply retries until the reader reads the same even sequence number before and after. In the normal case, when a writer does not interfere with a reader, a seqlock does not incur any additional cache-line movement. By performing all reads before any writes, MemFS ensures that it is safe to retry the reads.

To determine the timestamp of the linearization point for an operation that includes a lock-free read, MemFS uses a seqlock around that read operation (such as a lock-free hash table lookup). Inside of the seqlock-protected region, MemFS both performs the lock-free read, and reads the timestamp counter. If the seqlock does not succeed, MemFS retries, which produces a new timestamp. The timestamp of the last

retry corresponds to the linearization point. This scheme ensures that ScaleFS correctly orders operations in its log, since the timestamp falls within a time range when the read value was stable, and thus represents the linearization point. This scheme also achieves scalability because it allows read-only operations to avoid modifying shared cache lines.

5.2 Merging operations [C]

The timestamps of the linearization points allow operations executed by different cores to be merged in a linear order, but the merge must be done with care. ScaleFS's oplog maintains per-core logs so that cores can add entries without communicating with other cores, and merges the per-core logs when an `fsync` or a `sync` is invoked. This ensures that commutative operations do not conflict because of appending log entries. The challenge in using an oplog in ScaleFS is that the merge is subtle. Even though timestamps in each core's log grow monotonically, the same does not hold for the merge, as illustrated by the example shown in Figure 3.

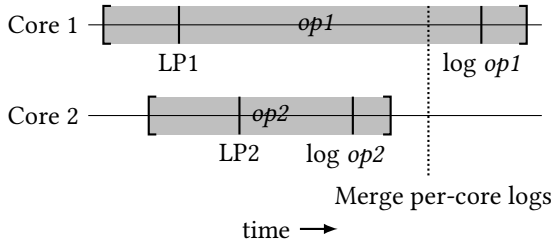

Figure 3: Two concurrent operations, *op1* and *op2*, running on cores 1 and 2 respectively. The grayed-out box denotes the start and end time of an operation. LP denotes the linearization point of each operation. "log" denotes the time at which each operation is inserted into that core's log. The dotted line indicates the time at which the per-core logs are merged.

Figure 3 shows the execution of two operations on two different cores: *op1* on core 1 and *op2* on core 2. The linearization point of *op1* is before the linearization point of *op2*, but *op2* is logged first. If the per-core logs happen to be merged before *op1* is logged, at the time indicated by the dotted line in the figure, *op1* will be missing from the merged log even though its linearization point was before that of *op2*.

To avoid this problem, the merge must wait for in-progress operations to complete. ScaleFS achieves this by tracking, for each core, whether an operation is currently executing, and if so, what its starting timestamp is. To merge, ScaleFS first obtains the timestamp corresponding to the start of the merge, and then waits if any core is running an operation with a timestamp less than that of the merge start.

Concurrent `fsync`s [P]. Per-core operation logs allow different CPUs to log operations in parallel without incurring cache conflicts. However, an `fsync` must merge the logged operations first before proceeding further. This means that having a single set of per-core operation logs for the entire filesystem would introduce a bottleneck when merging the operations, thus limiting the scalability of concurrent `fsync`s even when they are invoked on different files and directories. We solve this problem by using a set of per-core logs for every mnode (file or directory). Per-mnode logs allow ScaleFS to merge the operations for that mnode on an `fsync` without conflicting with concurrent operations on other mnodes. ScaleFS uses oplog's lazy allocation of per-core logs to avoid allocating a large number of logs that are never used [5].

5.3 Flushing an operation log [P, C]

ScaleFS's per-mnode operation logs allow `fsync` to efficiently locate the set of operations that modified a given file or directory, and then flush these changes to disk. There are three interesting complications in this process. First, for performance, ScaleFS should perform absorption when flushing multiple related operations to disk **[P]**. Second, as we discussed in §3, ScaleFS's `fsync` specification requires special handling of cross-directory `rename` to avoid losing files **[C]**. Finally, ScaleFS must ensure that the on-disk file system state is internally consistent, and in particular, that it does not contain any orphan directories, directory loops, links to uninitialized files, etc. **[C]**

In flushing the operations to disk, ScaleFS first computes the set of operations that must be made persistent, and then produces a physical journal representing these operations. Since the operations may exceed the size of the on-disk journal, ScaleFS may need to flush these operations in several batches. In this case, ScaleFS orders the operations by their timestamps, and starts writing operations from oldest to newest. By design, each operation is guaranteed to fit in the journal (the journal must be larger than the maximum operation size, and if the journal is too full, its contents are applied and the journal truncated to make space for the new operation), so ScaleFS can always make progress by writing at least one operation at a time to the journal. Furthermore, since an operation represents a consistent application-level change to the file system, it is always safe to flush an operation on its own (as long as it is flushed in the correct order with respect to other operations).

Absorption [P]. Once `fsync` computes the set of operations to be written to disk in a single batch, it removes operations that logically cancel each other out. For example, suppose the application invokes `fsync` on a directory, and there was a file created and then deleted in that directory, with no intervening `link` or `rename` operations on that file name. In this case, these two operations cancel each other, and it is safe for `fsync` to make no changes to the containing directory (in our prototype, we ignore modification and access times), reducing the amount of disk I/O needed for `fsync`.

One situation where this shows up often is the use of temporary files that are created and deleted before `fsync` is ever called. One subtle issue is that some process can still hold an open file descriptor for the non-existant file. Our

design deals with this by remembering that the in-memory file has no corresponding on-disk representation, and never will. This allows our design to *ignore* any `fsync` operations on an orphaned file's file descriptor.

Cross-directory `rename` [C]. Recall from §3 that ScaleFS needs to avoid losing a file if that file was moved out of some directory `d` and then `d` was flushed to disk. To achieve this goal, ScaleFS implements dependency tracking. Specifically, when ScaleFS encounters a cross-directory `rename` where the directory being flushed, `d`, was either the source or the destination directory, it also flushes the set of operations on the other directory in question (the destination or source of the `rename`, respectively). This requires ScaleFS to access the other directory's operation log as well. As an optimization, ScaleFS does not flush all of the changes to the other directory; it flushes only up to the timestamp of the `rename` in question. This optimizes for flushing the minimal set of changes for a given system call, but for some workloads, flushing all changes to the other directory may give more opportunities for absorption. ScaleFS avoids loops in dependency tracking by relying on timestamps: a dependency points not just to a particular mnode, but to an mnode at a specific timestamp. Since timestamps increase monotonically, there can be no loops.

