ScaleFS: A Multicore-Scalable File System

by

Rasha Eqbal

B.Tech.(Hons.) in Computer Science and Engineering
Indian Institute of Technology Kharagpur (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 Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

Author . . . .
Department of Electrical Engineering and Computer Science
August 29, 2014

Certified by . . . .
M. Frans Kaashoek
Professor
Thesis Supervisor

Certified by . . . .
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

ScaleFS: A Multicore-Scalable File System

by

Rasha Eqbal

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2014, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

It is difficult to achieve durability and crash consistency in file systems along with multicore scalability. Commutative file system operations, which should scale according to the Scalable Commutativity Property, conflict on shared resources like coarse-grained locks and pages present in the page cache or buffer cache. Furthermore, data structures that are on separate cache lines in memory (e.g., directory entries) are grouped together when the file system writes them to disk for durability. This grouping results in additional conflicts.

This thesis introduces a new design approach that decouples the in-memory file system from the on-disk file system, using per core operation logs. This facilitates the use of highly concurrent data structures for the in-memory representation, which is essential for commutative operations to proceed conflict free and hence scale perfectly. The in-memory representation does not propagate updates to the disk representation immediately, instead it simply logs the operation in a per core logical log. A `sync` or an `fsync` call processes these operations and applies them to the disk. Techniques based on time stamping linearization points of file system operations ensure crash consistency, and dependency tracking ensures good disk performance.

A prototype file system, SCALEFS, implements this new approach and techniques. Experiments using COMMUTER and SCALEFS show that the implementation is conflict free for 99% of test cases involving commutative operations.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

I would like to thank my thesis advisers Frans Kaashoek and Nickolai Zeldovich for their constant guidance. They have been incredibly supportive and I really enjoyed working with them. I am deeply grateful for all the ready help I received from Austin Clements. Thank you Austin for always being willing to discuss things, and all the bug fixes and tweaks that made my life so easy.

I want to thank my parents for their love and blessings, and for being so supportive in everything I do. I also want to thank all my wonderful friends for their love and encouragement and for taking care of me. Lastly, thank you Dheeraj for being an unwavering source of strength and support, and for staying by my side through highs and lows.

Contents

List of Figures

Chapter 1

Introduction

Many of today's file systems don't scale well on multicore machines, and much effort is spent on making them better to allow file system intensive applications to scale better. For example, a detailed study of the Linux file system evolution [18] indicates that a quarter of performance patches in file system code are related to synchronization. Yet, Klonatos et al. observe that for a variety of Linux file system operations I/O performance does not scale adequately with the number of available hardware threads [15]. While inefficient handling of I/O requests is a contributing factor, unnecessary lock contention in the file system caching layer also plays a role in limiting scalability. The authors recommend re-examining the entire I/O path in the Linux kernel. 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, SCALEFS, that implements this design.

SCALEFS's goals are as follows:

1. Achieve multicore scalability for file system operations. In particular, SCALEFS's design should scale perfectly for all file system operations that commute [8]. That is, SCALEFS should be able to choose implementations of file system data structures that are conflict-free—one core doesn't have to read data written by another core—for commutative file system operations.

2. Achieve durability, crash consistency, and fast recovery for file system data.

3. Achieve good disk throughput. The amount of data written to the disk should be commensurate with the changes that an application made to the file system, and that data should be written efficiently.

These goals are difficult to achieve together. For example, as we will discuss in related work, Linux achieves goals 2 and 3, but fails on 1. SV6 [8] achieves 1 but fails on goals 2 and 3.

The basic approach SCALEFS follows is to decouple the in-memory file system completely from the on-disk file system. Although existing file systems have done so to some degree, SCALEFS takes this approach to its logical extreme. For example, Linux represents directories in memory with a concurrent hash table, but when it updates a directory it also propagates these changes to the in-memory ext4 physical log, which is later flushed to the disk. This is essential to ensure durability, 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. By the definition of commutativity [8], the two creates commute and should be conflict-free to allow for scalability. However, if these operations happen to update the same disk block, they no longer remain conflict-free in this design, despite being commutative.

SCALEFS separates the in-memory file system from the on-disk file system using an operation log (oplog) [2]. The oplog consists of per-core logs of logical file system operations (e.g., `link`, `unlink`, `rename`). SCALEFS sorts the operations in the oplog, and applies them to the on-disk file system when `fsync` and `sync` are invoked. For example, SCALEFS implements directories in 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, at that point SCALEFS merges the per-core logical logs in an ordered log, prepares the on-disk representation, adds the updated disk blocks to a physical log, and finally flushes the physical log.

This organization allows decoupling of the in-memory file system from the on-disk file system. The in-memory file system can choose data structures that allow for good concurrency, choose ephemeral 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 efficient crash recovery and good disk throughput. In principle, the on-disk file system can even re-use an existing on-disk format.

This decoupling approach allows for good multicore scalability and disk performance, but it doesn't provide crash consistency. The on-disk representation must correspond to a snapshot at some linearization point of the in-memory representation of the file system. `fsync` must ensure that it flushes a snapshot of the file system that existed at a linearization point preceding the `fsync` invocation.

However, flushing out all the changes corresponding to this snapshot might incur a lot of disk write traffic. Some changes might not need to be present on the disk for the `fsync` to be causally consistent with the operations that preceded it. For example, an `fsync` on a file does not need to flush to disk the creation of another file in an unrelated directory. So it must ensure that it flushes only the data structures related to the file for which the application called `fsync`. This set of data structures is a subset of the snapshot that existed at a linearization point before `fsync` was invoked, such that the state on the disk is causally consistent with all the operations that preceded the `fsync` in memory.

SCALEFS solves the consistent-snapshot problem by time-stamping linearization points for logical in-memory operations, and merging the per-core logs with the operations sorted by the timestamps of the linearization points. SCALEFS ensures that the snapshot is minimal by tracking dependencies between operations in the unified log. At an invocation of `fsync`, SCALEFS applies only dependent operations that relate to the file or directory being fsynced.

We have implemented SCALEFS by modifying SV6. We adopted the in-memory data structures from SV6 [8] to implement the in-memory file system, but extended it with an on-disk file system from xv6 [10] using the decoupling approach.

