Understanding Manycore Scalability of File Systems

Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim, *Georgia Institute of Technology*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/min

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Understanding Manycore Scalability of File Systems

Changwoo Min   Sanidhya Kashyap   Steffen Maass   Woonhak Kang   Taesoo Kim
Georgia Institute of Technology

Abstract

We analyze the manycore scalability of five widely-deployed file systems, namely, `ext4`, `XFS`, `btrfs`, `F2FS`, and `tmpfs`, by using our open source benchmark suite, FxMark. FxMark implements 19 microbenchmarks to stress specific components of each file system and includes three application benchmarks to measure the macroscopic scalability behavior. We observe that file systems are hidden scalability bottlenecks in many I/O-intensive applications even when there is no apparent contention at the application level. We found 25 scalability bottlenecks in file systems, many of which are unexpected or counterintuitive. We draw a set of observations on file system scalability behavior and unveil several core aspects of file system design that systems researchers must address.

1 Introduction

Parallelizing I/O operations is a key technique to improve the I/O performance of applications [46]. Today, nearly all applications implement concurrent I/O operations, ranging from mobile [52] to desktop [46], relational databases [10], and NoSQL databases [6, 43]. There are even system-wide movements to support concurrent I/O efficiently for applications. For example, commercial UNIX systems such as HP-UX, AIX, and Solaris extended the POSIX interface [48, 50, 68], and open source OSes like Linux added new system calls to perform asynchronous I/O operations [17].

Two recent technology trends indicate that parallelizing I/O operations will be more prevalent and pivotal in achieving high, scalable performance for applications. First, storage devices have become significantly faster. For instance, a single NVMe device can handle 1 million IOPS [14, 72, 86], which roughly translates to using 3.5 processor cores to fully drive a single NVMe device [51]. Second, the number of CPU cores continues to increase [19, 62] and large, high-performance databases have started to embrace manycore in their core operations [9, 11, 44, 49, 53, 90]. These trends seem to promise an implicit scaling of applications already employing concurrent I/O operations.

In this paper, we first question the practice of *concurrent I/O* and understand the *manycore scalability*[1] behavior, that we often take for granted. In fact, this is the right moment to thoroughly evaluate the manycore scalability of file systems, as many applications have started hitting this wall. Moreover, most of the critical design decisions are "typical of an 80's era file system" [37]. Recent studies on manycore scalability of OS typically use memory file systems, e.g., `tmpfs`, to circumvent the effect of I/O operations [16, 20–22, 32–34]. So, there has been no in-depth analysis on the manycore scalability of file systems. In most cases, when I/O performance becomes a scalability bottleneck without saturating disk bandwidth, it is not clear if it is due to file systems, its usage of file systems, or other I/O subsystems.

Of course, it is difficult to have the complete picture of file system scalability. Nevertheless, in an attempt to shed some light on it, we present an extensive study of manycore scalability on file systems. To evaluate the scalability aspects, we design and implement the FxMark benchmark suite, comprising 19 microbenchmarks stressing each building block of a file system, and three application benchmarks representing popular I/O-intensive tasks (i.e., mail server, NoSQL key/value store, and file server). We analyze five popular file systems in Linux, namely, `ext4`, `XFS`, `btrfs`, `F2FS`, and `tmpfs`, on three storage mediums: `RAMDISK`, `SSD`, and `HDD`.

Our analysis revealed unanticipated scalability behavior that should be considered while designing I/O-intensive applications. For example, all operations on a directory are sequential regardless of read or write; a file cannot be concurrently updated even if there is no overlap in each update. Moreover, we should revisit the core design of file systems for manycore scalability. For example, the consistency mechanisms like journaling (`ext4`), copy-on-write (`btrfs`), and log-structured writing (`F2FS`) are not scalable. We believe that FxMark is effective in two ways. It can identify the manycore scalability problems in existing file systems and further guide and evaluate the new design of a scalable file system in the future.

In summary, we make the following contributions:

- We design an open source benchmark suite, FxMark, to evaluate the manycore scalability of file systems. The FxMark benchmark suite is publicly available at `https://github.com/sslab-gatech/fxmark`.
- We evaluate five widely-used Linux file systems on an 80-core machine with FxMark. We also analyze how the design of each file system and the `VFS` layer affect their scalability behavior.
- We summarize our insights from the identified scalability bottlenecks to design a new scalable file system for the future.

The rest of this paper is organized as follows: §2 pro-

[1] We sometimes mention manycore scalability as scalability for short.

