Designing multicore scalable filesystems with durability and crash consistency

by

Srivatsa S. Bhat

B.Tech in Information Technology
National Institute of Technology Karnataka, Surathkal, India (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2017

Certified by . . . .
M. Frans Kaashoek
Charles Piper Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . .
Nickolai Zeldovich
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Designing multicore scalable filesystems with durability and crash consistency

by

Srivatsa S. Bhat

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2017, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

It is challenging to simultaneously achieve multicore scalability and high disk throughput in a file system. For example, data structures that are on separate cache lines in memory (e.g., directory entries) are grouped together in a transaction log when the file system writes them to disk. This grouping results in cache line conflicts, thereby limiting scalability.

MCOREFS 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 conflicts and hence scale perfectly. MCOREFS logs operations in a per-core log so that it can delay propagating updates to the disk representation until an `fsync`. The `fsync` call merges the per-core logs and applies the operations to disk. MCOREFS 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 MCOREFS show that its implementation is conflict-free for 99% of test cases involving commutative operations generated by COMMUTER, scales well on an 80-core machine, and provides disk performance that matches or exceeds that of Linux ext4.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to express my deepest gratitude to my advisors Frans Kaashoek and Nickolai Zeldovich for their incredible guidance and support. I feel very fortunate to be a part of PDOS, and I would like to thank everyone for all the enlightening discussions. My heartfelt thanks to my brother and my parents for their constant support and encouragement through all my endeavors.

This thesis incorporates work done jointly with Rasha Eqbal, and builds on top of Rasha's Masters thesis on the ScaleFS multicore file system [13].

Contents

List of Figures

Chapter 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 [2, 9, 12, 21, 24, 28]. This thesis 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, McoreFS, that implements this design.

McoreFS achieves the following goals:

1. **Multicore scalability.** McoreFS scales well for a number of workloads on an 80-core machine, but even more importantly, the McoreFS implementation is conflict-free for almost all commutative operations [9]. This allows McoreFS to achieve disjoint-access parallelism [1, 18], which suggests that McoreFS will continue to scale even for workloads or machines we have not yet measured.

2. **Crash safety.** McoreFS recovers from a crash at any point and provides clear guarantees for `fsync`, consistent with what the POSIX specification implies (e.g., ensuring that writes prior to `sync` or `fsync` are durably stored on disk).

3. **Good disk throughput.** 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. Directories are represented in memory with a concurrent hash table. When Linux updates a directory it also propagates these changes to the in-memory ext4 physical log, which is later flushed to disk. This is essential to ensure crash safety, but can cause two commutative directory operations to contend for the same disk block in the in-memory ext4 physical 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 [9], the two `creates` commute and should be conflict-free to allow for scalability. However, if they happen to update the same disk block, they are not conflict-free and can limit scalability despite being commutative.

Our key insight is to *decouple* the in-memory file system from the on-disk file system. To enable decoupling, McoreFS separates the in-memory file system from the on-disk file system using an operation log (based on oplog [3]). The operation log consists of per-core logs of file system operations (e.g., `link`, `unlink`, `rename`). When `fsync` and `sync` are invoked, McoreFS sorts the operations in the operation log, and applies them to the on-disk file system. For example, McoreFS 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, McoreFS 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 to some degree (e.g., Linux has different in-memory and on-disk representations for directories), the operation log allows McoreFS to take this approach to its logical extreme: it enables two independent file systems: an in-memory file system and an on-disk file system, one tailored to achieve goal 1 (scalability) and the other tailored for goal 3 (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. On the other hand, the on-disk file system can choose data structures that allow for good disk throughput. The on-disk file system can even re-use an existing on-disk format.

To achieve goal 2 (crash safety), this thesis puts forward a set of principles that argue for what minimal durability guarantees POSIX must provide, and uses them to articulate an informal specification for `fsync`. McoreFS implements this specification by time-stamping linearization points of in-memory operations, and merging the per-core logs with the operations sorted by the time-stamps. McoreFS ensures that the set of changes written to disk is small by tracking dependencies between operations in the merged log. At an invocation of `fsync`, McoreFS applies only operations that relate to the file or directory being `fsync`ed, and absorbs operations that cancel each other out.

We have implemented McoreFS by modifying sv6 [9], which does not provide crash safety. We adopted the in-memory data structures from sv6 to implement the in-memory file system, but extended it with an on-disk file system from xv6 [11] using the decoupling approach. The implementation does not support all of the file system operations that, say, Linux supports (such as `sendfile`), but McoreFS does support complex operations such as `rename` across directories.

Experiments with McoreFS on Commuter [9] demonstrate that McoreFS's multicore scalability for commutative operations is as good as that of sv6 while providing crash safety. Experimental results also indicate that McoreFS achieves better scalability than the Linux ext4 [26] file system for in-memory workloads, while providing similar performance when accessing disk. McoreFS is conflict free in 99.2% of the commutative test cases Commuter generates, while Linux is conflict free for only 70% of them. Furthermore, experiments demonstrate that McoreFS achieves good disk performance.

The main contributions of the thesis 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.
- A set of principles for minimal durability guarantees provided by a file system, along with an informal specification of `fsync` that follows these principles.
- Techniques based on timestamping linearization points that ensure crash safety and high disk performance.
- An implementation of the above design and techniques in a McoreFS prototype.
- An evaluation of our McoreFS prototype that confirms that McoreFS achieves good scalability and performance.

The rest of the thesis is organized as follows. §2 describes related work, §3 introduces three principles for the design of `fsync`, §4 describes the design of McoreFS's operation log, §5 summarizes McoreFS's implementation, §6 presents experimental results, and §7 concludes.

Chapter 2

Related work

The main contribution of McoreFS 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 McoreFS's separation to previous designs, McoreFS's memory file system to other in-memory file systems, and McoreFS's disk file system to previous durable file systems.

File system scalability. McoreFS adopts its in-memory file system from sv6 [9]. 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 support writing data to durable storage. McoreFS's primary contribution over sv6's in-memory file system is showing how to combine multicore scalability with durability. To do so, McoreFS extends the in-memory file system using OpLog [3] to track linearization points, and adds an on-disk file system using an operation log sorted by the timestamps of linearization points.

Many existing file systems, including Linux ext4, suffer from multicore scalability bottlenecks [2, 9, 28], 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 [12]. This improves scalability for particular workloads and hardware configurations, but fails to achieve McoreFS'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 file system operations. For example, directories are often represented in memory differently than on disk to allow for higher performance and parallelism. Linux's dcache [10, 27], 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 on-disk data structures. This lack of decoupling causes unnecessary cache-line movement and limits scalability.

ReconFS [25] 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 McoreFS'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 [5]. However, these designs do not use logs designed for multicore scalability.

McoreFS implements the log that separates the in-memory file system from the on-disk file system using an oplog [3]. Oplog allows cores to append to per-core logs without any interactions with other cores. McoreFS extends oplog's design to sort operations by timestamps of linearization points of file system operations to ensure crash consistency.

Applying distributed techniques to multicore file systems. Hare is a scalable in-memory file system for multicore machines without cache coherence [16]. It does not provide persistence and poses this as an open research problem. McoreFS's decoupling approach is likely to be a good match for Hare too. One could use the Hare file system as the in-memory file system and McoreFS's on-disk file system for persistence. The main challenge will be to implement the operation log on a non-cache-coherent machine, in particular the merge operation.