Experiments with SCALEFS on COMMUTER [8] demonstrate that SCALEFS's multicore scalability for commutative operations is as good as SV6 while providing durability. The conflict heatmaps that COMMUTER generates for SV6 and SCALEFS are identical, except for in the rows corresponding to the `sync` and `fsync` system calls, which were not present at all in SV6. Experimental results also indicate that SCALEFS fares better than a Linux ext4 [20] file system. SCALEFS is conflict free in 99% of the commutative test cases

COMMUTER generates, while Linux is conflict free for only 70% of them.

The SCALEFS implementation has some limitations. First, it doesn't support all the file system operations that Linux supports and can therefore not run many Linux applications. Second, SCALEFS doesn't scale with disks; we use one disk for I/O. We believe that neither limitation is fundamental to SCALEFS's design.

A potential downside of the decoupling approach SCALEFS uses is that some file system data lives twice in memory: once in the in-memory file system and once to manage the on-disk representation. We haven't yet run experiments to measure the memory overhead this dual representation introduces. Another aspect of SCALEFS that is untested is its disk performance, i.e., whether SCALEFS can write data at the rate of the disk bandwidth.

The main contributions of the thesis are as follows. First, we propose a new design approach for durable multicore file systems by decoupling the in-memory file system from the on-disk file system using an oplog. Second, we describe techniques based on time stamping linearization points that ensure crash consistency and high disk performance. Third, we have implemented the design and techniques in a SCALEFS prototype and confirmed with experiments that the prototype achieves goals 1 and 2.

The rest of the thesis is organized as follows. §2 describes the work on which SCALEFS builds, §3 describes the motivation behind SCALEFS, §4 explains the design of SCALEFS, §5 contains implementation details, §6 presents experimental results and §7 concludes.

Chapter 2

Related Work

The main contribution of SCALEFS is the split design that allows the in-memory file system to be designed for multicore scalability and the on-disk file system for durability and disk performance. The key idea that allows a clean separation between the two is to use a logical log. The rest of this chapter relates SCALEFS's separation to previous designs, SCALEFS's memory file system to other in-memory file systems, and SCALEFS's disk file system to previous durable file systems.

Separation using logging. File systems use different data structures for in-memory file system operations and on-disk operations. For example, directories are often represented in memory differently than on disk to allow for higher performance and parallelism. Linux's dcache [9, 21], which uses a concurrent hash table is a good example. But, file systems don't decouple the in-memory file system from on-disk file system; operations that perform modifications to in-memory directories also manipulate on-disk data structures. This lack of decoupling causes cache-line movement that is unnecessary and limits scalability. One reason that previous file systems might not have separated the two so strongly is because separation adds complexity and memory overhead.

Some file systems specialized for non-volatile memory systems have pushed the separation further; for example, ReconFS [19] 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. SCALEFS's design doesn't 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 [4].

SCALEFS implements the log that separates the in-memory file system from the on-disk file system using an oplog [2]. Oplog allows cores to append to per-core logs without any interactions with other cores. SCALEFS extends oplog's design to sort operations by timestamps of linearization points of file system operations to ensure crash consistency.

In-memory file systems. SCALEFS adopts its in-memory file system from SV6 [8]. 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 doesn't support writing data to durable storage. SCALEFS's primary contribution over SV6's in-memory file system is showing how to combine multicore scalability with durability. To do so, SCALEFS extends the in-memory file system to track linearization points, and adds an on-disk file system using a logical log sorted by the timestamps of linearization points.

There has been recent work towards a better design of the page cache for multicore parallelism. For example, Zheng et al. introduce a set-associative page cache[1] whose goal is to provide performance equivalent to direct I/O for random read workloads and to preserve the performance benefits of caching when workloads exhibit page reuse. In principle, SCALEFS's in-memory file system could adopt such ideas, but it will require adding support for timestamping at linearization points.

On-disk file systems. SCALEFS's on-disk file system is a simple on-disk file system based on xv6 [10]. It runs file system updates inside of a transaction as many previous file systems have done [6, 13], has a physical log for crash recovery of metadata file system operations (but less sophisticated than say ext4's design [20]), and implements file system data structures on disk in the same style as the early version of Unix [23].

Because of the complete decoupling of the in-memory file system and on-disk file system, SCALEFS could be modified to use ext4's disk format, and adopt many of its techniques to support bigger files, more files, etc, and better disk performance. Similarly, SCALEFS's on-disk file system could be changed to use other ordering techniques than transactions; for example, it could use soft updates [12], a patch-based approach [11], or backpointer-based consistency [5].

SCALEFS's on-disk file system doesn't provide concurrent I/O using multiple storage devices; its throughput is limited to the performance of a single storage device. But, because the on-disk file system is completely decoupled from the in-memory file system, in principle SCALEFS's on-disk file system could be extended independently to adopt techniques from systems such as LinuxLFS, BTRFS, TABLEFS, NoFS, XFS [14, 24, 22, 5, 25] to increase disk performance.

SCALEFS's on-disk file system is agnostic about the medium that stores the file system, but in principle should be able to benefit from recent work using non-volatile memory such as buffer journal [17] or ReconFS to minimize intensive metadata writeback and scattered small updates [19].

Chapter 3

Design Overview

This chapter explains what problems motivated us to explore a design that decouples the in-memory file system from the on-disk file system. Furthermore, it explains the subtle issues that must be gotten right in a decoupled design by considering several variations that turn out to be incorrect.

SCALEFS's goal is to allow commutative operations to scale while providing durability. Consider two concurrent file system operations that do not involve any disk I/O; for example, `create(f1)` and `create(f2)` under a directory `dir1` where `f1` and `f2` are two distinct files. The Scalable Commutativity Property [8] tells us that these two `create` operations commute and can scale if they are conflict-free. Conflict-freedom can be achieved by using a concurrent hash table for the directory representation in memory. Each file maps to a different entry in the hash table and the `create` operations can proceed conflict-free and thus scale perfectly.

SCALEFS's durability goal complicates matters, however, because the two updates may end up sharing the same disk block. If that is the case, then the core which performed one update must read the update from another core's cache to construct the disk block to be written out. This construction of the disk block with shared updates is not conflict-free. Since file systems routinely pack several in-memory data structures into a single disk block (or whatever the smallest unit of disk I/O is), it is often the case that operations that are commutative and can be implemented in a conflict-free manner without durability will lose conflict-freedom (and thus multicore scalability) if durability is required.