Internal consistency [C]. ScaleFS must guarantee that its own data structures are intact after a crash. There are three cases that ScaleFS must consider to maintain crash safety for its internal data structures.

First, ScaleFS must ensure that directory links point to initialized inodes on disk. This invariant could be violated if a directory containing a link to a newly created file is flushed. ScaleFS must ensure that the file is flushed first, before a link to it is created. When flushing a directory, ScaleFS also flushes the allocation of new files linked into that directory. In the common case, both are flushed together in a single DiskFS transaction. However, in case ScaleFS builds up a large transaction that does not fit in the journal (e.g., `fsync`ing a directory containing many new files), ScaleFS must break up the changes into multiple DiskFS transactions. In this case, it is crucial that ScaleFS performs the file creations before linking the files into the directory.

Second, ScaleFS must ensure that there are no orphan directories on disk. This invariant could be violated when an application recursively deletes a directory, and then calls `fsync` on the parent. ScaleFS uses dependencies to prevent this problem, by ensuring that all deletion operations on the subdirectory are flushed before it is deleted from its parent.

Finally, ScaleFS must ensure that the on-disk file system does not contain any loops after a crash. This is a subtle issue, which we explain by example. Consider the sequence of operations shown in Figure 4. With all of the above checks (including `rename` dependencies), if an application issues the two `mv` commands shown in Figure 4 and then invokes `fsync(D)`, the `fsync` would flush changes to `D` and `A`, because they were involved in a cross-directory rename. However, flushing just `D` and `A` leads to a directory loop on disk, as Figure 4 illustrates, because changes to `B` (namely, moving `C` from `B` to the root directory) are not flushed.

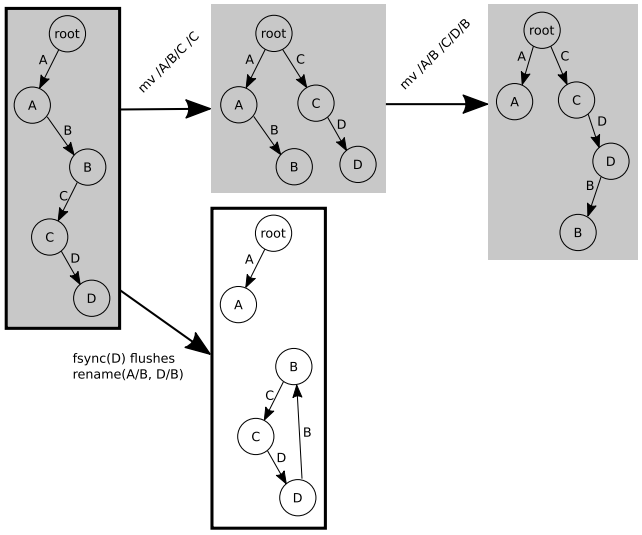

Figure 4: A sequence of operations that leads to a directory cycle (B-C-D-B) on disk with a naïve implementation of `fsync`. The state of the file system evolves as an application issues system calls. Large arrows show the application's system calls. Large rectangles represent the logical state of the file system, either in-memory (shaded) or on disk (thick border). The initial state of the file system, on the left, is present both on disk and in memory at the start of this example. Circles represent directories in a file system tree.

To avoid directory loops on disk, ScaleFS follows three rules. First, in-memory renames of a subdirectory between two different parent directories are serialized with a global lock. Although the global lock is undesirable from a scalability point of view, it allows for a simple and efficient algorithm that prevents loops in memory, and also helps avoid loops on disk in combination with the next two rules. Furthermore, we expect this lock to be rarely contended, since we expect applications to not rename subdirectories between different parent directories in their critical path. Second, when a subdirectory is moved into a parent directory `d`, ScaleFS records the path from `d` to the root of the file system. This list of ancestors is well defined, because ScaleFS is holding a global lock that prevents other directory moves. Third, when flushing changes to some directory `d`, if a child subdirectory was moved into `d`, ScaleFS first flushes to disk any changes to the ancestors of `d` as recorded by that `rename` operation. Intuitively, this ensures that the child being moved into `d` does not also appear to be `d`'s parent. As an optimization, ScaleFS flushes ancestor changes only up to the timestamp of the `rename` operation. A thesis describing ScaleFS presents an argument for the correctness of this algorithm [3: §A].

5.4 Multiple disks and journals [P]

On a computer with multiple disks or multiple I/O queues to a single disk (as in NVMe), ScaleFS should be able to take advantage of the aggregate disk throughput and parallel I/O queues by flushing in parallel to multiple journals. This would allow two cores running `fsync` to execute completely in parallel, not contending on the disk controller or bottlenecking on the same disk's I/O performance.

To take advantage of multiple disks for file data, ScaleFS stripes the DiskFS data across all of the physical disks in the system. ScaleFS also allocates a separate on-disk journal for every core, to take advantage of multiple I/O queues, and spreads these journals across disks to take advantage of multiple physical disks. The challenges in doing so are constructing and flushing the journal entries in parallel, even when there may be dependencies between the blocks being updated by different cores.

To construct the journal entries in parallel, ScaleFS uses two-phase locking on the physical disk blocks. This ensures that, if two cores are flushing transactions that modify the same block, they are ordered appropriately. Two-phase locking ensures that the result is serializable.

Dependencies between transactions also show up when flushing the resulting journal entries to disk. Suppose two transactions, T1 and T2, update the same block. Two-phase locking will ensure that their journal entries are computed in a serializable fashion. However, even if T1 goes first with two-phase locking, T2 may end up being flushed to disk first. If the computer crashes at this point, T2 will be recovered but T1 will not be. This will result in a corrupted file system.

To address this problem, ScaleFS uses timestamps to defer flushing dependent journal entries to disk. Specifically, ScaleFS maintains an in-memory hash table recording, for each disk block, what physical disk contains the last journal entry updating it, and the timestamp of that journal entry. When ScaleFS is about to flush a journal entry to a physical disk, it looks up all of the disk blocks from that journal entry in the hash table, and, for each one, ensures that its dependencies are met. This means waiting for the physical disk indicated in the hash table to flush journal entries up to the relevant timestamp. When ScaleFS finishes flushing a journal entry to disk (i.e., the disk returns success from a barrier command), ScaleFS updates an in-memory timestamp to reflect that this journal's timestamp has made it to disk.