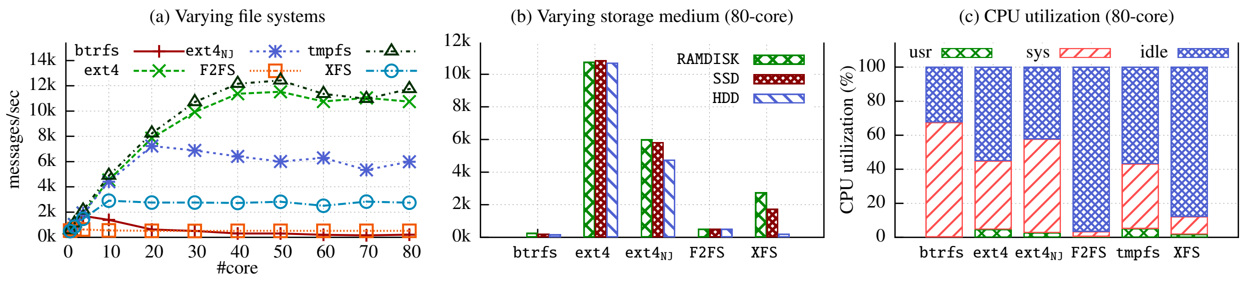


Figure 1: Exim throughput (i.e., delivering messages) on six file systems (i.e., `btrfs`, `ext4`, `F2FS`, `tmpfs`, `XFS`, and `ext4` without journaling, $\texttt{ext4}_{\texttt{NJ}}$) and three storage mediums (i.e., `RAMDISK`, `SSD`, `HDD` at 80-core). We found that the manycore scalability of Exim depends a lot on the file systems (e.g., `ext4` is 54× faster than `btrfs` at 80-core), but does not depend much on storage mediums (e.g., marginal performance difference of `ext4` on `RAMDISK` and `HDD`). To avoid known scalability bottlenecks in Exim, we modified Exim as in the previous study [21] and configured it to disable per-message `fsync()` call.

vides the motivation for our work with a case study; §3 provides an overview of FxMark, and §4 describes our evaluation strategies; §5 and §6 show analysis results; §7 summarizes our findings and §8 discusses implications for OS and file system designers; §9 compares our work with previous work; Finally, §10 provides the conclusion.

2 Case Study

In this section, we show how non-scalable file systems can break the scalability of embarrassingly parallel (i.e., ideally parallelizable) I/O-intensive applications in unexpected ways. The Exim [8] email server is one such application that is designed to scale perfectly, at least from an application's perspective. For example, Exim delivers messages to appropriate mail boxes in parallel and performs each delivery independently. In fact, the message delivery heavily utilizes I/O operations. It consists of a series of operations ranging from creating, renaming, and deleting small files in the spool directories to appending the message body to the per-user mail file. Unfortunately, Exim fails to scale over 10, 20, or 40-core, at most, among the popular file systems in Linux as shown in Figure 1.

File systems kill application scalability. Figure 1(a) shows the throughput of Exim with six different file systems on `RAMDISK`. Surprisingly, the scalability and performance of Exim are significantly dependent on the file system. The performance gap between the best file system, `tmpfs`, and the worst, `btrfs`, is 54× at 80-core. `ext4` and `tmpfs` scale linearly up to 40-core; then the Linux kernel becomes the bottleneck. However, Exim on `F2FS` is hardly scalable; it is 21× slower than `ext4` at 80-core.

Faster storage mediums do not guarantee scalability. With a reasonably large memory, the page cache will absorb most read and write operations, and most write operations can be performed in the background. In this case, the in-memory structures in the file systems determine the scalability, rather than the storage medium. Figure 1(b) shows that all file systems, except `XFS`, show a marginal performance difference among `RAMDISK`, `SSD`, and `HDD` at 80-core. In this case, performance with `XFS` is largely affected by the storage medium since `XFS` mostly waits for flushing journal data to disk due to its heavy metadata update operations.

Fine-grained locks often impede scalability. We may assume that the worst performing file system, `btrfs`, will be mostly in idle state, since it is waiting for events from the storage. On the contrary, Figure 1(c) shows that 67% of CPU time is spent in the kernel mode for `btrfs`. In particular, `btrfs` spends 47% of CPU time on synchronization to update its root node. In a common path without any contention, `btrfs` executes at least 10 atomic instructions to acquire a single B-tree node lock (`btrfs_tree_lock()`) and it must acquire locks of all interim nodes from a leaf to the root. If multiple readers or writers are contending to lock a node, each thread *retries* this process. Under heavy contention, it is typical to retry *a few hundreds times* to lock a single node. These frequent, concurrent accesses to synchronization objects result in a performance collapse after 4-core, as there is no upper bound on atomic instructions for updating the root node.

Subtle contentions matter. Figure 1(a) shows another anomaly with $\texttt{ext4}_{\texttt{NJ}}$ (i.e., `ext4` with no journaling) performing 44% *slower* than `ext4` itself. We found that two independent locks (i.e., a spinlock for journaling and a mutex for directory update) interleave in an unintended fashion. Upon `create()`, `ext4` first hits the journal spinlock (`start_this_handle()`) for metadata consistency and then hits the parent directory mutex (`path_openat()`) to add a new inode to its parent directory. In `ext4`, the serialization coordinated by the journal spinlock incurs little contention while attempting to hold the directory mutex. On the contrary, the contending directory mutex in $\texttt{ext4}_{\texttt{NJ}}$ results in expensive side-effects, such as sleeping on a waiting queue after a short period of opportunistic spinning.