3.1 Decoupling using an operation log

To solve this conundrum SCALEFS decouples the in-memory file system (MEMFS) from the on-disk file system (DISKFS). The in-memory file system uses concurrent data structures that would allow commutative in-memory operations to scale, while the other file system deals with constructing disk blocks and syncing those to disk. This approach allows all commutative file system operations to execute conflict-free until an invocation of `fsync`.

The main challenge in realizing this approach is that `fsync` writes a *crash-consistent* and *minimal* snapshot to disk. With crash-consistent we mean that after a failure the on-disk file system must reflect a state of the file system that is consistent with what a running program before the crash may have observed. For example, if a program observed the existence of a file in MEMFS before a crash, then after recovery DISKFS must also have that file. With *minimal* we mean that the snapshot that is written by `fsync` contains only the changes related to the file or directory being fsynced, and not dirty data of files that are not being fsynced.

These two properties are difficult to achieve together. Consider the strawman design that acquires a global lock on the whole of MEMFS when `fsync` is invoked, and then flushes any modified file system structures to the disk. This design would ensure that we have on disk a consistent snapshot of the file system since MEMFS is not being changed at all while the `fsync` occurs. But, this design has two problems: (1) operations on files unrelated to the fsynced file are stalled for the duration of the `fsync` and (2) `fsync` writes to disk a huge amount of data.

It may appear that fine-grained locking can solve this problem: `fsync` holds a lock on just the file or directory being fsynced and flushes its modified content to the disk. This design, however, isn't correct. Let us consider the following sequence of instructions: `rename(dir1/a, dir2/a)` followed by `fsync(dir1)`. Here we would end up losing file `a` since we only synced `dir1` and not `dir2`, violating the atomicity of the `rename` operation. This suggests that SCALEFS must somehow track that `dir2` was involved in an operation that `fsync(dir1)` exposes to the disk.

SCALEFS's approach is to maintain a log of operations that update file metadata spread

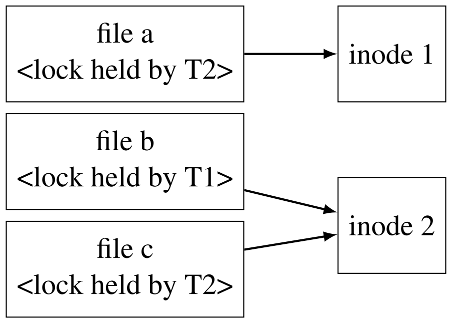

Figure 3-1: Concurrent execution of `rename(a,c)` and `rename(b,c)`. Thread T1 holds a lock on file `b`, while thread T2 holds a lock on files `a` and `c`. The set of locks held by these operations is disjoint, so timestamps obtained while holding the locks might not reflect the actual order of execution.

across the file system. This log must be a logical log instead of a physical log, because otherwise, as we observed earlier, two commutative `create` operations would no longer be conflict-free if they share the same physical disk block. Therefore, the log contains operations such as `link`, `unlink`, `rename`, etc. So in the motivating example, the log contains `rename(dir1/a, dir2/a)` and when `fsync(dir1)` is invoked, SCALEFS flushes the changes related to the entire `rename` operation to the disk.

3.2 Ordering operations

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. It may appear that fine-grained locking could be used, but it isn't sufficient to ensure crash consistency. For example, a possible solution might involve logging the operations in the logical log under all the MEMFS locks they acquire and only then releasing them. But consider the following scenario. 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.

Assume that T1 goes first. It sees that `b` and `c` are hardlinks to the same inode. So all it needs to do is remove the directory entry for `b`. The only lock the thread acquires is on `b` since it does not touch `c` at all. T2 then acquires locks on `a` and `c` and performs the `rename`. Figure 3-1 depicts this scenario. Both the threads now proceed to log the operations holding their respective locks. T2 might end up getting logged before T1, even though to the user the operations seemed to have executed in the reverse order. 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). The reason behind this discrepancy is that reads are not guarded by locks. However if we were to acquire read locks, SCALEFS would sacrifice conflict-freedom of in-memory commutative operations.

Instead of fine-grained locks, we might consider using version numbers on each mutable element, but that can lead to large space overhead. Say we maintain a version number in each mutable element in MEMFS. Each operation has a vector of the version numbers of all elements it read from or wrote to. Operations are logged in the logical log along with their version vectors. When `fsync` is invoked, the operations in the logical log are then sorted topologically on their version vectors and applied to the disk. This captures the correct order in which operations occurred on MEMFS and the disk remains crash-consistent with what applications observed happening in memory earlier. For example, for the previous example, T1 would have a smaller version number for `c` in its version vector, and hence would be ordered before T2. The problem with this design, however, is that it incurs a high memory overhead. Every entry in the name hash table would need a version number.

We can do better and not pay the space overhead for version vectors by observing that to achieve POSIX semantics all metadata operations must be linearizable. Thus, every metadata operation must have a linearization point. If we can timestamp each linearization point with a `tsc` value, we can be sure that the order indicated by these timestamps is the order in which MEMFS applied these operations. `fsync` can order the operations by their timestamps and ensure crash consistency without incurring any additional space overhead.

3.3 Determining linearization points

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

This design, however, would mean that reads must be protected by locks too, which conflicts with our goal of achieving scalability for commutative operations. Consider three file names `x`, `y` and `z` all pointing to the same inode. If one thread issues the syscall `rename(x,z)` while another calls `rename(y,z)` concurrently, the definition of commutativity tells us that these two syscalls commute. This is because it is impossible to determine which syscall completed first just by looking at the final result. Our design ensures that these syscalls are conflict-free; as mentioned earlier, if the source and destination for a `rename` are the same, the source is simply removed without holding any lock on the destination. However, if we were to protect reads with a lock too, these syscalls would no longer remain conflict-free and hence would no longer scale.

To solve this problem, MEMFS protects read operations with a seqlock [16, §6]. MEMFS reads the `tsc` value under the seqlock, and retries if the seqlock read does not succeed. Each retry returns a new timestamp. The timestamp of the last retry corresponds to the linearization point. In the normal case the two reads succeed without any conflicts and MEMFS incurs no additional cache line movement.