An alternative approach that avoids waiting would be to explicitly include the dependency information in the on-disk journal. On recovery, DiskFS would need to check that all dependencies of a transaction have been satisfied before applying that transaction. In our example, this would prevent T2 from being applied, because its on-disk dependency list includes T1, and T1 was not found during recovery.

5.5 Discussion

The design described above achieves ScaleFS's three goals. First, ScaleFS achieves good multicore scalability, because operations that commute should be mostly conflict-free. Since MemFS is decoupled from DiskFS, it is not restricted in the choice of data structures that allow for scalability. All that MemFS must do is log operations with a linearization timestamp in the operation log, which is a per-core data structure. As a result, commutative file system operations should run conflict-free when manipulating in-memory files and directories and while logging those operations in the operation log. `fsync` is also conflict-free for file system operations it commutes with (e.g., creation of a file in an unrelated directory).

Second, ScaleFS ensures crash safety through dependency tracking and loop avoidance protocols described above. Third, ScaleFS flushes close to the minimal amount of data to disk on every `fsync` operation. In most cases, `fsync` flushes just the changes to the file or directory in question. For cross-directory `renames`, ScaleFS flushes operations on other directories involved in the `rename`, and also on ancestor directories in case of a subdirectory rename. Although these can be unnecessary in some cases, the evaluation shows that in practice ScaleFS achieves high throughput for `fsync` (and ScaleFS flushes less data than the ext4 file system on Linux).

Finally, ScaleFS achieves good disk throughput through its optimizations (such as absorption and group commit). ScaleFS avoids flushing unnecessary changes to disk when applications invoke `fsync`, which in turn reduces the amount of data written to disk, and also enables better absorption and grouping. Finally, ScaleFS takes advantage of multiple disks to further improve disk throughput.

6 IMPLEMENTATION

We implemented ScaleFS in sv6, a research operating system whose design is centered around the Scalable Commutativity Rule [10]. MemFS is based on sv6's in-memory file system, modifying it to interact with the operation log. ScaleFS augments sv6's design with a disk file system DiskFS (that is based on the xv6 file system [12]), and an operation log. Figure 5 shows the number of lines of code involved in implementing each component of ScaleFS. ScaleFS is open-source and available at https://github.com/mit-pdos/scalefs.

ScaleFS component	Lines of C++ code
MemFS (§6.1)	2,458
DiskFS (§6.2)	2,331
MemFS–DiskFS interface	4,094

Figure 5: Lines of C++ code for each component of ScaleFS.

ScaleFS does not support the full set of file system calls in POSIX, such as `sendfile` and `splice`; ScaleFS does not support triply indirect blocks (limiting file sizes); and ScaleFS does not evict file system caches in response to memory pressure. However, ScaleFS does support the major operations needed by applications, specifically `creat`, `open`, `openat`, `mkdir`, `mkdirat`, `mknod`, `dup`, `dup2`, `lseek`, `read`, `pread`,

`write`, `pwrite`, `chdir`, `readdir`, `pipe`, `pipe2`, `stat`, `fstat`, `link`, `unlink`, `rename`, `fsync`, `sync`, and `close`. We believe that supporting the rest of the POSIX operations would not affect ScaleFS's design.

6.1 MemFS

Decoupling the in-memory and the on-disk file systems allows MemFS to be designed for multicore concurrency. MemFS represents directories using chained hash tables that map file/directory names to mnode numbers. Each bucket in the hash table has its own lock, allowing commutative operations to remain conflict-free with high probability.

Files use radix arrays to represent their pages, with each page protected by its own lock; this design follows RadixVM's representation of virtual memory areas [9]. MemFS creates the in-memory directory tree on demand, reading in components from the disk as they are needed.

To allow concurrent creates to scale, mnode numbers in MemFS are independent of inode numbers on the disk and might even change each time the file system is mounted (e.g., after a reboot). MemFS assigns mnode numbers in a scalable manner by maintaining per-core mnode counts. On creation of a new mnode by a core, MemFS computes its mnode number by appending the mnode count on that core with the core's CPU ID, and then increments the core's mnode count. MemFS never reuses mnode numbers. However, if an mnode is evicted to disk and then brought back into memory, ScaleFS preserves the mnode number, using the mnode-inode table.

ScaleFS maintains a hash table from mnode numbers to inode numbers and vice versa for fast lookup. ScaleFS creates this hash table when the system boots up, and adds entries to it each time a new inode is read in from the disk and a corresponding mnode created in memory. Similarly ScaleFS adds entries to the hash table when an inode is created on disk corresponding to a new mnode in memory.

ScaleFS does not update the hash table immediately after the creation of an mnode; it waits for DiskFS to create the corresponding inode on a `sync` or an `fsync` call, which is when DiskFS looks up the on-disk free inode list to find a free inode. This means that MemFS does not allocate an on-disk inode number right away for a `create` operation.

Operations in the log. If MemFS were to log all file system operations, the operation log would incur a large space overhead. For example, writes to a file would need to store the entire byte sequence that was written to the file. So MemFS logs operations selectively.

By omitting certain operations from the log, ScaleFS not only saves space that would otherwise be wasted to log them, but it also simplifies merging the per-core logs and dependency tracking as described in §5.3. As a result the per-core logs can be merged and operations applied to the disk much faster.

To determine which operations are logged, MemFS divides all metadata into two categories: ologged and non-ologged. All directory state is ologged metadata, and a file's link count, which is affected by directory operations, is also ologged metadata. However, other file metadata, such as the file's length, its modification times, etc., is not ologged. For example, MemFS logs the creation of a file since it modifies directory state as well as a file's link count. On the other hand, MemFS does not log a change in file size, or a write to a file, since it affects non-ologged file data and metadata.