SpanFS [20] 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 not to access the same file or directory from multiple cores. In contrast, McoreFS does not require the application developer to think about partitioning of files or directories; McoreFS provides scalability for commutative operations, even if they happen to modify the same file or directory (e.g., a mail server delivering messages into a shared spool directory on multiple cores). In addition, SpanFS flushes all outstanding journal entries upon `fsync`, and invokes a two-phase commit protocol when multiple cores are involved. In contrast, McoreFS computes a minimal set of disk writes to flush for each `fsync` using operational dependency tracking, which allows `fsync` to both write less data to disk and to incur less contention.

On-disk file systems. McoreFS's on-disk file system uses a simple design based on xv6 [11]. It runs file system updates inside of a transaction as many previous file systems have done [7, 17], has a physical log for crash recovery of metadata file system operations (but less sophisticated than, say, ext4's design [26]), and implements file system data structures on disk in the same style as the early version of Unix [31].

Because of the complete decoupling of the in-memory file system and on-disk file system, McoreFS could be modified to use ext4's disk format, and adopt many of its techniques to support bigger files, more files, and better disk performance. Similarly, McoreFS's on-disk file system could be changed to use other ordering techniques than transactions; for example, it could use soft updates [15], a patch-based approach [14], or backpointer-based consistency [6].

McoreFS'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 [19], BTRFS [32], TABLEFS [30], NoFS [6], and XFS [34] to increase disk performance.

McoreFS'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 buffer journal [23] or ReconFS to minimize intensive metadata writeback and scattered small updates [25].

Chapter 3

Durability semantics

A challenge in ensuring that McoreFS is both crash-safe and high-performance lies in determining the minimal set of requirements that McoreFS must satisfy to achieve crash safety. For instance, it is easy to provide a crash-safe file system by flushing all changes to disk immediately, but this achieves poor performance. Modern file systems implement different strategies for deferring disk writes, but do not agree on a consistent set of crash consistency guarantees [29, 35, 36]; for instance, Linux provides different guarantees depending on the file system type and mount options.

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 [29]. On the other hand, an interface that provides cleaner crash safety properties may give up performance that sophisticated application programmers want. In this thesis, McoreFS aims to provide high performance, targeting programmers who carefully manage the number and placement of calls to sync and fsync in their applications, with a view to obtain the best performance.

McoreFS's semantics for fsync build on three principles. The first principle is that *fsync's effects should be 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 [29, 35].

The local principle 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 principle, as follows.

The second principle for McoreFS's fsync semantics is that *the file system should be able to 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 principle 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.

Our third principle aims to resolve this anomaly, by requiring that *rename should not cause a file or directory to be lost*. Taking McoreFS's three principles together, the final semantics of fsync are that it flushes changes to the file or directory being fsynced, and, in the case of fsync on a directory, it also flushes changes to other directories where files may have been renamed to.

These semantics are sufficient to build crash-safe applications, since they are similar to the semantics provided by some Linux file systems (e.g., applications already assume that they need to call fsync on the parent directory to ensure crash safety for a newly created file). While these semantics require for careful programming at the application level, they also allow for the file system developer to provide high performance to applications (since the file system need not flush changes unrelated to the file or directory being fsynced, and those other changes can be deferred).

Chapter 4

Design overview

McoreFS 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 would allow commutative in-memory operations to scale, while the on-disk file system deals with constructing disk blocks and syncing 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 [3], which logs operations such as `link`, `unlink`, `rename`, etc. Although an operation log is designed for update-heavy workloads, it is a good fit for McoreFS's design because MemFS handles all of the reads, and the log is consulted when flushing changes to DiskFS.

McoreFS's design faces two challenges—performance and correctness—and the rest of this section describes how McoreFS achieves both. We explicitly mark the aspects of the design related to either performance or correctness with **[P]** and **[C]** respectively. Performance optimizations need to achieve good performance for workloads that matter to real applications, while correctness guarantees must be upheld regardless of what operations the application issues.

4.1 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 difficult, because MemFS uses lock-free data structures, which in turn makes it difficult to control the order in which different operations run. 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 4-1. 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)` and thread T2 issues `rename(a, c)`. Both these threads run concurrently. 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.

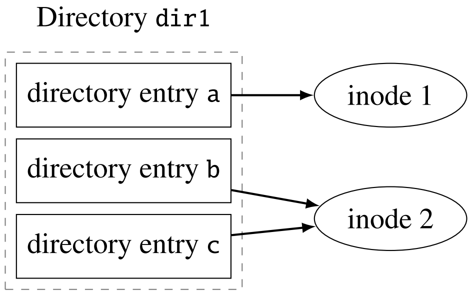

Figure 4-1: Data structures involved in a concurrent execution of `rename(a,c)` and `rename(b,c)`. Arrows depict a directory entry referring to an inode. There are three directory entries in a single directory, two of which are hard links to the same inode.

Assume that T1 goes first. To achieve conflict-freedom for commutative operations, 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 to decrement the inode's reference count, which it can do without holding a lock by using a distributed reference counter like Refcache [8]). The only lock T1 acquires is on `b`, since it does not touch `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, McoreFS 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 [3: §6.1], which our file system design 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, our design orders in-memory operations by requiring the in-memory file system to be *linearizable*. Moreover, our design 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.

Since MemFS uses lock-free data structures, determining the linearization points is challenging. The linearization points are in regions made up of one or more atomic instructions. A simple way to get a timestamp at the linearization point would be to protect this region with a lock and read the timestamp counter anywhere within the locked region. For operations that update part of a data structure under a per-core lock this scheme works well.

For read operations, obtaining the linearization point is complicated by the fact that, to achieve scalability, McoreFS does not use read locks. For example, T1 from Figure 4-1 does not take a read lock on `c` to achieve scalability, but we do have to order T1 and T2 correctly.

To solve this problem, MemFS protects such read operations with a seqlock [22: §6]. 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.

To determine the timestamp of the linearization point for a read operation, 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 McoreFS 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.

4.2 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. McoreFS'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 McoreFS is that the merge is tricky. 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 4-2.

Figure 4-2 shows the execution of two operations on two different cores: *op1* on core 1 and *op2* on core 2. LP1 is the linearization point of *op1*, and LP2 of *op2*. If the per-core logs happen to be merged 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. McoreFS achieves this by tracking, for each core, whether an operation is currently execut-

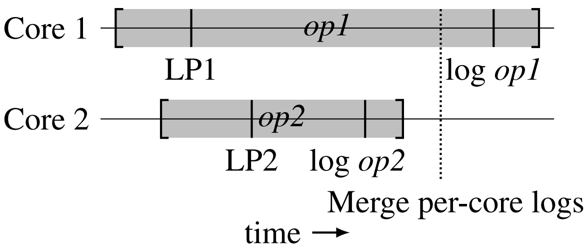

Figure 4-2: Two concurrent operations, *op1* and *op2*, running on cores 1 and 2 respectively. The grayed-out box denotes the start and end time of an operation. LP denotes the linearization point of each operation. "log" denotes the time at which each operation is inserted into that core's log. The dotted line indicates the time at which the per-core logs are merged.

ing, and if so, what its starting timestamp is. To merge, MCOREFS 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` has to 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 inode (file or directory). Per-inode logs allow MCOREFS to merge the operations for that inode on an `fsync` without conflicting with concurrent operations on other inodes. MCOREFS uses oplog's lazy allocation of per-core logs for efficiency [3], so that MCOREFS does not need to allocate a large number of logs that are never used.