Speculating scalability is precarious. The Exim case study shows that it is difficult for application developers or even file systems developers to speculate on the scalability of file system implementations. To identify such scalability problems in file systems, our community needs a proper benchmark suite to constantly evaluate and guide the design of file systems for scalability.

Type	Mode	Operation	Sharing Level		
			Low	Medium	High
Data	Read	Block read	✓	✓	✓
	Write	Overwrite	✓	✓	-
		Append	✓	-	-
		Truncate	✓	-	-
		Sync	✓	-	-
Meta	Read	Path name read	✓	✓	✓
		Directory list	✓	✓	-
	Write	Create	✓	✓	-
		Unlink	✓	✓	-
		Rename	✓	✓	-

Table 1: Coverage of the FxMark microbenchmarks. FxMark consists of 19 microbenchmarks that we categorize based on four criteria: data types, modes, operations, and sharing levels that are accordingly represented in each column on the table.

3 FxMark Benchmark Suite

There are 19 microbenchmarks in FxMark that are designed for systematically identifying scalability bottlenecks, and three well-known I/O-intensive application benchmarks to reason about the scalability bottlenecks in I/O-intensive applications.

3.1 Microbenchmarks

Exploring and identifying scalability bottlenecks *systematically* is difficult. The previous studies [21, 22, 32–34] on manycore scalability show that most applications are usually stuck with a few bottlenecks, and resolving them reveals the next level of bottlenecks.

To identify scalability problems in file system implementations, we designed microbenchmarks stressing seven different components of file systems: (1) path name resolution, (2) page cache for buffered I/O, (3) inode management, (4) disk block management, (5) file offset to disk block mapping, (6) directory management, and (7) consistency guarantee mechanism.

Table 1 illustrates the way FxMark categorizes each of these 19 microbenchmarks with varying stressed data types (i.e., file data or file system metadata), a related file system operation (e.g., `open()`, `create()`, `unlink()`, etc.), and its contention level (i.e., low, medium and high).

A higher contending level means the microbenchmark shares a larger amount of common code with the increasing number of cores, marked as *sharing level* for clarity.

For reading data blocks, FxMark provides three benchmarks based on the sharing level: (1) reading a data block in the *private file* of a benchmark process (low), (2) reading a *private data block* in the *shared file* among benchmark processes (medium), and (3) reading the *same data block* in the shared file (high). As a convention, we denote each microbenchmark with four letters representing type, mode, etc. For instance, we denote three previous examples with `DRBL` (i.e., Data, Read, Block read, and Low), `DRBM`, and `DRBH`, respectively. Table 2 shows a detailed description of each benchmark, including its core operation and expected contention.

To measure the scalability behavior, each benchmark runs its file system-related operations as a separate process pinned to a core; for example, each benchmark runs up to 80 physical cores to measure the scalability characteristics. Note that we use processes instead of threads, to avoid a few known scalability bottlenecks (e.g., allocating file descriptors and virtual memory management) in the Linux kernel [21, 33]. FxMark modifies the CPU count using CPU hotplug mechanism [76] in Linux. To minimize the effect of NUMA, CPU cores are assigned on a per socket basis; for example, in the case of 10 cores per socket, the first 10 CPU cores are assigned from the first CPU socket and the second 10 CPU cores are assigned from the second CPU socket. To remove interference between runs, FxMark formats a testing file system and drops all memory caches (i.e., page, inode, and dentry caches) before each run.

3.2 Application Benchmarks

Although a microbenchmark can precisely pinpoint scalability bottlenecks in file system components, scalability problems in applications might be relevant to only some of the bottlenecks. In this regard, we chose three application scenarios representing widely-used I/O-intensive tasks: mail server, NoSQL database, and file server.

Mail server. Exim is expected to linearly scale with the number of cores, at least from the application's perspective. But as Figure 1 shows, Exim does not scale well even after removing known scalability bottlenecks [21]. To further mitigate scalability bottlenecks caused by the Linux kernel's process management, we removed one of the two process invocations during the message transfer in Exim.

NoSQL database. RocksDB is a NoSQL database and storage engine based on log-structured merge-trees (LSM-trees) [43, 73]. It maintains each level of the LSM-tree as a set of files and performs multi-threaded compaction for performance, which will eventually determine the write performance. We use `db_bench` to benchmark RocksDB using the `overwrite` workload with disabled compression, which overwrites randomly generated key/value-pairs.

File server. To emulate file-server I/O-activity, we use the DBENCH tool [5]. The workload performs a sequence of create, delete, append, read, write, and attribute-change operations, with a specified number of clients processing the workload in parallel.

4 Diagnosis Methodology

4.1 Target File Systems

We chose four widely-used, disk-based file systems and one memory-based file system: `ext4`, `XFS`, `btrfs`, `F2FS`, and `tmpfs`.