3.4 Using oplog

SCALEFS uses an oplog [2] for implementing the logical log. An 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 conflict-free operations don't conflict because of appending log entries. The challenge in using an oplog in SCALEFS 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 3-2.

Figure 3-2 shows the execution of two operations on two different cores: $op1$ on $core1$

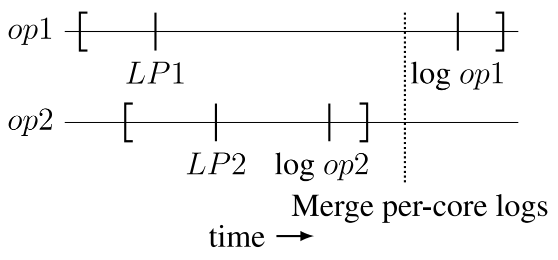

Figure 3-2: Merging of the per-core operation logs might need to wait for some operations to complete.

and $op2$ on $core2$. $LP1$ is the linearization point of $op1$, $LP2$ of $op2$. If the per-core logs happen to be merged at the time as indicated in the figure, $op1$ will end up missing from the merged log even though its linearization point was before that of $op2$. Hence the merge must wait for certain operations that are in progress to complete. SCALEFS handles this by keeping track of per-core timestamp counters for when an operation enters and later leaves its critical section. SCALEFS uses this information at the time of the merge to determine if there are any operations in flight that the merge should wait for.

3.5 Discussion

The design described in the previous sections should achieve good multicore scalability, because operations that commute should be mostly conflict free. The design of MEMFS can achieve this by implementing files and directories using highly concurrent data structures. Since MEMFS is not coupled with the disk, it is not restricted in the choice of data structures that fulfill its purpose. All that MEMFS needs to do in order to later sync the operations to the disk is log them in the logical log, which is also a per-core data structure. As a result, file system operations should conflict neither during execution on files and directories in MEMFS nor while logging themselves in the logical log.

`fsync` should also be conflict free for most file system operations it commutes with (e.g., creation of a file in an unrelated directory). The only time an `fsync` conflicts with a commutative operation is when MEMFS logs the operation in a per-core log and `fsync` scans that log while merging. Since the merge only flushes the logs on cores that have a non-empty one, this conflict need not always occur. For example, a previous `sync` or `fsync` might have just flushed out all the logs.

Another benefit of using a decoupled design is that we can use an existing on-disk file system format and change the design of DISKFS accordingly. Since MEMFS and the logical logging mechanism do not depend on the disk format at all, DISKFS can be changed with relative ease to support any disk format.

Chapter 4

ScaleFS

This chapter details the design of ScaleFS. ScaleFS consists of MemFs– the in-memory file system representation, DiskFs– the on-disk file system representation, and the logical log that acts as an interface layer between the two.

4.1 MemFs

Decoupling the in-memory and the disk file systems allows MemFs to be designed for multicore concurrency. MemFs represents directories using chain hash tables that map file/directory names to inode numbers (see Figure 4-1). 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, each page protected by its own lock. This representation of file pages is very similar to the way RadixVM represents virtual memory areas [7]. MemFs creates the in-memory directory tree on demand, reading in components from the disk as they are needed. We will 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 on each new instantiation of MemFs from the disk. If MemFs were to use the same mnode numbers as inode numbers on the disk, it would also have to maintain a free mnode list much like the inode free list in DiskFs, since the maximum number of inodes a file system can have is pre-determined.

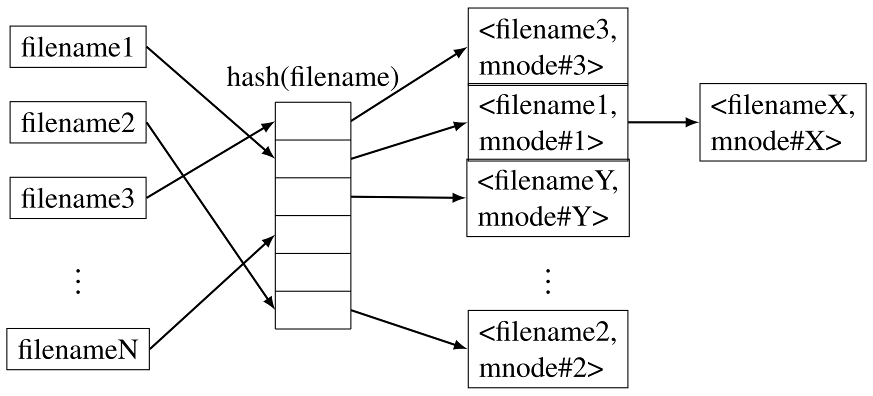

Figure 4-1: Representation of a directory as a hash table in MEMFS. The table uses hashed filenames as keys, and the buckets store mappings between filenames and mnode numbers.

The creation of every new file or directory would then have to traverse this free mnode list to find an available mnode number, and concurrent creates would not scale.

Instead, 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.

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

SCALEFS does not need to update the hashmap immediately after the creation of an mnode. It can wait for DISKFS to create the corresponding inode on a `sync` or an `fsync` call, which is also when DISKFS looks up the disk free inode list in order to find a free inode. As a result, MEMFS can process `create` calls scalably.

4.2 DISKFS

The implementation of DISKFS depends on the disk file system format used (e.g., ext3 [3], ext4 [20], etc.). SCALEFS uses the xv6 [10] file system, which has a physical journal for crash recovery. The file system follows a simple Unix-like format with conventional inodes,

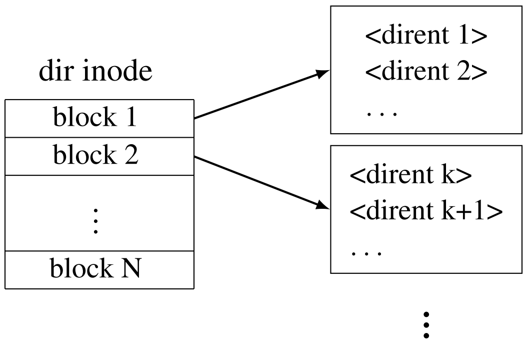

Figure 4-2: Representation of a directory in DISKFS. The directory inode stores a list of block addresses. Each block contains a series of directory entries. The last but one address points to an indirect block, and the last one points to a doubly indirect block.

files and directories. Inodes can have up to doubly indirect blocks. Figure 4-2 shows the representation of a directory in DISKFS, which is completely different from the MEMFS representation (Figure 4-1).