4.3 Flushing an operation log [P, C]

MCOREFS's per-inode 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, MCOREFS should perform absorption when flushing multiple related operations to disk **[P]**. Second, as we discussed in §3, MCOREFS's `fsync` specification requires special handling of cross-directory `rename` to avoid losing files **[C]**. Finally, McoreFS 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, McoreFS 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, McoreFS may need to flush these operations in several batches. In this case, McoreFS 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, so McoreFS 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, reducing the amount of disk I/O needed for `fsync`. One subtle issue is that some process can still hold an open file descriptor for the non-existent 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 McoreFS 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, McoreFS implements dependency tracking. Specifically, when McoreFS 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 McoreFS to access the other directory's operation log as well. As an optimization, MCOREFS 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.

Internal consistency [C]. In addition to providing application-level consistency, MCOREFS must also guarantee that its own data structures are intact after a crash. There are three cases that MCOREFS must consider to maintain crash safety for its internal data structures.

First, MCOREFS 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. MCOREFS must ensure that the file is flushed first, before a link to it is created. MCOREFS achieves this using dependencies, much as with cross-directory renames; when flushing a directory, MCOREFS also flushes the allocation of new files linked into that directory.

Second, MCOREFS 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. MCOREFS 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, MCOREFS 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 showed in Figure 4-3. With all of the above checks (including `rename` dependencies), if an application issues the two `mv` commands shown in Figure 4-3 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 4-3 illustrates, because changes to `B` (namely, moving `C` from `B` to the root directory) are not flushed.

To avoid directory loops on disk, MCOREFS 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

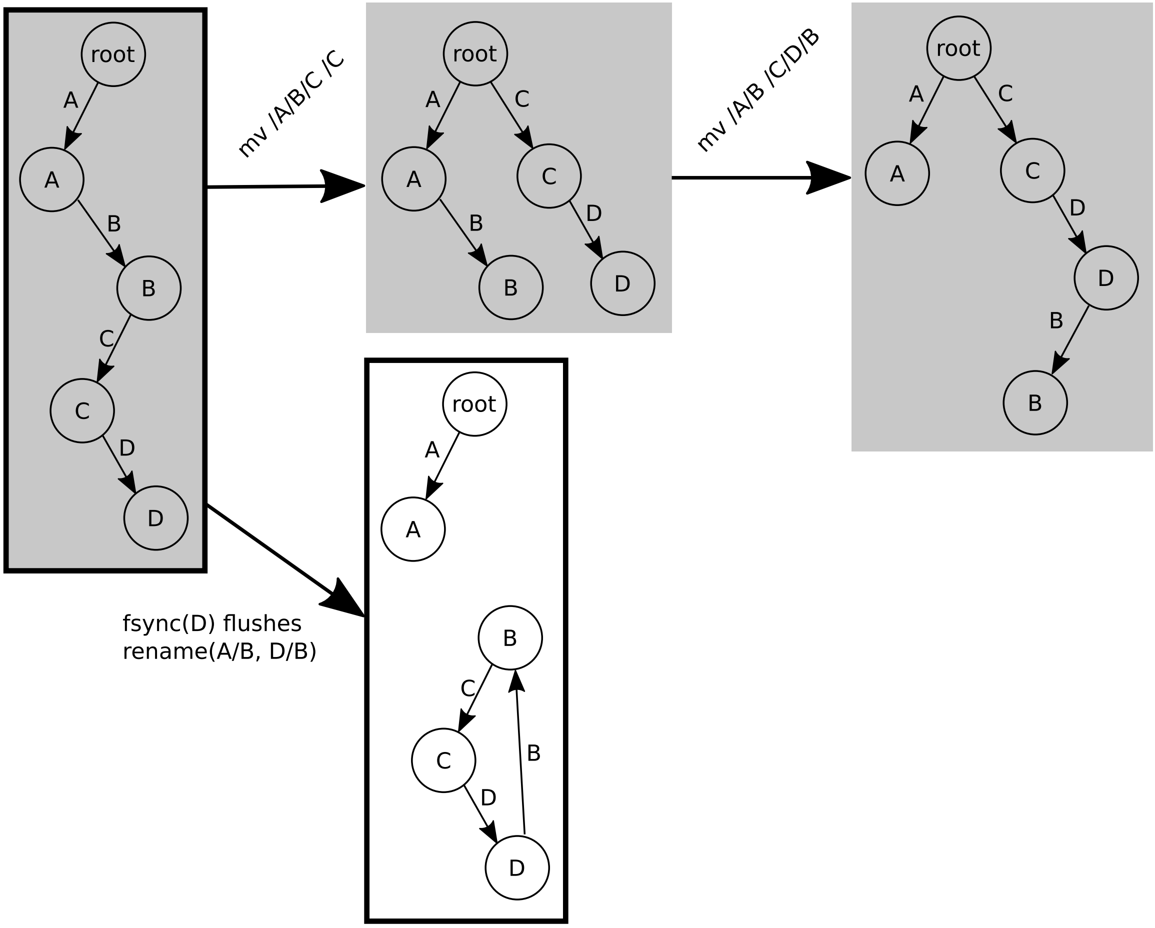

Figure 4-3: A sequence of operations that leads to a directory cycle (B-C-D-B) on disk with a naïve implementation of `fsync`. The state of the file system evolves as an application issues system calls. Large arrows show the application's system calls. Large rectangles represent the logical state of 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.

on disk in combination with the next two rules. Furthermore, this lock is rarely contended, since we expect applications to not rename subdirectories between parent directories in their critical path. Second, when a subdirectory is moved into a parent directory `d`, McoreFS records the path from `d` to the root of the file system. This list of ancestors is well defined, because McoreFS 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`, McoreFS first flushes to disk any changes to the ancestors of `d` as recorded by that `rename` operation. As an optimization, McoreFS flushes ancestor changes only up to the timestamp of the rename operation. An argument about this algorithm's correctness is available in appendix §A.

4.4 Multiple disks and journals [P]

On a computer with multiple disks or multiple I/O queues to a single disk (as in NVMe), MCOREFS 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, MCOREFS stripes the DISKFS data across all of the physical disks in the system. MCOREFS also allocates a separate on-disk journal for every core, to take advantage of multiple I/O queues, and spreads these journals across cores 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, MCOREFS 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.

Crash-safety [C]. 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, MCOREFS uses timestamps to defer flushing dependent journal entries to disk. Specifically, MCOREFS 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 McoreFS 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 McoreFS finishes flushing a journal entry to disk, it updates an in-memory timestamp to reflect that this journal's timestamp has made it to disk.

Applying transactions before a crash. During normal operations, for performance reasons, a call to `fsync` only commits transactions to the on-disk journals and does not apply them to the on-disk file system immediately. Applying the committed transactions from a journal is deferred until that journal gets full. When a subsequent `fsync` finds that there is no space left in a journal to commit more transactions, it applies all the existing transactions from that journal and recovers all the space from that journal. However, this apply must be done with care, so as to honor the dependencies that might exist between transactions committed via different journals. We reuse the per-block timestamp-based dependency tracking scheme outlined above to apply committed transactions in the correct order to the on-disk file system.

Applying transactions after a crash. Applying dependent transactions from the various per-core journals in the correct order can be challenging when performing recovery after a crash, because the hash table that tracks transaction dependencies on a per-block basis would no longer be available (as it is maintained only in memory). To solve this problem, we timestamp every transaction at the time of committing it to an on-disk journal, and persist this timestamp in the journal entry representing that transaction. Since the ordering between dependent transactions would have been already resolved at the time of commit, these timestamps will compactly reflect the correct ordering between transactions. This simplifies applying dependent transactions during crash-recovery, because we can simply read all the transactions from all the journals and apply them to the on-disk file system after sorting them by their commit-timestamps.