T	M	Name	Operation	Description	Expected Contention
DATA	READ	DRBL	`pread("$ID/data",b,4K,0)`	Read a block in a private file	None
		DRBM	`pread("share",b,4K,$ID*4K)`	Read a private block in a shared file	Shared file accesses
		DRBH	`pread("share",b,4K,0)`	Read a shared block in a shared file	Shared block accesses
	WRITE	DWOL	`pwrite("$ID/data",b,4K,0)`	Overwrite a block in a private file	None
		DWOM	`pwrite("share",b,4K,$ID*4K)`	Overwrite a private block in a shared file	Updating inode attributes (e.g., `m_time`)
		DWAL	`append("$ID/data",b,4K)`	Append a block in a private file	Disk block allocations
		DWTL	`truncate("$ID/data",4K)`	Truncate a private file to a block	Disk block frees
		DWSL	`pwrite("$ID/data",b,4K,0)` `fsync("$ID/data")`	Synchronously overwrite a private file	Consistency mechanism (e.g., journaling)
META	READ	MRPL	`close( open("$ID/0/0/0/0"))`	Open a private file in five-depth directory	Path name look-ups
		MRPM	`close( open("$R/$R/$R/$R/$R"))`	Open an arbitrary file in five-depth directory	Path name look-ups
		MRPH	`close( open("0/0/0/0/0"))`	Open the same file in five-depth directory	Path name look-ups
		MRDL	`readdir("$ID/")`	Enumerate a private directory	None
		MRDM	`readdir("share/")`	Enumerate a shared directory	Shared directory accesses
	WRITE	MWCL	`create("$ID/$N")`	Create an empty file in a private directory	Inode allocations
		MWCM	`create("share/$ID.$N")`	Create an empty file in a shared directory	Inode allocations and insertions
		MWUL	`unlink("$ID/$N")`	Unlink an empty file in a private directory	Inode frees
		MWUM	`unlink("share/$ID.$N")`	Unlink an empty file in a shared directory	Inode frees and deletions
		MWRL	`rename("$ID/$N","$ID/$N.2")`	Rename a private file in a private directory	None
		MWRM	`rename("$ID/$N","share/$ID.$N")`	Move a private file to a shared directory	Insertions to the shared directory

NOTE. `$ID`: a unique ID of a test process `b`: a pointer to a memory buffer `$R`: a random integer between 0 and 7 `$N`: a test iteration count

Table 2: Microbenchmarks in FxMark. Each benchmark is identified by four letters based on type (marked as T), mode (marked as M), operation, and sharing level, as described in Table 1. Each microbenchmark is designed to stress specific building blocks of modern file systems (e.g., journaling and dcache) to evaluate their scalability on manycore systems.

Ext4 is arguably the most popular, mature Linux file system. It inherits well-proven features (e.g., bitmap-based management of inodes and blocks) from Fast File System (FFS) [69]. It also implements modern features such as extent-based mapping, block group, delayed allocation of disk blocks, a hash tree representing a directory, and journaling for consistency guarantee [27, 42, 67]. For journaling, ext4 implements write-ahead logging (as part of JBD2). We use ext4 with journaling in ordered mode and without it, marked as ext4 and $\texttt{ext4}_{\texttt{NJ}}$, to see its effect on file system scalability.

XFS is designed to support very large file systems with higher capacity and better performance [85]. XFS incorporates B+ trees in its core: inode management, free disk block management, block mapping, directory, etc [83]. However, using B+ trees incurs huge bulk writes due to tree balancing; XFS mitigates this by implementing *delayed logging* to minimize the amount of journal writes: (1) logically log the operations rather than tracking physical changes to the pages and (2) re-log the same log buffer multiple times before committing [30, 31].

Btrfs is a copy-on-write (CoW) file system that represents everything, including file data and file system metadata, in CoW optimized B-trees [77]. Since the root node of such B-trees can uniquely represent the state of the entire file system [78], btrfs can easily support a strong consistency model, called version consistency [29].

F2FS is a flash-optimized file system [23, 60, 63]. It follows the design of a log-structured file system (LFS) [80] that generates a large sequential write stream [55, 70]. Unlike LFS, it avoids the wandering tree problem [15] by updating some of its metadata structures in-place.

Tmpfs is a memory-based file system that works without a backing storage [79]. It is implemented as a simple wrapper for most functions in the VFS layer. Therefore, its scalability behavior should be an ideal upper bound of the Linux file systems' performance that implements extra features on top of VFS.

4.2 Experimental Setup

We performed all of the experiments on our 80-core machine (8-socket, 10-core Intel Xeon E7-8870) equipped with 512 GB DDR3 DRAM. For storage, the machine has both a 1 TB SSD (540 MB/s for reads and 520 MB/s for writes) and a 7200 RPM HDD with a maximum transfer speed of 160 MB/s. We test with Linux kernel version 4.2-rc8. We mount file systems with the noatime option to avoid metadata update for read operations. RAMDISK is created using tmpfs.