DISKFS maintains a buffer cache to cache blocks that are read in from the disk. In our current design DISKFS's buffer cache is not multicore because our current design is focused on a single disk. To achieve I/O scalability with multiple disks, the buffer cache could be split up on multiple cores, and all the cores could flush out their buffer caches to the disks in parallel.

4.3 The logical log

The logical log ties MEMFS and DISKFS together but allows in-memory commutative file system operations to proceed completely independently of each other. If a core updates the in-memory data structures, it does not construct the updated disk block. Instead, upon successful completion of file system operations, MEMFS logs the file system operations in per-core logical logs, timestamping them with the `tsc` value at their linearization points as explained in §3.3. SCALEFS represents an operation logically by an operation type, the files and directories involved in the operation, and any further information required to carry out the operation on DISKFS later. Typically this information is the set of arguments that were passed to the corresponding MEMFS system call.

SCALEFS maintains the per-core logical logs using a modified version of oplog [2]. The per-core logs are simply vectors of operations sorted in increasing order of their timestamps. A `sync` or an `fsync` merges these vectors into one, maintaining the timestamp ordering, and clears the per-core logs. It then processes the operations in timestamp order. For each operation, DISKFS constructs the disk blocks that were updated and writes them to a journal on the disk within a transaction. In order to provide crash consistency, DISKFS syncs the dirty blocks to the disk only after they have been successfully committed to the disk journal within the transaction.

4.4 Operations in the logical log

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

By omitting certain operations from the log, SCALEFS not only saves space that would otherwise be wasted to log them, but it also simplifies merging the per-core logs and dependency tracking as described in §4.6. As a result the per-core logs can be merged and operations applied to the disk much faster.

MEMFS logs only file system operations that involve more than one mnode in the logical log. For example, it logs the creation of a file since there are two mnodes involved in the operation: the new file and the parent directory.

However it does not log a change in file size, or a write to a file, since there is only one mnode involved here. In this case, a single `fsync` would reflect the changes properly on the disk. It is not possible for the operation to end up being synced to the disk only partially. So such an operation does not need to be logged. In the example of a file write, a single `fsync` call on the file would flush all the updates to the disk in a consistent manner.

In the other example involving a file creation, simply syncing the new file does not work. DISKFS must also create a directory entry corresponding to this new file under the parent directory. Moreover, both these actions need to take place within a single transaction in order to preserve the atomicity of the `create`. Therefore MEMFS logs all `create` operations, and by the same reasoning logs `link`, `unlink` and `rename` operations too.

```
 1: mfile.fsync_lock.acquire()
 2: fsync_tsc ← rdtscp()
 3: process_logical_log(fsync_tsc)
 4: size ← mfile.size
 5: offset ← 0, page_index ← 0
 6: while offset < size do
 7:     if mfile.pages[page_index] == nullptr then
 8:         size ← mfile.size
 9:         if size ≤ offset then
10:             break
11:         end if
12:     end if
13:     if mfile.pages[page_index].dirty_bit.is_set() then
14:         Flush mfile.page[page_index] to disk
15:         mfile.page[page_index].dirty_bit.clear()
16:     end if
17:     page_index ← page_index+1
18:     offset ← offset + PAGE_SIZE
19: end while
20: Create fsync_transaction
21: add_to_transaction(inode length ← size)
22: add_to_transaction(inode direct and indirect blocks)
23: Mark disk blocks in inode past size as free, add_to_transaction(free bitmap changes)
24: Mark newly allocated blocks as allocated, add_to_transaction(free bitmap changes)
25: write_to_disk(fsync_transaction)
26: Free blocks in the in-memory free block list, corresponding to line 23
27: mfile.fsync_lock.release()
```

Figure 4-3: `fsync` call on an mfile

4.5 The `fsync` call

The steps involved in an `fsync` call on a file are shown in Figure 4-3. `fsync` first merges the per-core logical logs and processes relevant operations, applying them to the disk. Then it flushes out dirty file pages. Finally it syncs file inode updates and metadata changes.

`fsync` operates within a transaction consisting only of the updates in lines 21-24 of Figure 4-3 (i.e., metadata updates). The dirty file pages synced out in lines 6-19 are not a part of the transaction, thereby saving space in the physical journal on disk. Syncing all the file pages is not atomic either, which is an accepted design that many file systems follow.

Acquiring the `fsync_lock` in line 1 permits only one `fsync` call to be invoked on a file at a time. This lock ensures that the file pages flushed out, which are not a part of the transaction, are consistent with the metadata updates synced to the disk. If no lock was held, two concurrent `fsync` calls might observe two different states of the mfile and hence flush out different numbers of file pages. Then each `fsync` would make changes to the inode length and direct and indirect blocks within its own transaction depending on the state of the mfile it observed. These transactions might then end up being logged to disk in any order, depending on the order in which they acquire the journal lock while writing to disk in line 25. Thus the file inode on the disk might become inconsistent with the file contents.

Line 2 assigns to `fsync_tsc` the value of the Time Stamp Counter (TSC) register on the core the `fsync` was invoked on. `process_logical_log()` requires `fsync_tsc` in order to merge and process only those operations that have their linearization points (and their timestamps) before `fsync_tsc`. The merge might have to wait for some in-flight operations to complete in order to counter the scenario illustrated in Figure 3-2. SCALEFS implements waiting using per-core operation start and end timestamps as described in §3.4. `process_logical_log()` only processes operations the `fsync` depends on, details of which are described in §4.6.

The `while` loop (lines 6-19) traverses the radix array of file pages and flushes out dirty pages, resetting their dirty bits. If `fsync` notices that a page in the sequence is not present (lines 7-12), it reads the size of the mfile again. An absent page could indicate that the page was removed by a concurrent file `truncate`, which is quite possible as the only MEMFS lock `fsync` holds is the `fsync_lock`. If the current `offset` is larger than the size of the mfile, `fsync` has already flushed out all the pages in that range and can break out of the loop.