When flushing a file from MemFS to DiskFS, ScaleFS must combine the ologged and non-ologged metadata of the flushed mnode. It updates the ologged metadata of the mnode by merging that mnode's olog. The non-ologged part of the mnode (such as a file's length and data contents) are directly accessed by reading the current in-memory state of that mnode. These changes are then written to disk using DiskFS, which is responsible for correctly representing these changes on disk, including allocating disk blocks, updating inode block pointers, etc. DiskFS journals all metadata (both metadata that was ologged as well as non-ologged file metadata such as the file's length), but DiskFS writes file data directly to the file's blocks, which is similar to how Linux ext4 writes data in `data=ordered` mode.

When MemFS flushes an ologged change to DiskFS that involves multiple mnodes (such as a cross-directory `rename`), ScaleFS ensures that this change is flushed to disk in a single DiskFS transaction. This in turn guarantees crash safety: after a crash and reboot, either the `rename` is applied, or none of its changes appear on disk.

6.2 DiskFS

ScaleFS's DiskFS is based on the xv6 [12] file system, which has a physical journal for crash recovery. The file system follows a simple Unix-like format with inodes, indirect, and double-indirect blocks to store files and directories. DiskFS maintains a buffer cache to cache physical disk blocks that are read in from the disk. DiskFS does not cache file data blocks, since they would be duplicated with any cache maintained by MemFS. However, DiskFS does use the buffer cache to store directory, inode, and bitmap blocks, to speed up read-modify-write operations.

DiskFS implements high-performance per-core allocators for inodes and disk blocks by carving out per-core pools of free inodes and free blocks during initialization. This enables DiskFS to satisfy concurrent requests for free inodes and free blocks from `fsync`s in a scalable manner, as long as the per-core pools last. When the per-core free inode or free block pool runs out on a given core, DiskFS satisfies the request by either allocating from a global reserve pool or borrowing free inodes and free blocks from other per-core pools.

These scalability optimizations improve the scalability of `fsync`, and the scalability of reading data from disk into memory. However, when an application operates on a file or directory that is already present in memory, no DiskFS code is invoked. For instance, when an application looks up

a non-existent name in a directory, MemFS caches the entire directory, and does not need to invoke `namei` on DiskFS. Similarly, when an application creates or grows a file, no DiskFS state is updated until `fsync` is called.

7 EVALUATION

This section evaluates ScaleFS's overall performance and scalability by answering the following questions:

- Does ScaleFS achieve conflict freedom for commutative operations? (§7.2)
- Does durability introduce unnecessary conflicts in ScaleFS? (§7.3)
- Does ScaleFS empirically achieve good scalability and performance on real hardware? (§7.4)
- Does ScaleFS achieve good disk throughput? (§7.5)
- What overheads are introduced by ScaleFS's split of MemFS and DiskFS? (§7.6)

7.1 Methodology

To measure scalability of ScaleFS, we primarily rely on Commuter [10] to determine if commutative filesystem operations incur cache conflicts, limiting scalability. This allows us to reason about the scalability of a file system without having to commit to a particular workload or hardware configuration. We disable background sync to obtain repeatable measurements.

To confirm the scalability results reported by Commuter, we also experimentally evaluate the scalability of ScaleFS by running it on an 80-core machine with Intel E7-8870 2.4 GHz CPUs and 256 GB of DRAM, running several workloads. We also measure the absolute performance achieved by ScaleFS, as well as measuring ScaleFS's disk performance, comparing the performance with a RAM disk, with a Seagate Constellation.2 ST9500620NS rotational hard drive, and with up to four Samsung 850 PRO 256 GB SSDs.

To provide a baseline for ScaleFS's scalability and performance results, we compare it to the results achieved by the Linux ext4 filesystem running in Linux kernel version 4.9.21. We use Linux ext4 as a comparison because ext4 is widely used in practice, and because its design is reasonably scalable, as a result of many improvements by kernel developers. Linux ext4 is one of the more scalable file systems in Linux [31].

7.2 Does ScaleFS achieve conflict freedom?

To evaluate ScaleFS's scalability, we used Commuter, a tool that checks if shared data structures experience cache conflicts. Commuter takes as input a model of the system and the operations it exposes, which in our case is the kernel with the system calls it supports, and computes all possible pairs of commutative operations. Then it generates test cases that check if these commutative operations are actually conflict-free by tracking references to shared memory addresses. Shared memory addresses indicate sharing of cache lines and hence loss of scalability, according to the Scalable

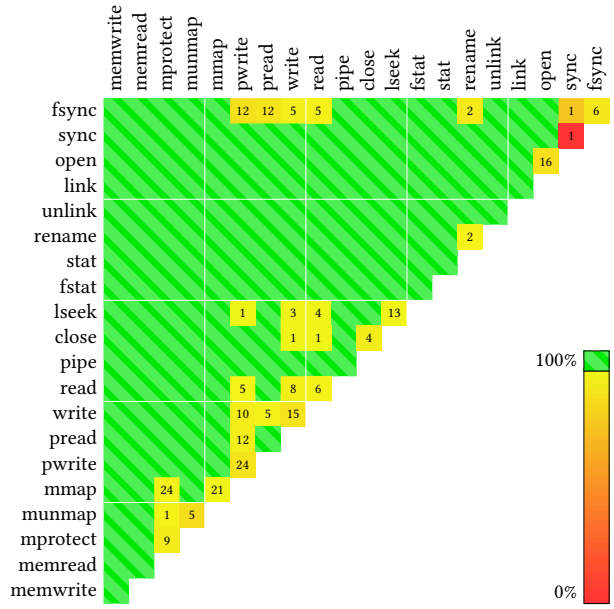

Figure 6: Conflict-freedom of commutative operations in ScaleFS. Numbers in boxes indicate the absolute number of test cases that had conflicts. Out of 31,551 total test cases generated, 31,317 (99.2%) were conflict-free.

Commutativity Rule [10]. We augmented the model Commuter uses to generate test cases by adding the `fsync` and `sync` system calls, and used the resulting test cases to evaluate the scalability of ScaleFS.

Figure 6 shows results obtained by running Commuter with ScaleFS. 99% of the test cases are conflict-free. The green regions show that the implementation of MemFS is conflict free for almost all commutative file system operations not involving `sync` and `fsync`, as there is no interaction with DiskFS involved at this point. MemFS simply logs the operations in per-core logs, which is conflict-free. MemFS also uses concurrent data structures that avoid conflicts.