4.3 Performance Profiling

While running each benchmark, FxMark collects the CPU utilization for sys, user, idle, and iowait to see a microscopic view of a file system reaction to stressing its components. For example, a high idle time implies that the operation in a microbenchmark impedes the scalability behavior (e.g., waiting on locks); a high iowait time implies that a storage device is a scalability bottleneck. For further analysis, we use perf [7] to profile the entire system and to dynamically probe a few interesting functions (e.g., mutex_lock(), schedule(), etc.) that likely induce such idle time during file system operations.

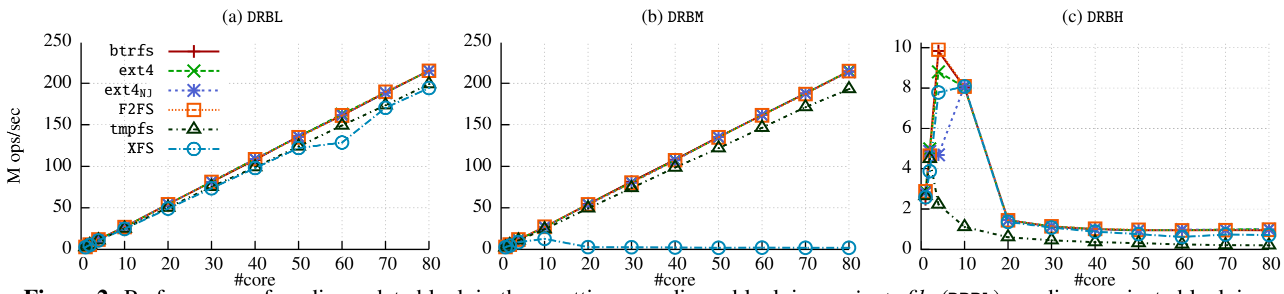


Figure 2: Performance of reading a data block in three settings; reading a block in a *private file* (DRBL), reading a private block in a *shared file* (DRBM), and reading a *shared block* in a shared file (DRBH). All file systems scale linearly in DRBL. XFS fails to scale in DRBM because of the per-inode read/write semaphore. In DRBH, all file systems show their peak performance around 10-core because of contending per-page reference counters in VFS (§5.1.1).

5 Microbenchmark Analysis

In this section, we first describe the analysis results of each microbenchmark in buffered I/O mode starting from simple operations on file data (§5.1) and going to more complicated file system metadata operations (§5.2). Then, we analyze the results of file data operation in direct I/O mode (§5.3). Lastly, we analyze how the performance of the storage medium affects scalability (§5.4).

5.1 Operation on File Data

5.1.1 Block Read

We start from the simplest case: each test process reads a block in its respective private file (DRBL). As Figure 2 shows, all test file systems scale nearly linearly.

However, when each test process reads a private block in the *shared file* (DRBM), we observe that XFS's performance collapses near 10-core. Unlike other file systems, it spends 40% of execution and idle time at per-inode read/write semaphores (`fs/xfs/xfs_file.c:329`) when running on 80 cores. XFS relies heavily on per-inode read/write semaphores to orchestrate readers and writers in a fine-grained manner [85]. However, the performance degradation does not come from unnecessary waiting in the semaphores. At every `read()`, XFS acquires and releases a read/write semaphore of a file being accessed in shared mode (`down_read()` and `up_read()`). A read/write semaphore internally maintains a reader counter, such that every operation in shared mode updates the reader counter *atomically*. Therefore, concurrent read operations on a shared file in XFS actually perform atomic operations on a shared reader counter. This explains the performance degradation after 10 cores. In fact, at XFS's peak performance, the cycles per instruction (CPI) is 20 at 10-core, but increases to 100 at 20-core due to increased cache line contention on updating a shared reader counter.

For reading the same block (DRBH), all file systems show a performance collapse after 10-core. Also, the performance at 80-core is 13.34× (for tmpfs) lower than that on a single core. We found that more than 50% of the time is being spent on reference counter operations for the page cache. The per-page reference counting is used for concurrency control of a page in a per-inode page cache. The per-page reference counter is also updated using atomic operations so that all test processes contend to update the same cache line. In the case of ext4, the CPI is 14.3 at 4-core but increases to 100 at 20-core due to increased cache-coherence delays.

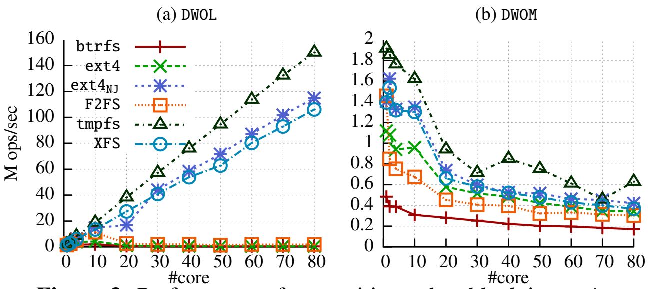


Figure 3: Performance of overwriting a data block in a *private file* (DWOL) and overwriting a *private block* in a shared file (DWOM). In DWOL, although they do not seem to have explicit contention, btrfs, ext4 and F2FS fail to scale due to their consistency mechanisms. Note that XFS is an exception among journaling file systems (see §5.1.2). In DWOM, all file systems fail to scale regardless of consistency mechanisms.