DISKFS maintains a copy of the free block list in memory in order to keep the free block bitmap on the disk in a consistent state if the `fsync` operation fails before completion. Since file content writes do not take place within a transaction, DISKFS keeps track of new disk blocks that were allocated using the in-memory free block list, and does not update the free block bitmap on the disk while flushing out dirty pages. `fsync` logs changes to the free block bitmap along with inode metadata updates in the transaction. So if the `fsync` call fails after syncing the dirty file pages but before flushing out the `fsync` transaction, the disk will not be in a state where certain newly allocated blocks are marked allocated in the block free bitmap but have not been linked to the file inode yet.

After flushing out dirty file pages, `fsync` creates a transaction consisting of metadata updates: the inode size, changes to the direct and indirect blocks and free bitmap updates. `fsync` converts the new block allocations that were tracked in the in-memory free block list, into free bitmap updates on the disk and logs them in the transaction. It also logs free bitmap changes caused by truncation of pages beyond the new inode size, but holds off marking those blocks as free in the in-memory free block list until the `fsync` transaction has committed to the disk. This ensures that blocks that are freed as a result of the `fsync` are not reused, perhaps by a concurrent `fsync` on another file, before this `fsync` commits.

An `fsync` call on a directory is relatively simple. The only tasks that must be performed are steps 2 and 3 in the pseudocode in Figure 4-3, because MEMFS logs all operations that pertain to directories —`create`, `link`, `unlink` and `rename` —in the per-core logical logs. So all a directory `fsync` needs to do is pick out operations from the logical log that the directory was involved in, or that the `fsync` depends on in some way (as the dependency tracking mechanism describes in §4.6). An `fsync` call on a directory involves no interaction with the MEMFS mdir at all, it only needs to consult the logical log. As a result MEMFS can support operations on a directory in parallel with an `fsync` call on the same directory.

4.6 Dependency tracking in an `fsync` call

One of SCALEFS's goals is to ensure that the amount of disk write traffic an `fsync` on a file or a directory incurs is comparable to the changes the file or directory was involved in. Therefore, an `fsync` processes the logical log to calculate the set of operations that need to make it to the disk before it can be processed, instead of flushing the entire log. This set is roughly the transitive closure of all operations that the file or directory was involved in.

The operations that MEMFS logs in the logical log are similar in the sense that they all involve creation or deletion of links to mnodes. Two kinds of mnodes are involved in these operations: the target or child (i.e., the mnode the link points to), and the parent directory under which the link is created or removed from. For a `create`, the child is the new file or directory being created and the parent is the directory the creation takes place under. Similarly, for a `link` and an `unlink`, the child is the mnode being linked or unlinked and the parent is the directory the `link` or `unlink` takes place under. For a `rename`, the child is the mnode being renamed and the parents are the source and the destination directories.

The function `process_logical_log()` that `fsync` calls in line 3 in Figure 4-3 runs an algorithm that calculates dependencies after the per-core logical logs have been merged into a global one. It creates a list of dependent mnodes, which initially consists only of the mnode the `fsync` was called on. The algorithm processes the logical log starting with the most recent operation, and for each operation checks to see if the child mnode of the operation is present in the list of dependent mnodes. If it is, then the `fsync` depends on the operation, and the operation is added to the dependency list, and the parent mnode(s) of the operation are added to the list of dependent mnodes. The check then continues over the rest of the log, augmenting the list of dependent mnodes and the dependency list if required.

For a directory `fsync`, apart from this check the algorithm also includes operations where the directory was a parent. Having calculated the dependency list, `fsync` applies the operations to disk in increasing order of their timestamps, ensuring that only essential operations get flushed out in the correct order. It removes the operations that have been processed from the global log and leaves the others intact. A future merge of the per-core logs simply appends to this global log.

4.7 The `sync` call

A `sync` call simply merges the per-core logical logs into a global one and flushes out all operations. `sync` records the timestamp `sync_tsc` at which it is invoked and passes it on to `process_logical_log()`, which then merges the logical logs to include all operations that have linearization points before `sync_tsc`. It might have to wait for any operations that are still in flight using the wait and merge procedure described in §3.4. `process_logical_log()` does not run the dependency tracking algorithm when invoked from within a `sync`.

The `sync` then applies all the operations in the merged log to disk in timestamp order.

Since single mnode operations are not logged in the logical log, `sync` also traverses the mnode list to `fsync` any dirty files.

Chapter 5

Implementation

We implemented SCALEFS in SV6, a research operating system whose design is centered around the Scalable Commutativity Property [8]. Previously SV6 consisted of an in-memory file system that MEMFS reuses, modifying it to interact with the logical log. SCALEFS augments SV6's design with a disk file system DISKFS (that is based on the xv6 [10] file system), and a logical log. We implemented the entire system in C++. The numbers of lines of code change involved in implementing each component of SCALEFS are shown in Figure 5-1.

While determining linearization points of operations we discovered a bug in the implementation of MEMFS in SV6. In §3.3 we discussed how a `rename` where both the source and the destination point to the same inode, presents complications in determining the linearization point. MEMFS holds a lock only on the source which it deletes, without acquiring any lock on the destination. Acquiring a lock on the destination would violate the Scalable Commutativity Property. In order to determine the linearization point, we then decided to protect reads with a seqlock.

However a `rename` in SV6 should have read the destination under a seqlock too, as

SCALEFS component	lines of code changed
MEMFS	962
DISKFS	1764
logical log	2055

Figure 5-1: Number of lines of code change involved in implementing SCALEFS.

is demonstrated by the following example. Consider two file names `A` and `B` that point to the same inode, and two concurrent renames: `rename(A,B)` and `rename(B,A)`. Let us assume that the first rename reads `B` and finds it to point to the same inode as `A`, while the second rename also reads `A` and finds it to point to the same inode as `B` at the same time. The `rename` operations would then proceed to remove `A` and `B` respectively, and would end up deleting both the file names. We modified MEMFS to read under a seqlock. With this modification, one of the comparison checks fails and the problem is avoided.

Chapter 6

Evaluation

We ran experiments to try to answer the following questions:

- How do file systems today fare in achieving multicore scalability?
- Does SCALEFS achieve multicore scalability?
- Does SCALEFS scale as well as SV6?