ScaleFS does have some conflicts when `fsync` or `sync` calls are involved. Some of the additional conflicts incurred by `fsync` are due to dependencies between different files or directories being flushed. Specifically, our Commuter model says that `fsync` of one inode commutes with changes to any other inode. However, this does not capture the dependencies that must be preserved when flushing changes to disk to ensure that the on-disk state is consistent. As a result, `fsync` of one inode accesses state related to another inode (such as its oplog and its dirty bits).

Some `fsync` calls conflict with commutative `read` operations. These conflicts are brought about by the way MemFS implements the radix array of file pages. In order to save space, the radix array element stores certain flags in the last few bits of the page pointer itself, the page dirty bit being one of them. As a result, an `fsync` that resets the page dirty bit conflicts with a read that accesses the page pointer.

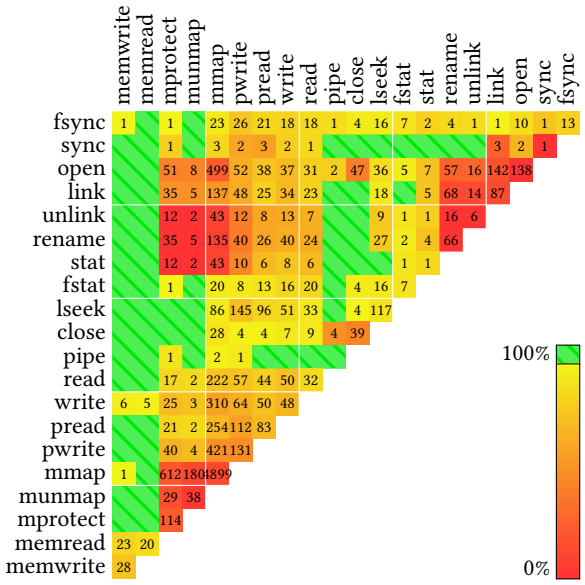

Figure 7: Conflict-freedom of commutative operations in the Linux kernel using an ext4 file system. Numbers in boxes indicate the absolute number of test cases that had conflicts. Out of 31,551 total test cases generated, 20,394 (65%) were conflict-free.

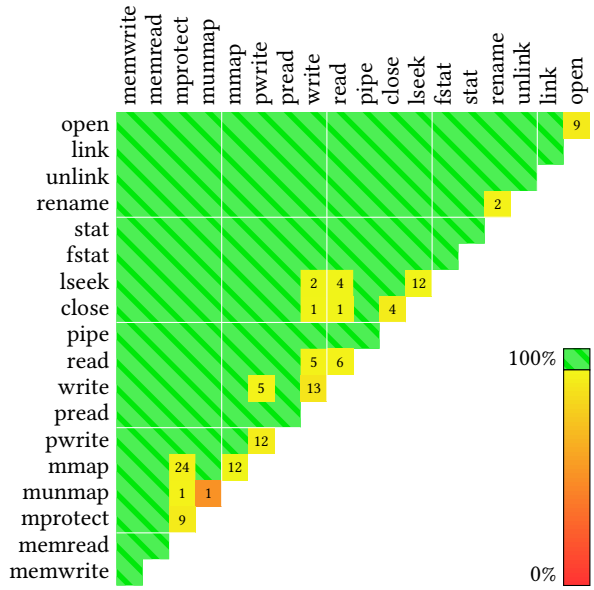

Figure 8: Conflict-freedom between commutative operations in sv6 with only an in-memory file system. Numbers in boxes indicate the absolute number of test cases that had conflicts. Out of 13,664 total test cases generated, 13,528 (99%) were conflict-free.

MemFS could, in principle, avoid these conflicts by keeping track of the page dirty bits outside of the page pointer. But in that case, as long as the dirty flags are stored as bits, there would be conflicts between accesses to dirty bits of distinct pages. In order to provide conflict freedom MemFS would need to ensure that page dirty flags of different pages do not share a cache line, which would incur a huge space overhead. Our implementation of MemFS makes this trade-off, saving space at the expense of incurring conflicts in some cases.

`sync` conflicts only with itself, and its only conflict is in acquiring the locks to flush per-core logs. Although we could, in principle, optimize this by using lock-free data structures, we do not believe that applications would benefit from concurrent calls to `sync` being conflict-free. Note that concurrent calls to `sync` and every other system call *are* conflict-free.

The rest of the conflicts are between idempotent operations. Two `fsync` calls are commutative because they are idempotent, but they both contend on the operation log as well as the file pages. `fsync` and `pwrite` also conflict despite being commutative when `pwrite` performs an idempotent update.

To provide a baseline for ScaleFS's heatmap, we also ran Commuter on the Linux kernel with an ext4 file system.[2] The heatmap in Figure 7 shows the results obtained. Out of a total of 31,551 commutative test cases, the heatmap shows 11,157 of them (35%) conflicting in the Linux kernel. Many of these can be attributed to the fact that the in-memory representation of the file system is closely coupled with the disk representation. Some commutative operations conflict in the page cache layer. These results are in line with the bottlenecks uncovered in prior work [4, 31]. Manually analyzing the source of some of these bottlenecks indicates that they are the result of maintaining the on-disk data structure while executing in-memory operations. For example, when creating files in a directory, multiple cores contend on the lock protecting that directory, which is necessary to serialize updates to the on-disk directory structure. This is precisely the problem that ScaleFS's split design avoids.

[2]We ran Commuter on Linux kernel version v3.16 (from 2014) because we have not ported the Linux changes necessary to run Commuter to a more recent version of the kernel. We expect that the results are not significantly different from those that would be obtained on a recent version of Linux, based on recent reports of Linux file system scalability [31].

7.3 Does durability reduce conflict freedom?

To evaluate the scalability impact of ScaleFS's approach for achieving durability, we compare the results of Commuter on ScaleFS to the results of running Commuter on the original sv6 in-memory file system, on which MemFS is based. Figure 8 shows results for sv6; this heatmap does not have a column for `fsync` or `sync` because sv6 did not support these system calls (it had no support for durability). Compared to Figure 6, we see that ScaleFS introduces almost no new conflicts between existing syscalls. The exceptions have to do with conflicts in various bitmaps, such as `pread` conflicting with `pwrite` on the bits keeping track of the state of a page in a file (including its dirty bit). Although ScaleFS could avoid these cache line conflicts, its absolute performance and memory cost would be worse.