Recovery with partially applied journals. A further challenge arises when performing recovery after a reboot, if some of the journals contain a mix of transactions, some of which have been already applied to the on-disk file system and others that have been merely committed to the journal. This could happen for example, when transactions are applied from the journals during normal operations to make space for committing newer transactions. Re-applying transactions that have been already applied is safe during crash-recovery only if all the transactions they depend on are also re-applied in the correct order. However, this could be problematic if some of the dependent transactions were stored in journals that were wiped out as part of making space for newer transactions, and hence are no longer available after reboot. To enable correct file system recovery in such scenarios, we note down the timestamp of the latest transaction that was successfully applied from a given journal, in that journal's header, and sync the header to disk every time it is updated as part of transaction apply. This helps the recovery code skip over applied transactions (if any) in that journal and apply only those transactions that were only committed to the journal before the crash. Committed transactions are not erased from the journals until they are successfully applied to the on-disk file system, and hence all the dependencies among committed transactions will remain available in the journals after reboot.

4.5 Discussion

The design described above achieves McoreFS's three goals.

First, McoreFS 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, McoreFS ensures crash safety through dependency tracking and loop avoidance protocols described above. Third, McoreFS 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`, McoreFS 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 McoreFS achieves high throughput for `fsync` (and McoreFS flushes less data than the ext4 file system on Linux).

Finally, McoreFS achieves good disk throughput through its optimizations (such as absorption and group commit). McoreFS 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, McoreFS takes advantage of multiple disks to further improve disk throughput.

Chapter 5

Implementation

We implemented MCOREFS in sv6, a research operating system whose design is centered around the Scalable Commutativity Rule [9]. Previously sv6 consisted of an in-memory file system that MEMFS reuses, modifying it to interact with the operation log. MCOREFS augments sv6's design with a disk file system DISKFS (that is based on the xv6 [11] file system), and an operation log. The numbers of lines of code change involved in implementing each component of MCOREFS are shown in Figure 5-1. MCOREFS is open-source and available for download.

MCOREFS component	**Lines of C++ code**
MEMFS (§5.1)	2,445
DISKFS (§5.2)	2,320
MEMFS-DISKFS interface	4,177

Figure 5-1: Lines of C++ code for each component of MCOREFS.

5.1 MEMFS

Decoupling the in-memory and the on-disk file systems allows MEMFS to be designed for multicore concurrency; our prototype's MEMFS is based on the in-memory file system from sv6. MEMFS represents directories using chained hash tables that map file/directory names to inode numbers. Each bucket in the hash table has its own lock, allowing commutative operations to remain conflict-free.

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 [8]. MemFS creates the in-memory directory tree on demand, reading in components from the disk as they are needed. We refer to inodes, files and directories in MemFS as mnodes, mfiles and mdirs.

To allow for concurrent creates to be scalable, 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.

McoreFS maintains a hashmap from mnode numbers to inode numbers and vice versa for fast lookup. McoreFS creates this hashmap 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 McoreFS adds entries to the hashmap when an inode is created on disk corresponding to a new mnode in memory.

McoreFS does not update the hashmap 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. Although this violates the POSIX requirement that a file can be uniquely identified by its device and inode number as reported by `stat`, few applications rely on this (qmail being the only example we know of).

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 in the operation logs selectively.

By omitting certain operations from the log, McoreFS 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 §4.3. 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: oplogged and non-oplogged. All directory state is oplogged metadata, and a file's link count, which is affected by directory operations, is also oplogged metadata. However, other file metadata, such as the file's length, its modification times, etc, is not oplogged. 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-oplogged file data and metadata.

When flushing a file from MemFS to DiskFS, McoreFS must combine the oplogged and non-oplogged metadata of the flushed mnode. It updates the oplogged metadata of the mnode by merging that mnode's oplog. The non-oplogged 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 journals all metadata (both metadata that was oplogged as well as non-oplogged file metadata), but writes file data directly to the file's blocks, which is similar to how Linux ext4 writes data in "ordered" mode.

When MemFS flushes an oplogged change to DiskFS that involves multiple mnodes (such as a cross-directory `rename`), McoreFS 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.

5.2 DiskFS

The implementation of DiskFS depends on the disk file system format used (e.g., ext3 [4], ext4 [26], etc.). McoreFS's DiskFS is based on the xv6 [11] 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.

5.3 Limitations

McoreFS does not support the full set of file system calls in POSIX, such as `sendfile` and `splice`. However, McoreFS does support the major operations needed by applications, and we believe that supporting the rest of the POSIX operations would not affect McoreFS's design. Our prototype does not implement background flushing, and flushes only when an application calls `fsync` or `sync`.

Chapter 6

Evaluation

We ran experiments to try to answer the following questions:

- Does McoreFS achieve multicore scalability? (§6.2, §6.3)
- Does durability reduce multicore scalability in McoreFS? (§6.4)
- Does McoreFS achieve good disk throughput? (§6.5)
- Does McoreFS provide crash safety to applications? (§6.6)
- What overheads are introduced by McoreFS's split of MemFS and DiskFS? (§6.7)

The evaluation focuses on the performance aspects of McoreFS's design, which can be evaluated empirically, rather than on the correctness aspects, which are always ensured by McoreFS's design.

6.1 Methodology

To measure scalability of McoreFS, we primarily rely on Commuter [9] 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.

To confirm the scalability results reported by Commuter, we also experimentally evaluate the scalability of McoreFS 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 McoreFS, as well as measuring McoreFS'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 256GB SSDs.

To provide a baseline for McoreFS'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 [28].

6.2 Does McoreFS achieve multicore scalability?

To evaluate McoreFS'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 Commutativity Rule [9]. 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 McoreFS.

Figure 6-1 shows results obtained by running Commuter with McoreFS. 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.

McoreFS 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

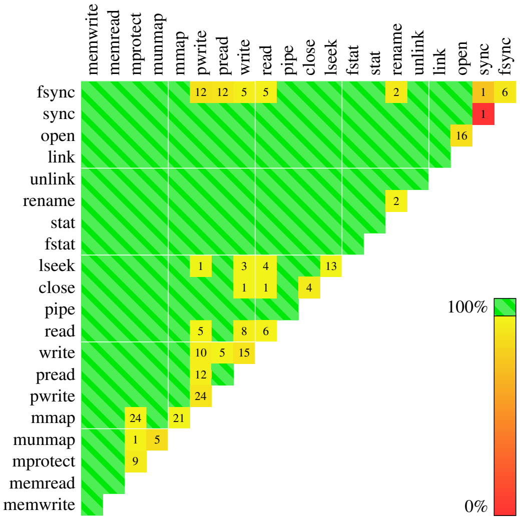

Figure 6-1: Conflict-freedom of commutative operations in MCOREFS. Out of 31,551 total test cases generated, 31,317 (99.2%) were conflict-free.