5.1.2 Block Overwrite

At first glance, overwriting a block in a private file (DWOL) should be an ideal, scalable operation; only the private file block and inode features such as last modified time need to be updated. However, as shown in Figure 3, ext4, F2FS, and btrfs fail to scale. In ext4, starting and stopping journaling transactions (e.g., `jbd2_journal_start()`) in JBD2 impedes its scalability. In particular, acquiring a read lock (`journal->j_state_lock`) and atomically increasing a counter value (`t_handle_count`) take 17.2% and 9.4% of CPU time, respectively. Unlike ext4, XFS scales well due to delayed logging [30, 31], which uses logical logging and re-logging to minimize the amount of journal write. In F2FS, write operations eventually trigger segment cleaning (or garbage collection) to reclaim invalid blocks. Segment cleaning freezes all file system operations for checkpointing of the file system status by holding the exclusive lock of a file system-wide read/write semaphore (`sbi->cp_rwsem`). btrfs is a CoW-based file system that never overwrites a physical block. It allo-

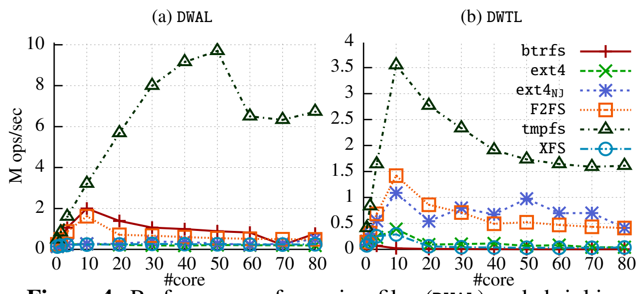


Figure 4: Performance of growing files (DWAL) and shrinking files (DWTL). None of the tested file systems scales.

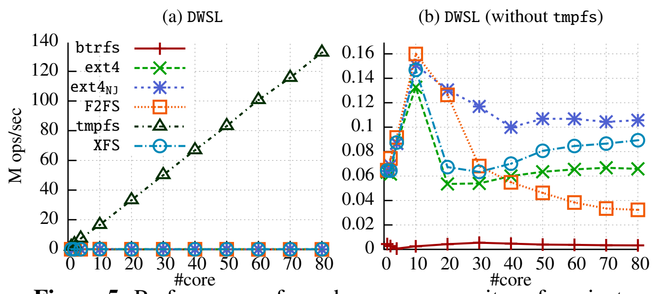


Figure 5: Performance of synchronous overwrites of a private file (DWSL). Only tmpfs ignoring fsync() scales.

cates a new block for every write operation so that its disk block allocation (i.e., updating its extent tree) becomes the sequential bottleneck.

When every test process overwrites each private block in the shared file (DWOM), none of the file systems scale. The common bottleneck is an inode mutex (inode->i_mutex). The root cause of this sequential bottleneck stems from file system-specific implementations, not VFS. None of the tested file systems are implementing a range-based locking mechanism, which is common in parallel file systems [81]. In fact, this is a serious limitation in scaling I/O-intensive applications like DBMS. The common practice of running multiple I/O threads is not effective when applications are accessing a shared file, regardless of the underlying file systems. Furthermore, it may incur a priority inversion between I/O and latency-sensitive client threads [24, 25], potentially disrupting the application's scalability behavior.

5.1.3 File Growing and Shrinking

For file growing and shirking, all disk-based file systems fail to scale after 10-core (DWAL in Figure 4). In F2FS, allocating or freeing disk blocks entails updating two data structures: SIT (segment information table) tracking block validity, and NAT (node address table) tracking inode and block mapping tables. Contention in updating SIT and NAT limits F2FS's scalability. When allocating blocks, checking disk free-space and updating the NAT consumes 78% of the execution time because of lock contentions (nm_i->nat_tree_lock). Similarly, there is another lock contention for freeing blocks to invalidate free blocks. This exhausts 82% of the execution time (sit_i->sentry_lock). In btrfs, when growing a file, the sequential bottleneck is checking and reserving free space (data_info->lock and delalloc_block_rsv->lock). When shrinking a file, btrfs suffers from updating an extent tree, which keeps track of the reference count of disk blocks: A change in reference counts requires updates all the way up to the root node.