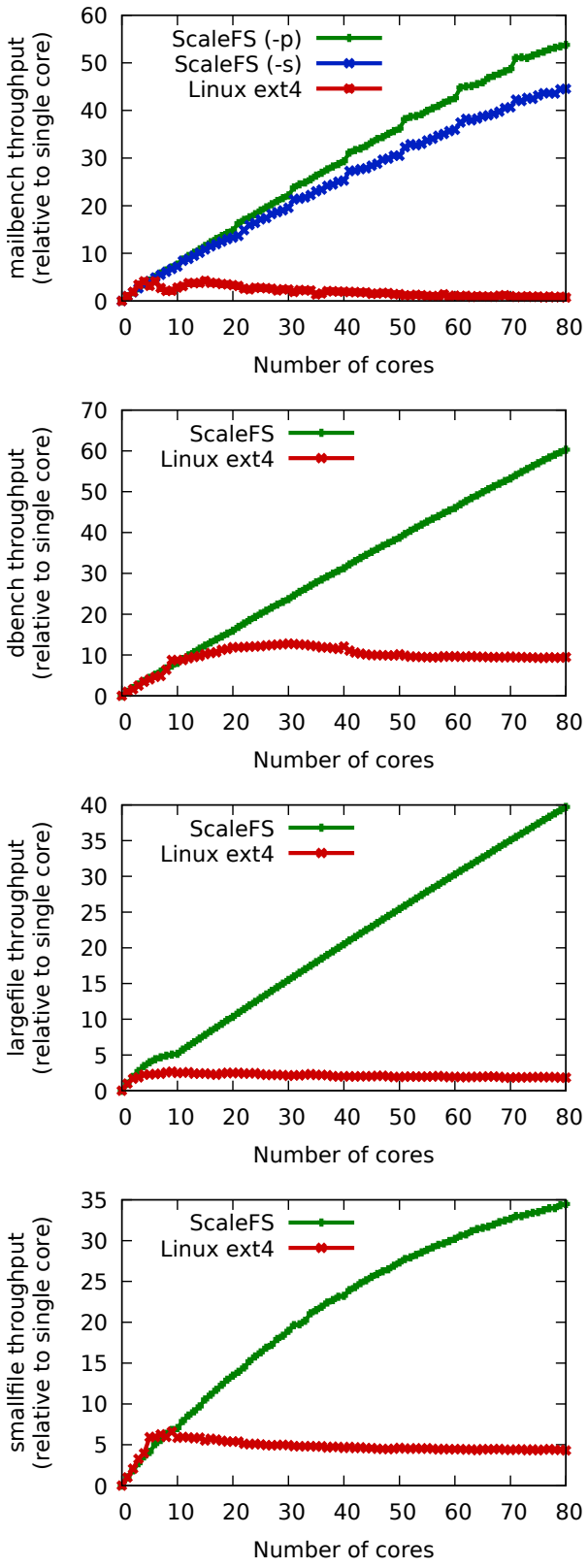

Figure 9: Throughput of the mailbench, dbench, largefile, and smallfile workloads respectively, for both ScaleFS and Linux ext4 in a RAM disk configuration.

7.4 Empirical scalability

To confirm that Commuter's results translate into scalability for actual workloads on real hardware, we compared the performance of several workloads on ScaleFS to their performance on Linux ext4; unfortunately, porting applications to sv6 is difficult because sv6 is not a full-featured OS. To avoid the disk bottleneck, we ran these experiments on a RAM disk. Both ScaleFS and ext4 still performed journaling, flushing, etc., as they would on a real disk. §7.5 presents the performance of both file systems with real disks.

mailbench. One of our workloads is mailbench, a qmail-like mail server benchmark from sv6 [10]. The version of mailbench used by sv6 focused on in-memory file system scalability. To make this workload more realistic, we added calls to `fsync`, both for message files and for the containing directories, to ensure mail messages are queued and delivered in a crash-safe manner (6 `fsync`s total to deliver one message). The benchmark measures the number of mail messages delivered per second. We run mailbench with per-core spools, and with either per-core user mailboxes (**mailbench-p**) or with 1,000 shared user mailboxes (**mailbench-s**). **mailbench-s** in particular requires all cores to read and write the same set of shared mailboxes; this workload would be difficult to scale on a file system that partitioned directories across cores (such as Hare or SpanFS).

dbench. The dbench benchmark [41] generates an I/O workload intended to stress file systems and file servers. This workload is more read-heavy than mailbench: the configuration used in our experiments performs a mix of 124,199 reads and 39,502 writes (in a loop), as well as other operations (79,230 creates, 16,000 `unlink`s, 5,553 `fsync`s, and 3,355 `rename`s). We run the benchmark with the `--sync-dir` flag to call `fsync` after `unlink` and `rename` operations. This configuration stresses both MemFS and DiskFS. The dbench benchmark was used by FxMark [31] to demonstrate scalability bottlenecks in Linux file systems.

largefile. Inspired by the LFS benchmarks [39], largefile creates a 100 MByte file and calls `fsync` after creating it. Each core runs a separate copy of the benchmark, creating and `fsync`ing its own 100 MByte file. All of the files are in the same directory. We report the combined throughput achieved by all cores, in MB/sec.

smallfile. The smallfile microbenchmark creates a new file, writes 1 KByte to it, `fsync`s the file, and deletes the file, repeated 10,000 times (for different file names). Each core runs a separate copy of the smallfile benchmark, each of which performs 10,000 iterations. The files are spread among 100 directories that are shared across all cores. We report the combined throughput achieved by all cores, in files/sec.

Results. Figure 9 shows the results. ScaleFS scales well for all of our workloads, achieving approximately 40× performance at 80 cores. ScaleFS does not achieve perfect 80× scalability because going across sockets is more expensive than accessing cache and DRAM within a single socket (which occurs with 1-10 cores), and because multiple cores contend

for the same shared L3 cache. Linux ext4 fails to scale for all workloads, achieving no more than 13× the performance of a single core, and collapsing at 80 cores. We run only the mailbench-p variant on Linux since it is more scalable.