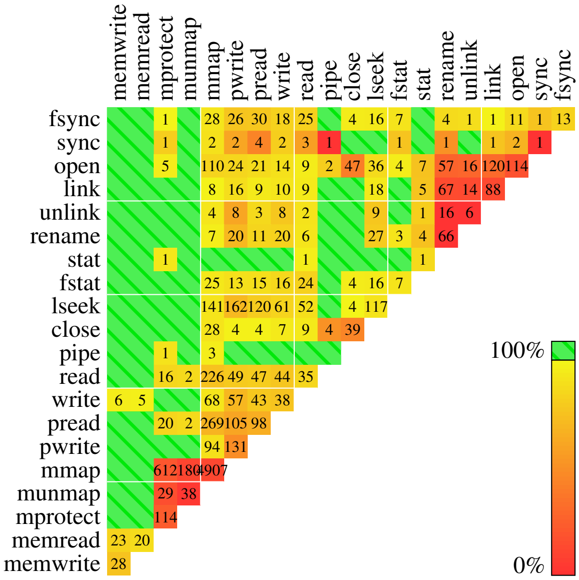

Figure 6-2: Conflict-freedom of commutative operations in the Linux kernel using an ext4 file system. Out of 31,539 total test cases generated, 22,096 (70%) were conflict-free.

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

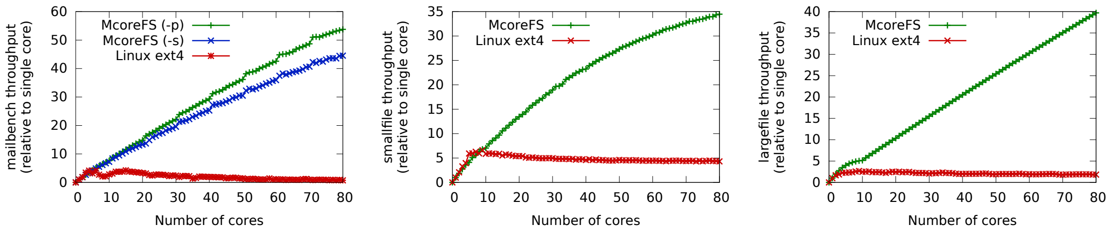

Figure 6-3: Throughput of the mailbench, smallfile, and largefile workloads respectively, for both McoreFS and Linux ext4 in a RAM disk configuration.

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.

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. So 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 McoreFS's heatmap, we also ran Commuter on the Linux kernel with an ext4 file system.[1] The heatmap in Figure 6-2 shows the results obtained. Out of a total of 31,539 commutative test cases, the heatmap shows 9,443 of them (30%)

[1] We ran Commuter on Linux kernel version v3.16 (from 2014) because the authors of Commuter have not ported their Linux changes 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 [28].

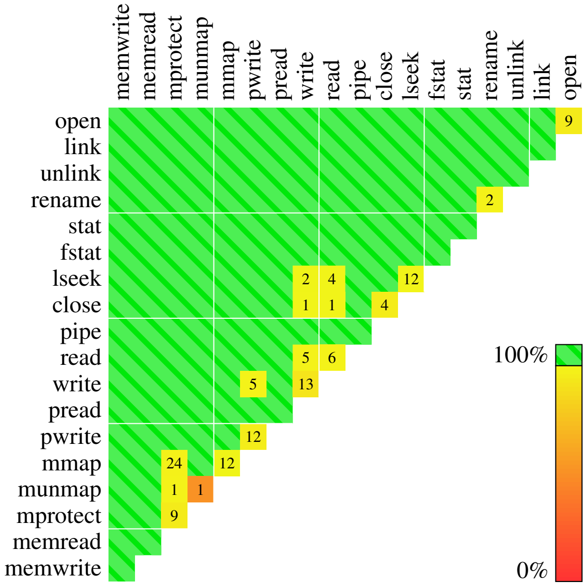

Figure 6-4: Conflict-freedom between commutative operations in sv6 with only an in-memory file system. Out of 13,664 total test cases generated, 13,528 (99%) were conflict-free.

conflicting in the Linux kernel. Many of these can be attributed to the fact that the in-memory representation of the file system is very closely coupled with the disk representation. Commutative operations end up conflicting in the page cache layer. These results are in line with the bottlenecks uncovered in prior work [2, 28]. 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 McoreFS's split design avoids.

6.3 Empirical scalability

To confirm that Commuter's results translate into scalability for actual workloads on real hardware, we compared the performance of three workloads on McoreFS to their performance on Linux ext4; unfortunately, porting applications to sv6 is difficult because sv6 is not a full-featured OS.

Disk	Benchmark	McoreFS	Linux ext4
RAM disk	largefile	331 MB/sec	378 MB/sec
	smallfile	7151 files/sec	3553 files/sec
	mailbench-p	641 msg/sec	675 msg/sec
SSD	largefile	180 MB/sec	180 MB/sec
	smallfile	364 files/sec	277 files/sec
	mailbench-p	61 msg/sec	66 msg/sec
HDD	largefile	83 MB/sec	92 MB/sec
	smallfile	51 files/sec	27 files/sec
	mailbench-p	9 msg/sec	9 msg/sec

Figure 6-5: Performance of workloads on McoreFS and Linux ext4 on a single disk and a single core.

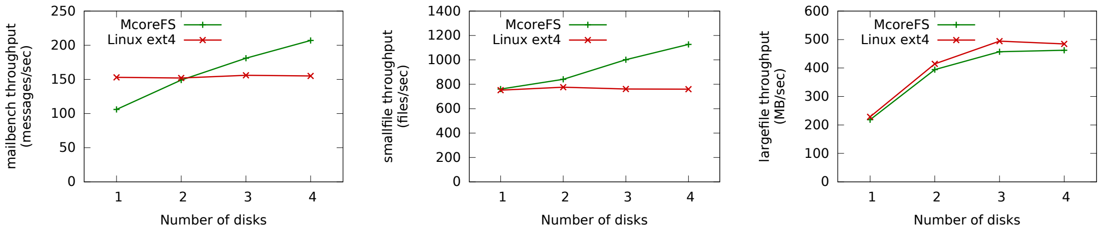

Figure 6-6: Throughput of the mailbench-p, smallfile, and largefile workloads respectively, for both McoreFS and Linux ext4, using 4 CPU cores. The x-axis indicates the number of SSDs striped together.

mailbench. One of our workloads is mailbench, a qmail-like mail server benchmark from sv6 [9]. 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**).

largefile. Inspired by the LFS benchmarks [33], 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.

To avoid the disk bottleneck, we ran these experiments on a RAM disk. Both MCOREFS and ext4 still performed journaling, flushing, etc, as they would on a real disk. §6.5 presents the performance of both file systems with real disks.

Figure 6-3 shows the results. MCOREFS scales well for all three workloads, achieving approximately 40× performance at 80 cores. MCOREFS 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. mailbench-s scales to a lesser extent than mailbench-p at high core counts due to contention on the shared mailboxes, not due to any MCOREFS bottlenecks; mailbench-p continues to scale[2]. Linux ext4 fails to scale for all three workloads, achieving no more than 7× the performance of a single core, and collapsing as the number of cores grows beyond 10. We run only the mailbench-p variant on Linux since it is more scalable.

6.4 Does durability reduce multicore scalability?

To evaluate the scalability impact of MCOREFS's approach for achieving durability, we compare the results of COMMUTER on MCOREFS to the results of running COMMUTER on the original sv6 in-memory file system, on which MEMFS is based. Figure 6-4 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 6-1, we see that MCOREFS introduces no new conflicts between existing syscalls.

[2] A scalability bottleneck was observed in the virtual memory subsystem when running mailbench, caused by frequent calls to sync the reverse-map entries in the Oplog. We alleviated this bottleneck by syncing reverse-map entries in batches of 1 million (as opposed to batches of 100).