In ext4 and XFS, the *delayed allocation* technique, which defers block allocation until writeback to reduce fragmentation of a file, is also effective in improving manycore scalability by reducing the number of block allocation operations. Because of this, the scalability bottleneck of ext4 and XFS is not the block allocation but rather their journaling mechanisms. About 76–96% of the execution time was spent on journaling. ext4 spends most of its execution time on manipulating JBD2 shared data structures (e.g., journal_j_flags) protected by spinlocks and atomic instructions. XFS spends most of its execution time waiting on flushing its log buffers. Upon truncating files, these file systems face the same performance bottleneck.

In tmpfs, checking the capacity limit (__vm_enough_memory()) becomes a scalability problem. As the used space approaches the capacity limit (at 50-core in this case), the checking takes a slow path for precise comparison of the remaining disk space. Tracking the used space by using a per-CPU counter scales up to 50-core, but fails to scale for more cores because of a contending spinlock in a per-CPU counter on the slow path to get a true value. When freeing blocks, updating per-cgroup page usage information using atomic decrements causes a performance collapse after 10-core.

5.1.4 File Sync Operation

When using fsync(), a file system synchronously flushes dirty pages of a file and disk caches. In this regard, all file systems can scale up to the limitation of the storage medium. Although we use memory as a storage backend, none of the file systems scale, except tmpfs, which ignores fsync() operations. Notice that fsync() on btrfs is significantly slower than other file systems (see Figure 5). Similar to §5.1.2, btrfs propagates a block update to its root node so a large number of metadata pages need to be written[2]. All file systems (except for tmpfs and btrfs) start to degrade after 10-core. That is due to locks protecting flush operations (e.g., journal->j_state_lock), which start contending after roughly 10-core.

[2] To minimize fsync() latency, btrfs maintains a special *log-tree* to defer checkpointing the entire file system until the log is full. In the case of fsync()-heavy workloads, like DWSL, the log quickly becomes full; therefore, checkpointing or updating the root node becomes a sequential bottleneck.

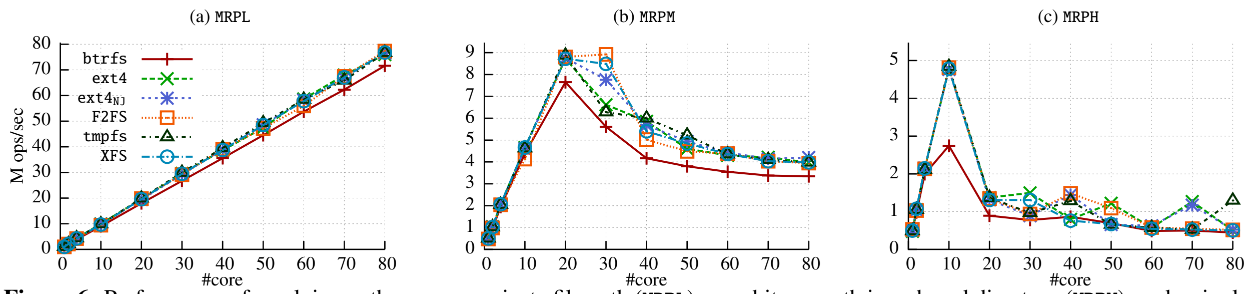


Figure 6: Performance of resolving path names; a private file path (MRPL), an arbitrary path in a shared directory (MRPM), and a single, shared path (MRPH). Surprisingly, resolving a single, common path name is the slowest operation (MRPH).

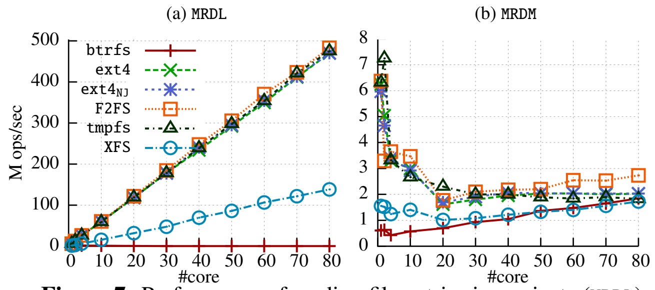


Figure 7: Performance of reading file entries in a private (MRDL) and a shared directory (MRDM). btrfs fails to scale in MRDL and none of the file systems scales in MRDM.

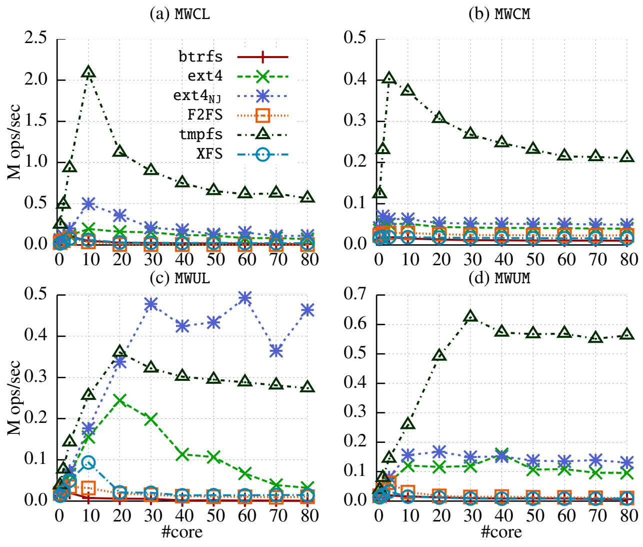


Figure 8: Performance of creating and deleting files in a private directory (i.e., MWCL and MWUL) and a shared directory (i.e., MWCM and MWUM).