7.5 Disk performance

Single disk, single core. To evaluate whether ScaleFS can achieve good disk throughput, we compare the performance of our workloads running on ScaleFS to their performance on Linux ext4. Figure 10 shows the results, running with a single disk and a single CPU core. ScaleFS achieves comparable or better performance to Linux ext4 in all cases. For the smallfile microbenchmark, ScaleFS achieves higher throughput because ScaleFS's precise `fsync` design flushes only the file being `fsync`ed. Linux ext4, on the other hand, maintains a single journal, which means that flushing the `fsync`ed file also flushes all other preceding entries in the journal as well, which includes the modification of the parent directory. ScaleFS achieves higher performance than Linux ext4 on the dbench benchmark for the same reason.

Disk	Benchmark	ScaleFS	Linux ext4
RAM disk	mailbench-p	641 msg/sec	675 msg/sec
	dbench	317 MB/sec	216 MB/sec
	largefile	331 MB/sec	378 MB/sec
	smallfile	7151 files/sec	3553 files/sec
SSD	mailbench-p	61 msg/sec	66 msg/sec
	dbench	47 MB/sec	28 MB/sec
	largefile	180 MB/sec	180 MB/sec
	smallfile	364 files/sec	277 files/sec
HDD	mailbench-p	9 msg/sec	9 msg/sec
	dbench	7 MB/sec	5 MB/sec
	largefile	83 MB/sec	92 MB/sec
	smallfile	51 files/sec	27 files/sec

Figure 10: Performance of workloads on ScaleFS and Linux ext4 on a single disk and a single core.

Multiple disks, 4 cores. Since the disk is a significant bottleneck for both ScaleFS and ext4, we also investigated whether ScaleFS can achieve higher disk throughput with additional physical disks. In this experiment, we striped several SSDs together, and ran either ScaleFS or Linux ext4 on top. To provide sufficient parallelism to take advantage of multiple disks, we ran the workload using 4 CPU cores. Figure 11 shows the results for the four workloads. For the largefile workload, both ScaleFS and Linux cannot obtain more throughput because the SATA controller is saturated.

With four disks, ScaleFS achieves more throughput than Linux for mailbench, dbench, and smallfile. This is because when Linux ext4's `fsync` issues a barrier to the striped disk, the Linux striping device forwards the barrier to every disk in the striped array. ScaleFS is aware of multiple disks, and its `fsync` issues a barrier only to the disks that have been written to by that `fsync`.

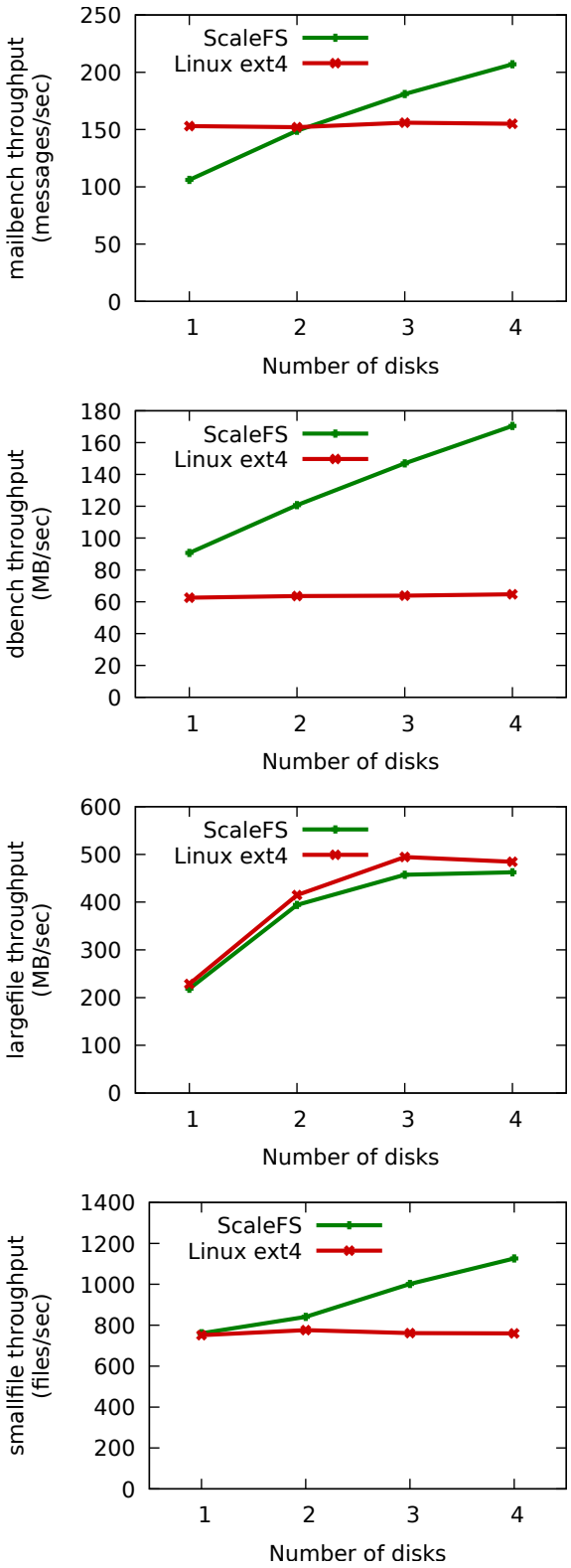

Figure 11: Throughput of the mailbench-p, dbench, largefile, and smallfile workloads respectively, for both ScaleFS and Linux ext4, using 4 CPU cores. The x-axis indicates the number of SSDs striped together.

7.6 Overhead of splitting MemFS and DiskFS

The main worry about ScaleFS's split of a separate in-memory MemFS file system and an on-disk DiskFS file system is that ScaleFS might incur higher memory overhead because it has to keep the operation log in memory and because metadata (e.g., block allocator state, inodes, and directory contents) may be kept in memory twice: once in MemFS's in-memory representation and once in DiskFS's buffer cache. However, file data blocks are not stored twice.

To evaluate how severe this overhead is, we measured the peak memory usage for each benchmark. Figure 12 shows the results. For largefile, there is little metadata, so the overhead is minimal. For smallfile, the overhead is significant because ScaleFS allocates an oplog for each directory and, in our prototype, for each file as well. At 25 KB per oplog, 10,000 oplogs translates into 250 MBytes of memory overhead. The memory overhead for mailbench is also dominated by the oplogs. The mailbench-s configuration in particular creates 1,000 user mailboxes, with each Maildir-format mailbox consisting of four directories (a parent directory and three subdirectories). In the mailbench-p configuration, there are only 80 user mailboxes (one per core), which accounts for the lower memory usage. For dbench, there are relatively fewer files and directories involved, and ScaleFS does not introduce much memory overhead.