6.5 Disk performance

Single disk, single core. To evaluate whether McoreFS can achieve good disk throughput, we compare the performance of our workloads running on McoreFS to their performance on Linux ext4. Figure 6-5 shows the results, running with a single disk and a single CPU core. McoreFS achieves comparable performance to Linux ext4 in all cases. For the smallfile microbenchmark, McoreFS is faster because McoreFS'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.

Multiple disks, 4 cores. Since the disk is a significant bottleneck for both McoreFS and ext4, we also investigated whether McoreFS can achieve higher disk throughput with additional physical disks. In this experiment, we striped several SSDs together, and ran either McoreFS or Linux ext4 on top. To provide sufficient parallelism to take advantage of multiple disks, we ran the workload using 4 CPU cores. Figure 6-6 shows the results for our workloads. McoreFS achieves similar throughput to Linux ext4. For the largefile workload, both McoreFS and Linux cannot obtain more throughput because the SATA controller is saturated.

McoreFS scales better than Linux for mailbench 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. McoreFS is aware of multiple disks, and its `fsync` issues a barrier only to the disks that have been written to by that `fsync`.

6.6 Crash safety

To evaluate the crash safety properties of McoreFS, we performed a series of experiments to observe whether McoreFS recovers correctly from crashes, under scenarios governed by the crash model below.

Crash model. We run the experiments on MCOREFS that has been configured to use an SSD as the storage device. Crashes are caused by triggering an external machine reset, which power cycles the machine. Given this setup, the system could be in any of the following crash states in the event of a crash:

- All the disk writes ever issued by MCOREFS had durably reached the disk before the crash.
- A subset of the disk writes issued by MCOREFS had durably reached the disk before the crash. This includes scenarios where the disk controller had signaled completion of some disk writes issued by MCOREFS, but those writes had only been recorded in the disk's volatile buffer, and they had not actually made it to the durable storage medium on the disk before the crash.

Internal consistency tests. We ran a number of tests to observe the behavior of MCOREFS in scenarios involving failures at particularly tricky crash points in the file system code. Each test is run in verbose mode where it prints its progress after every successful call to `fsync`. This enables us to note down the state of the files and directories manipulated by the test, at each progress point, which we expect to see persisted intact on DISKFS even in the event of a crash. To verify the crash safety properties of MCOREFS, we trigger a crash (via machine reset) after a randomly chosen time interval has elapsed since the start of the test. We then compare the state of the file system after recovery, with the state corresponding to the last progress point that was printed by the program before the crash. The test is deemed to have passed if the two states are consistent with each other. The tests themselves are described below.

- Create a large number of files in a shared directory and `fsync` the newly created files and the containing directory from multiple cores, thereby causing transaction dependencies that span across multiple cores and journals. This helps us verify the correctness of the dependency tracking and resolution code in the most challenging scenarios, both when committing transactions as well as when applying transactions from the journals during crash recovery.

- Create a large number of dependent transactions that span multiple cores and journals as above, but randomize the number of transactions flushed from each core so as to overflow some but not all of the journals. This test helps us verify that McoreFS recovers correctly from crashes even when it encounters transaction dependencies that span across partially applied journals.
- Perform a long series of cross-directory file renames that form a chain of dependencies across different directories, and invoke `fsync` on these directories. This helps us evaluate whether McoreFS preserves the atomicity of file renames and applies them in the correct order during crash recovery.
- Perform a series of sub-directory renames across different source and destination directories, and invoke `fsync` on some of these directories to create scenarios that can potentially result in directory loops in the on-disk file system. This helps us verify the correctness of our loop avoidance protocols in the event of crashes.

Our experiments confirmed that McoreFS recovers correctly from crashes in all of these scenarios, under the given crash model described above.

Application-level crash consistency tests. To evaluate whether the crash safety guarantees of McoreFS helps preserve the semantics of applications running on top of it, we ran crash recovery tests by running mailbench-s on multiple cores. We employed a scheme very similar to the one we used for the internal consistency tests (i.e., using verbose prints after every successful call to `fsync`, triggering crashes after random timeouts after the start of the test, and then comparing the state of the files and directories manipulated by the benchmark before crash and after crash recovery) to verify whether McoreFS provides crash consistency to mailbench.

To deliver a mail message, an instance of mailbench creates the mail text and spawns a mail-enqueue process to queue the message durably to a per-core spool directory and communicate the recipient and the message identifier to a mail-manager instance that is running on that core. This involves one cross-directory file rename, and two file `fsync`s and two directory `fsync`s. Upon receiving a mail transfer request, the mail-manager spawns a mail-deliver process to durably forward the mail message to the recipient's mail directory. This involves another cross-directory file rename and a file fsync as well as a directory fsync.

We run mailbench-s on multiple cores with per-core spool directories and a single user mailbox, so as to create a large number of dependencies (spanning multiple cores and journals) across transactions that try to update the shared user's mail directory. This is the most challenging scenario for crash recovery in the context of mailbench, as this calls for careful dependency resolution when applying transactions from multiple journals to the on-disk file system after a crash.

Incorrect file system recovery can result in a variety of inconsistencies for mailbench, ranging from delivered mail messages missing after a crash, dangling directory entries (i.e., links pointing to uninitialized inodes), missing or corrupted file contents among the delivered mail messages, and rename operations that appear to violate atomicity (such as losing files involved in the rename, or finding duplicates corresponding to both the source and the destination of the rename).

In our experiments, we were able to verify that McoreFS recovers all the files and directories managed by mailbench correctly after crashes under our crash model, and never exposes any of the possible inconsistencies outlined above.

6.7 Overhead of splitting MemFS and DiskFS

The main worry about using McoreFS's split-design, with an in-memory MemFS file system and an on-disk DiskFS file system is that McoreFS might incur higher memory overhead because it has to keep the operation log in memory and because metadata blocks (those containing bitmaps, inodes, and directory blocks) may be kept in memory twice: once in MemFS and once in DiskFS. However, file data blocks are not stored twice.

To evaluate how severe this overhead is, we measured the memory usage for the largefile benchmark. During the benchmark's execution, memory usage is similar to Linux. When the application invokes fsync, memory usage of the file system increases by the size of the file being fsynced (i.e., 100 MBytes for the largefile benchmark), because McoreFS creates a copy of the file's blocks in memory in preparation for writing them to disk. We could avoid this by using copy-on-write, but have not implemented this in our prototype. Once the data is written to disk, the DiskFS buffer cache discards copies of blocks for file data, bringing the memory usage back in line with Linux. A more sophisticated design could avoid this double-buffering by allowing DiskFS to keep pointers directly into user-accessible pages from MemFS.

Chapter 7

Conclusion

It is a challenge to achieve multicore scalability, crash consistency and good disk throughput in a file system. This thesis 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 thesis introduces an operation log that extends oplog [3] 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 that 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, MCOREFS, that was built on the existing sv6 kernel, and analyzed the implementation using COMMUTER. We experimentally evaluated the multicore scalability and disk performance of MCOREFS by running benchmarks on an 80-core machine, in different configurations including a RAM disk, SSDs and an HDD. We evaluated the crash safety properties of MCOREFS by using several consistency tests and an application benchmark. The results show that the implementation of MCOREFS achieves good multicore scalability and disk throughput, while providing crash consistency to applications.