6.1 Methodology

To answer these questions we used COMMUTER [8], which is 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 down references to shared memory addresses. Shared memory addresses indicate sharing of cache lines and hence loss of scalability, according to the Scalable Commutativity Property.

We augmented the model COMMUTER uses to generate test cases by adding the `fsync` and `sync` system calls. Previously the model was tailored according to the the design of SV6 which did not support these syscalls.

We do not measure the scalability of SCALEFS on a multicore system. All the results in this chapter are obtained solely by running COMMUTER, but previous work has shown that if commuter does not report conflicts, then software scales well on actual hardware. We also do not measure the disk performance of SCALEFS.

6.2 Does Linux file system achieve multicore scalability?

To answer the first question, we ran COMMUTER on the Linux kernel (v3.8) with an ext4 file system operating in `data=ordered` mode, which corresponds to SCALEFS's `fsync` model of flushing out data to disk before metadata changes are committed. The heatmap in Figure 6-1 shows the results obtained. The green cells in the heatmap indicate no conflicts between the corresponding row and column syscalls. Conflicts are indicated with colors ranging from yellow to red, a shade closer to red indicating a greater fraction of conflicting test cases.

Out of a total of 31539 commutative test cases, the heatmap shows 9443 of them (30%) 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. Previous work [8] reports that some of these conflicts are caused by coarse-grained locks: for example, a lock on the parent directory on creation of two distinct files.

6.3 Does SCALEFS achieve multicore scalability?

Figure 6-2 shows results obtained by running COMMUTER with SCALEFS. 99% of the test cases are conflict-free, which is better than the 70% conflict freedom Linux achieves. The green regions show that the implementation of MEMFS is completely conflict free for 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 —for example, hash tables to represent directory entries, and radix arrays to represent file pages —which also avoid conflicts.

SCALEFS does have conflicts when the `fsync` and `sync` calls are involved. An `fsync`

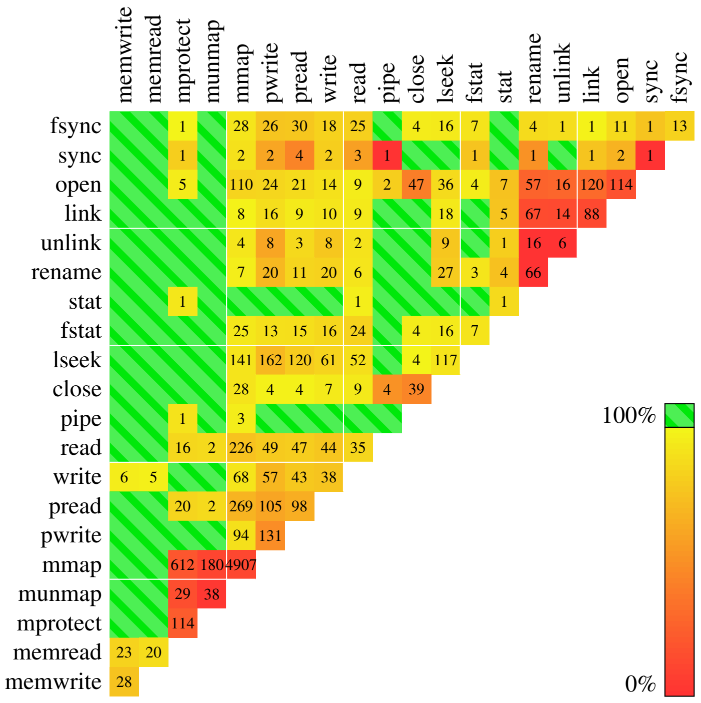

Figure 6-1: Conflict-freedom of commutative operations in the Linux kernel using an ext4 file system. Out of 31539 total test cases generated 22096 (70%) were conflict-free.

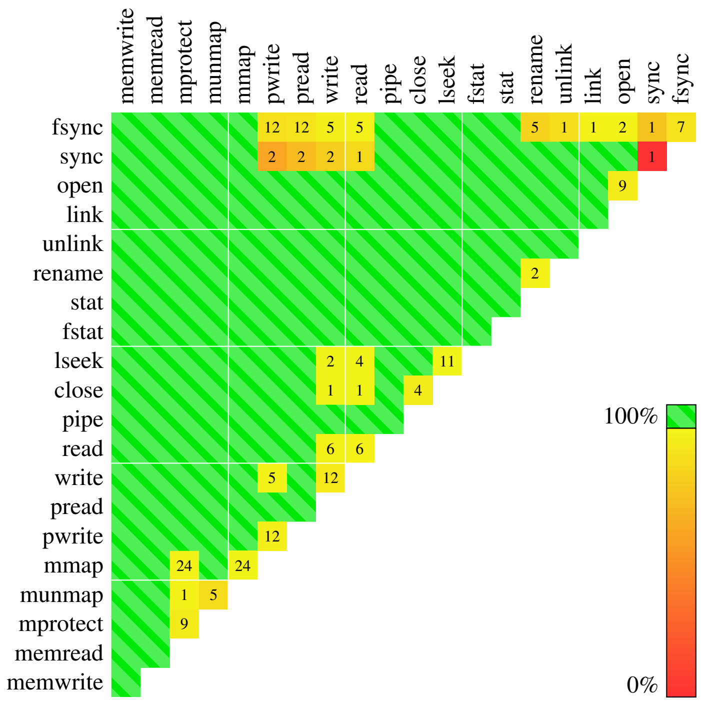

Figure 6-2: Conflict-freedom between commutative operations in ScaleFS. Out of 30863 total test cases generated, 30666 (99%) were conflict-free.

call could conflict with a successful commutative `rename`, `unlink`, `link` and `open` because `fsync` merges the per-core logs while MEMFS is logging those operations. The merge might even have to wait for operations that are in flight, for which it reads the per-core operation start and end timestamp counters. The `rename`, `unlink`, `link` and `open` calls could be updating these counters at the same time, introducing conflicts.