Benchmark	Memory use in MemFS	Memory use in ScaleFS	ScaleFS overhead
mailbench-p	12 MB	21 MB	75%
mailbench-s	515 MB	770 MB	49.5%
dbench	84 MB	94 MB	11.9%
largefile	106 MB	107 MB	0.9%
smallfile	296 MB	561 MB	89.5%

Figure 12: Peak memory use in MemFS and ScaleFS during the execution of different benchmarks.

8 CONCLUSION

It is a challenge to achieve multicore scalability, durability, and crash consistency in a file system. This paper proposes a new design that addresses this challenge using the insight of completely decoupling the in-memory file system from the on-disk file system. The in-memory file system can be optimized for concurrency and the on-disk file system can be tailored for durability and crash consistency. To achieve this decoupling, this paper introduces an operation log that extends oplog [5] with a novel scheme to timestamp the logged operations at their linearization points in order to apply them to the disk in the same order a user process observed them in memory. The operation log also minimizes the data that must be written out at an `fsync` by computing dependencies and absorbing operations that cancel out each other. We implemented this design in a prototype file system, ScaleFS, that was built on the existing sv6 kernel and we analyzed the implementation using Commuter. The results show that the implementation of ScaleFS achieves good multicore scalability.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, and our shepherd, Remzi Arpaci-Dusseau. This research was supported by NSF award CNS-1301934.

REFERENCES

[1] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 69–78, Calgary, Canada, Aug. 2009.

[2] D. J. Bernstein. qmail internals, 1998. http://www.qmail.org/man/misc/INTERNALS.txt.

[3] S. S. Bhat. Designing multicore scalable filesystems with durability and crash consistency. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 2017.

[4] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Vancouver, Canada, Oct. 2010.

[5] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Sept. 2014.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186, New Orleans, LA, Feb. 1999.

[7] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 101–116, San Jose, CA, Feb. 2012.

[8] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode file system. In *Proceedings of the Winter 1992 USENIX Technical Conference*, pages 43–59, Jan. 1992.

[9] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM EuroSys Conference*, pages 211–224, Prague, Czech Republic, Apr. 2013.

[10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Farmington, PA, Nov. 2013.

[11] J. Corbet. Dcache scalability and RCU-walk, Apr. 2012. http://lwn.net/Articles/419811/.

[12] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system, 2016. http://pdos.csail.mit.edu/6.828/xv6.

[13] M. Curtis-Maury, V. Devadas, V. Fang, and A. Kulkarni. To Waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 419–434, Savannah, GA, Nov. 2016.

[14] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the 9th ACM EuroSys Conference*, Amsterdam, The Netherlands, Apr. 2014.

[15] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 307–320, Stevenson, WA, Oct. 2007.

[16] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–60, Monterey, CA, Nov. 1994.

[17] C. Gruenwald, III, F. Sironi, M. F. Kaashoek, and N. Zeldovich. Hare: a file system for non-cache-coherent multicores. In *Proceedings of the 10th ACM EuroSys Conference*, Bordeaux, France, Apr. 2015.

[18] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 155–162, Austin, TX, Nov. 1987.

[19] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems*, 12(3):463–492, 1990.

[20] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Los Angeles, CA, Aug. 1994.

[21] M. Jambor, T. Hruby, J. Taus, K. Krchak, and V. Holub. Implementation of a Linux log-structured file system with a garbage collector. *ACM SIGOPS Operating Systems Review*, 41(1):24–32, Jan. 2007.

[22] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai. SpanFS: A scalable file system on fast storage devices. In *Proceedings of the 2015 USENIX Annual Technical Conference*, Santa Clara, CA, July 2015.

[23] Y. Klonatos, M. Marazakis, and A. Bilas. A scaling analysis of Linux I/O performance. Poster presented at EuroSys, 2011. http://eurosys2011.cs.uni-salzburg.at/pdf/postersubmission/eurosys11-poster-klonatos.pdf.

[24] C. Lameter. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*, May 2005. http://www.lameter.com/gelato2005.pdf.

[25] E. Lee, H. Bahn, and S. H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 73–80, San Jose, CA, Feb. 2013.

[26] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–44, San Jose, CA, Feb. 2013.

[27] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 81–96, Broomfield, CO, Oct. 2014.

[28] Y. Lu, J. Shu, and W. Wang. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 75–88, Santa Clara, CA, Feb. 2014.

[29] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, pages 21–34, Ottawa, Canada, June 2007.

[30] P. E. McKenney, D. Sarma, and M. Soni. Scaling dcache with RCU. *Linux Journal*, 2004(117), Jan. 2004.

[31] C. Min, S. Kashyap, S. Maass, and T. Kim. Understanding manycore scalability of file systems. In *Proceedings of the 2016 USENIX Annual Technical Conference*, Denver, CO, June 2016.

[32] D. Park and D. Shin. iJournaling: Fine-grained journaling for improving the latency of fsync system call. In *Proceedings of the 2017 USENIX Annual Technical Conference*, pages 787–798, Santa Clara, CA, July 2017.

[33] S. Park, T. Kelly, and K. Shen. Failure-atomic `msync()`: A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM EuroSys Conference*, pages 225–238, Prague, Czech Republic, Apr. 2013.

[34] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, Oct. 2014.

[35] T. S. Pillai, R. Alagappana, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Application crash consistency and performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 181–196, Santa Clara, CA, Feb.–Mar. 2017.

[36] K. Ren and G. Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Annual Technical Conference*, pages 145–156, San Jose, CA, June 2013.

[37] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[38] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage*, 9(3): 9:1–32, Aug. 2013.

[39] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Pacific Grove, CA, Oct. 1991.

[40] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference*, San Diego, CA, Jan. 1996.

[41] A. Tridgell and R. Sahlberg. DBENCH, 2013. https://dbench.samba.org/.

[42] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, Santa Clara, CA, Feb. 2016.

[43] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, Nov. 2006.