Appendix A

Correctness of the directory loop avoidance algorithm in MCOREFS

MCOREFS employs a number of techniques to ensure the internal consistency of the file system. However, MCOREFS's split design with a separate MEMFS and DISKFS, coupled with some of the optimizations it uses to limit `fsync`'s writeback, poses challenges to achieve the consistency goal, and notable among them is the task of preventing orphaned directory loops in MEMFS and DISKFS. The problem was explained in chapter §4 in section §4.3, accompanied by an outline of the solution. This appendix describes an argument for the correctness of the directory loop avoidance algorithm in MCOREFS.

It is helpful to recall some of the salient properties of MCOREFS that are central to the correctness argument: MCOREFS maintains a set of per-core operation logs for every inode in the system. Each operation in an operation log is accompanied by a timestamp, which represents the linearization point of that operation when it was performed in MEMFS. The operations in a given operation log are flushed to disk strictly in the increasing order of their timestamps.

We describe the notation and conventions used in the argument below:

- For any node *d* (representing a file or directory) in the file system tree, *parent(d)* denotes its containing directory.
- The set of ancestors of any node *d* is defined as:
  ancestors(d) := *{d}*, if *d* is the root directory.
  ancestors(d) := *parent(d)* $\bigcup$ *ancestors(parent(d))* otherwise.
- For any node *d*, the notation MemFS.*ancestors(d) @ t* represents *ancestors(d)* as seen in MemFS at timestamp *t*.

Definition: Loop in a file system.

A loop is said to exist in a file system, if:
$\exists$ some timestamp *t*, $\exists$ some node *d* in the file system ($d \neq$ root directory), such that
$d \in$ *ancestors(d) @ t*

Theorem 1: MemFS is loop-free.

Claim: MemFS never contains loops in the file system hierarchy, which can be expressed as:

$\forall\, t$, $\forall\, d$ ($d \neq$ root directory), $d \notin$ MemFS.*ancestors(d) @ t*

Correctness argument for Theorem 1. We argue this theorem by deriving a contradiction. Suppose there exists a loop in MemFS at timestamp t_{loop}. We focus on the very first instance of such a loop, implying that there is no loop in MemFS at any timestamp $t < t_{loop}$. Now let us consider the set of all file system operations in MemFS that can potentially cause the loop at t_{loop}.

Observation 1.1. Neither file operations nor the `mkdir` and `rmdir` directory operations can cause loops in a file system.

Explanation. We observe that the property of a loop (as described above) is common to all the nodes that belong to the loop. In particular, every node in the loop has an ancestor, or equivalently, every node in the loop has at least one child node. This implies that leaf nodes (nodes that don't have any children) cannot be part of any loop.

Therefore, we can conclude that any operation that adds, deletes or modifies only leaf nodes in the file system cannot cause a loop. All file operations belong to this category, as do `mkdir` and `rmdir`. □

Eliminating several classes of operations using observation 1.1 leaves us with directory renames as the only file system operations that can potentially cause the loop at t_{loop}. In MemFS, directory renames are performed while holding a global, file system-wide lock, which means that at most one directory rename can occur in MemFS at a given timestamp t.

Now consider the directory rename that could have caused the loop in MemFS at t_{loop}. It can be represented as:

`rename(src/target, dst/target)` @ t_{loop}, where *src*, *dst* and *target* are all directories. It is easy to see that renaming a subdirectory within the same containing directory cannot cause a loop (by the same argument that we used in observation 1.1), and hence we can conclude that $src \neq dst$.

Let us recall our assumption about loops in MemFS:

$\forall\ d$ ($d \neq$ root directory), $d \notin$ MemFS.*ancestors(d)* @ t, for $t < t_{loop}$, and

$\exists\ d$ ($d \neq$ root directory), $d \in$ MemFS.*ancestors(d)* @ t_{loop}

This implies that, to cause the loop at t_{loop}, the loop-causing operation at t_{loop} must have altered the set of ancestors of some node d in the file system. Now consider the directory `rename` operation mentioned above. The only file system nodes it deals with are *src*, *dst* and *target* directories, and clearly it alters the set of ancestors of *target* (since we had argued that $src \neq dst$). It can be shown that if *target* is not part of a loop at the end of the `rename` (despite the change of ancestors), neither *src* nor *dst* could have become parts of any loops either. But by our assumption, there is a loop in MemFS at t_{loop}. So the `rename` must have caused *target* to be part of a loop, which can be represented as:

target $\in$ MemFS.*ancestors(target)* @ t_{loop}

This in turn implies that *dst* is also part of the same loop as *target*, because:

MemFS.*ancestors(target)* @ t_{loop} = $\{dst\} \bigcup$ MemFS.*ancestors(dst)* @ t_{loop}

Now let us consider the implicit change in ancestors of *dst* caused by the `rename` operation. If t_{rename_begin} denotes the timestamp at the start of that `rename`, and t_{rename_end} (= t_{loop}) denotes the timestamp at the end of the `rename`, then we know that:

MemFS.*ancestors(dst)* @ $t_{rename\_begin} \notin$ MemFS.*ancestors(dst)* @ t_{rename_begin}

MemFS.*ancestors(dst)* @ $t_{rename\_end} \in$ MemFS.*ancestors(dst)* @ t_{rename_end}, and

target $\in$ MemFS.*ancestors(dst)* @ t_{rename_end}

However, since the `rename` operation moves *target* into *dst*, making it a child directory of *dst*, it follows that *target* must have been an ancestor of *dst* at t_{rename_begin} itself. In other words, the `rename` operation at t_{loop} must have moved an ancestor directory (*target*) into one of its descendant directories (*dst*). But this is impossible because MemFS's loop avoidance algorithm explicitly checks (while holding the global file system-wide lock), for attempts to move an ancestor into a descendant, and fails such `rename` operations. This leaves us with no way to cause a loop in MemFS at t_{loop}, which completes our argument for theorem 1, by contradiction.

□

Observation 1.2. The only way to cause a loop in a file system hierarchy is to move an ancestor directory into one of its descendants.

Theorem 2: DiskFS is loop-free.

Claim: DiskFS never contains loops in the file system hierarchy, which can be expressed as:

$\forall$ *t*, $\forall$ *d* (*d* $\neq$ root directory), *d* $\notin$ DiskFS.*ancestors(d)* @ *t*

Proof of Theorem 2. By observation 1.1, neither file operations nor the `mkdir` and `rmdir` directory operations can cause loops in a file system tree. Since the argument of observation 1.1 assumed nothing about MemFS, this property is applicable to DiskFS too. Thus, only directory renames can potentially cause loops in DiskFS.

Once again, we argue this theorem by deriving a contradiction. Suppose there exists a loop in DiskFS at timestamp t_{loop}. We focus on the very first instance of such a loop, implying that there is no loop in DiskFS at any timestamp $t < t_{loop}$. Now consider the file system operation flushed to disk at t_{loop}, which caused the loop. It must have been a directory rename (based on our deduction above), of the form:

`rename(src/target, dst/target)`, which was performed in MemFS at $t_{rename} < t_{loop}$

Using a similar reasoning as we used for theorem 1, we can argue that flushing this operation causes *target* to be part of a loop in DiskFS: *target* $\in$ DiskFS.*ancestors(target)* @ t_{loop}, and further, *dst* also becomes part of the same loop at timestamp t_{loop}.

From observation 1.2, flushing this `rename` could have caused *target* to be part of a loop only if *target* was already an ancestor of *dst* in DiskFS, just before flushing the `rename` operation. Now we take a moment to make a few additional observations, to assist the argument for theorem 2.