A commutative `rename`, `unlink`, `link`, or `open` operation is conflict free with a `sync` when the operation fails. Successful `rename`, `unlink`, `link`, and `open` operations never commute with a `sync`. So COMMUTER only generates test cases where these operations either fail or have no effect, for example, renaming a file to itself. MEMFS does not log such operations at all, so `sync` can process the logical log concurrently with the execution of these operations without any conflicts. The green regions in the heatmap agree with this reasoning.

Extending the same argument, an `fsync` is conflict free with an unsuccessful `rename`, `unlink`, `link`, and `open` too. The yellow cells in the heatmap indicate some conflict free test cases, which validates this claim.

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

MEMFS can 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 byte, 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.

An `fsync` conflicts with a `sync` because they both try to obtain a lock on the per-core logical logs in order to merge them. Two `fsync` calls on different files conflict on the logical log as well.

The rest of the conflicts are between idempotent operations. Two `sync` calls are com-

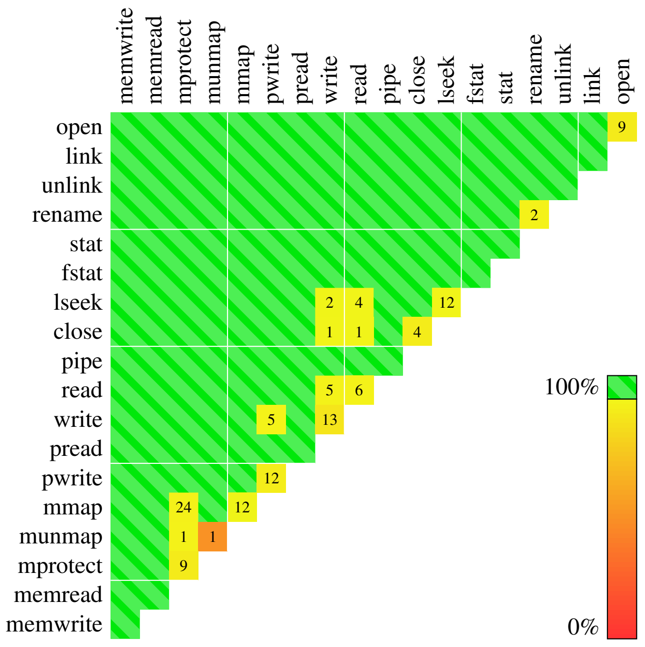

Figure 6-3: Conflict-freedom between commutative operations in sv6 with only an in-memory file system. Out of 13664 total test cases generated 13528 (99%) were conflict-free.

mutative because they are idempotent, but they both contend on the logical log and any dirty files they need to sync. Similarly two `fsync` calls on the same file are idempotent but they conflict on the logical log as well as the file pages.

6.4 Does ScaleFS scale as well as sv6?

Figure 6-3 answers the last question. The figure shows results obtained from running Commuter on the previous version of sv6 with only an in-memory file system. The sv6 heatmap does not have columns for `sync` and `fsync` because sv6 did not support these system calls due to the absence of a disk file system. Comparing this heatmap with Figure 6-2 we can see that ScaleFS introduces no new conflicts between existing syscalls.

Chapter 7

Conclusion and Future Work

We presented a file system design that decouples the in-memory representation from the disk representation, allowing for multicore scalability along with durability and crash consistency. We described a logging mechanism extending oplog [2] to use timestamps passed as arguments to order operations in the per-core logs, and a merge algorithm that waits for all operations before a certain timestamp to be logged before merging the per-core logs. We also introduced a novel scheme to timestamp the logged operations at their linearization points in order to apply them to the disk in the same order a user process observed them in memory, thereby providing crash consistency. We implemented this design in a prototype file system, SCALEFS, that was built on the existing SV6 kernel and we analyzed the implementation using COMMUTER. The results show that the implementation of SCALEFS achieves multicore scalability.

There are several issues we would like to explore in future work. One, measure SCALEFS's scalability on a multicore system to support the results obtained using COMMUTER; two, quantify the memory overhead incurred by the dual file system representation (i.e., MEMFS and DISKFS); three, measure SCALEFS's disk performance; and four, extend the design of DISKFS to support multiple disks and try to achieve I/O scalability too.

Bibliography

[1] A parallel page cache: IOPS and caching for multicore systems. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, Berkeley, CA, 2012. USENIX.

[2] Silas Boyd-Wickizer. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels*. PhD thesis, Massachusetts Institute of Technology, February 2014.

[3] Mingming Cao, Theodore Y. T'so, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the art: Where we are with the ext3 filesystem. In *Proceedings of the Ottawa Linux Symposium(OLS)*, pages 69–96, 2005.

[4] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[5] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[6] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, Robert N. Sidebotham, and Transarc Corporation. The Episode file system. pages 43–60, 1992.

[7] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013. ACM.

[8] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, New York, NY, USA, 2013. ACM.

[9] Jonathan Corbet. Dcache scalability and RCU-walk, April 23, 2012. `http://lwn.net/Articles/419811/`.

[10] Russ Cox, M. Frans Kaashoek, and Robert T. Morris. Xv6, a simple Unix-like teaching operating system. `http://pdos.csail.mit.edu/6.828/2012/xv6.html`.

[11] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. *SIGOPS Oper. Syst. Rev.*, 41(6):307–320, October 2007.

[12] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[13] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 155–162, New York, NY, USA, 1987. ACM.

[14] Martin Jambor, Tomas Hruby, Jan Taus, Kuba Krchak, and Viliam Holub. Implementation of a Linux log-structured file system with a garbage collector. *SIGOPS Oper. Syst. Rev.*, 41(1):24–32, January 2007.

[15] Yannis Klonatos, Manolis Marazakis, and Angelos Bilas. A scaling analysis of Linux I/O performance.

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

[17] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 73–80, San Jose, CA, 2013. USENIX.

[18] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 31–44, San Jose, CA, 2013. USENIX.

[19] Youyou Lu, Jiwu Shu, and Wei Wang. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 75–88, Berkeley, CA, USA, 2014. USENIX Association.

[20] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Ottawa Linux Symposium (2007). http://ols.108.redhat.com/2007/ Reprints/mathur-Reprint.pdf.*

[21] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling dcache with RCU. *Linux J.*, 2004(117):3–, January 2004.

[22] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 145–156, Berkeley, CA, USA, 2013. USENIX Association.

[23] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[24] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[25] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.