Observation 2.1. If directory *B* is a child of directory *A* in the file system at some timestamp *t*, then the only way to make *B* an ancestor of *A* at some later timestamp *t' > t* involves renaming *B* out of *A* as a necessary step.

Explanation. The only operations that alter ancestor-descendant relationships among directories are `mkdir`, `rmdir` and `rename`. It is straight-forward to see that neither `mkdir` nor `rmdir` can change the relationship between *A* and *B*, which leaves us with `renames`.

Consider the sequence of `renames` that can make *B* an ancestor of *A*. The last operation would have to be a `rename` of *A* into *B* or one of *B*'s descendants. However, we know from observation 1.2 that moving an ancestor into a descendant (in a single step) is not a legal `rename` operation. Hence, just before performing this last rename which moves *A* into *B* (or one of its descendants), we know that *A* could not have been an ancestor of *B*. But since *B* was *A*'s child to begin with, it implies that *B* had to be moved out of *A* along the way, as a necessary step, before performing the last `rename`. □

Observation 2.2. Let *absolute_path(d)* @ *t* denote the absolute path of directory *d* at timestamp *t*. A directory in MemFS is said to be clean (non-dirty) at timestamp *t*, if there are no pending operations modifying that directory, yet to be flushed at timestamp *t*. If a directory *d* and all its ancestors in MemFS are clean at timestamp *t*, then:

MemFS.*absolute_path(d)* @ *t* = DiskFS.*absolute_path(d)* @ *t*

Explanation. The property obviously holds for the root directory, since its absolute path never changes in memory or on the disk. Using this as the basis, we use a recursive induction to argue for any other directory in the file system.

If d is a directory and p is *parent(d)*, then if p is clean at timestamp t, then it implies that p has the same children in MemFS and DiskFS at timestamp t. Hence d has the same relative path from p, in MemFS and DiskFS, at timestamp t.

Now, if we set $d \leftarrow p$ and $p \leftarrow$ *parent(p)* and re-run this argument, we see that d's relative path from *parent(p)* would be the same in MemFS and DiskFS at timestamp t. Going this way all the way up to the root directory, we can conclude that if all the ancestors of a directory are clean at some timestamp t, then the absolute path of that directory would be the same in MemFS and DiskFS at that timestamp. □

Observation 2.3. If a directory d and all its ancestors were clean at timestamp t, and there were operations that modified some of them in MemFS after t, but none of those operations have been flushed to the disk yet at some later timestamp $t' > t$, then:

DiskFS.*absolute_path(d) @ t'* = MemFS.*absolute_path(d) @ t*

Continuing the argument of theorem 2, let us recall that in order to have a loop in DiskFS at t_{loop}, *target* must have been an ancestor of *dst* in DiskFS just before flushing the `rename` operation at t_{loop}. Denoting that timestamp as $t_{rename\_flush\_begin}$ ($< t_{loop}$), we have:

target ∈ DiskFS.*ancestors(dst)* @ $t_{rename\_flush\_begin}$

The loop avoidance protocol in McoreFS `prepare`s the destination directory of a `rename` operation, just before flushing the `rename` itself. Thus, *dst* gets `prepare`d at $t_{rename\_flush\_begin}$, where `prepare` is defined as:

```
prepare(directory d, timestamp t):
  if (d != root):
    prepare(MemFS.parent(d) @ t, t)
  Flush all operations with timestamp <= t that modify d
```

Specifically, *dst* gets prepared at timestamp $t_{rename\_flush\_begin}$ by invoking `prepare(dst,` t_{rename}`)` (recall that t_{rename} is the timestamp at which the directory `rename` operation was performed in MemFS). This call to `prepare` ensures that for every directory $d \in$ MemFS.*ancestors(dst)* @ t_{rename}, there is no pending operation yet to be flushed to the disk, which modifies d and has a timestamp <= t_{rename}. Thus, when `prepare` returns, the state of each of these directories in DiskFS is at least as up-to-date as they were in MemFS at timestamp t_{rename}.

However, we had already established that in order to form a loop in DiskFS at t_{loop}, *target* must have been an ancestor of *dst* at $t_{rename\_flush\_begin}$. This implies that at the end of `prepare(dst,` $\texttt{t}_{rename}$`)`, *target* must have been an ancestor of *dst* in DiskFS, as there is no intermediate flush step between `prepare(dst,` $\texttt{t}_{rename}$`)` and flushing the `rename` operation at t_{loop}.

So, now we evaluate the different scenarios in which *target* could have ended up being an ancestor of *dst* at the end of the `prepare`. By the definition of `prepare`, for every directory $d \in$ MemFS.*ancestors(dst)* @ t_{rename}, its state on the disk would be *at least* as up-to-date as its state in memory at timestamp t_{rename}. Thus, we have 2 possible cases to consider, for each such directory $d \in$ MemFS.*ancestors(dst)* @ t_{rename}:

- **Case 1: DiskFS.*d* after `prepare` is *exactly* as up-to-date as MemFS.*d* @ t_{rename}** Using observation 2.3, we can conclude that:

  DiskFS.*absolute_path(dst)* just after `prepare` = MemFS.*absolute_path(dst)* @ t_{rename}

  In other words, if *target* $\in$ DiskFS.*ancestors(dst)* just after `prepare`, then *target* $\in$ MemFS.*ancestors(dst)* at the beginning of t_{rename}. But that would have caused a loop in MemFS at the end of the `rename` operation at t_{rename}, contradicting theorem 1. This rules out case 1.

- **Case 2: DiskFS.*d* after `prepare` is *more* up-to-date than MemFS.*d* @ t_{rename}**

  This implies that by the end of `prepare`, some operations with timestamps > t_{rename} that modified the ancestors of *dst*, have also been flushed to the disk. Since we ruled out case 1, it follows that the operations with timestamps > t_{rename} that were flushed by the end of `prepare` must have caused *target* to become an ancestor of *dst* on the disk.

  However, note that *target* was moved into *dst* in MemFS by the `rename` operation at timestamp t_{rename}. Using observation 2.1, it is clear that if *target* had to become an ancestor of *dst* once again in MemFS after t_{rename}, it must have involved moving *target* out of *dst* as a necessary step, using another `rename` operation, with timestamp > t_{rename}. But these two `rename` operations both modify *dst*, and hence they are guaranteed to be flushed to disk strictly in the increasing order of their timestamps. This implies that the set of operations modifying *dst* with timestamps > t_{rename} that may have been flushed to disk by the end of `prepare` does not include the `rename` that moves *target* out of *dst* after t_{rename}. Hence, even case 2 cannot cause *target* to become an ancestor of *dst* on the disk just before t_{loop}. This leaves us with no way to cause a loop in DiskFS at t_{loop}, which completes our argument for theorem 2 by contradiction.

□

Bibliography

[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] 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.

[3] 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.

[4] M. Cao, T. Y. T'so, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the art: Where we are with the ext3 filesystem. In *Proceedings of the Linux Symposium*, pages 69–96, Ottawa, Canada, July 2005.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system, 2012. http://pdos.csail.mit.edu/6.828/2012/xv6.html.

[12] 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.

[13] R. Eqbal. ScaleFS: A multicore-scalable file system. Master's thesis, Massachusetts Institute of Technology, Aug. 2014.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Y. Klonatos, M. Marazakis, and A. Bilas. A scaling analysis of Linux I/O performance. Poster presented at EuroSys, 2011.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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.

[34] 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.

[35] 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.

[36] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 449–464, Broomfield, CO, Oct. 2014.