5.2 Operation on File System Metadata

5.2.1 Path Name Resolution

The Linux kernel maintains a directory cache, or dcache, which caches dentry structures. Only when a dcache miss happens, the kernel calls the underlying file system to fill up the dcache. Since dcache hits are dominant in our microbenchmark, MRPL, MRPM, and MRPH stress the dcache operations implemented in the VFS layer. Thus there is little performance difference among file systems, as shown in Figure 6. Our experiment shows that (1) dcache is scalable up to 10-core if multiple processes attempt to resolve the occasionally shared path names (MRPM), and (2) contention on a shared path is so serious that resolving a single common path in applications becomes a scalability bottleneck. In MRPM and MRPH, a lockref in a dentry (i.e., dentry->d_lockref), which combines a spinlock and a reference count into a single locking primitive for better scalability [38], becomes a scalability bottleneck. Specifically, the CPI of ext4 in MRPH increases from 14.2 at 10-core to 20 at 20-core due to increased cache-coherence delays.

5.2.2 Directory Read

When listing a private directory (i.e., MRDL in Figure 7), all file systems scale linearly, except for btrfs. Ironically, the performance bottleneck of btrfs is the fine-grained locking. To read a file system buffer (i.e., extent_buffer) storing directory entries, btrfs first acquires read locks from a leaf, containing the buffer, to the root node of its B-tree (btrfs_set_lock_blocking_rw()); moreover, to acquire a read lock of a file system buffer, btrfs performs two read/write spinlock operations and six atomic operations for reference counting. Because such excessive synchronization operations increase cache-coherence delays, CPI in btrfs is 20.4× higher than that of ext4 at 80-core (20 and 0.98, respectively). XFS shows better scalability than btrfs even though its directory is represented as a B+-tree due to coarser-grained locking, i.e., per-directory locking.

Unexpectedly, listing the shared directory (i.e., MRDM in Figure 7) is not scalable in any of the file systems; the VFS holds an inode mutex before calling a file system-specific directory iteration function (iterate_dir()).

5.2.3 File Creation and Deletion

File creation and deletion performance are critical to the performance of email servers and file servers, which frequently create and delete small files. However, none of the file systems scale, as shown in Figure 8.

In tmpfs, a scalability bottleneck is adding and deleting a new inode in an inode list in a super block (i.e., sb->s_inodes). An inode list in a super block is protected by a system-wide (not file system-wide) spinlock (i.e., inode_sb_list_lock), so the spinlock becomes a performance bottleneck and incurs a performance col-

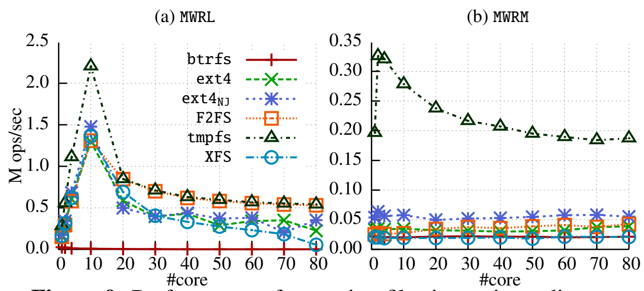


Figure 9: Performance of renaming files in a private directory (`MWRL`) and shared directory (`MWRM`). None of file systems scale.

lapse after 10-core. In fact, CPI at 10-core increases from 20 to 50 at 20-core due to shared cache line contention on the spinlock.

In `ext4`, inode allocation is a per-block group operation, so the maximum level of concurrency is the number of block groups (256 in our evaluation). But `ext4`'s policy to preserve spatial locality (e.g., putting files under the same directory to the same block group) limits the maximum concurrency. Upon file deletion, `ext4` first puts a deleted inode onto an orphan inode list in a super block (i.e., `sbi->s_orphan`), which is protected by a per-file-system spinlock. This list ensures that inodes and related resources (e.g., disk blocks) are freed even if the kernel crashes in the middle of the delete. Adding an inode to an orphan list is a sequential bottleneck.

Like `ext4`, `XFS` also maintains inodes per-block group (or allocation group). But, unlike `ext4`, a B+-tree is used to track which inode numbers are allocated and freed. Inode allocation and free incurs changes in the B+-tree and such changes need to be logged for consistency. So, journaling overhead waiting for flushing log buffers is the major source of bottlenecks (90% of time).

In `btrfs`, files and inodes are stored in the file system B-tree. Therefore, file creation and deletion incur changes in the file system B-tree, and such changes eventually need to be propagated to the root node. Similar to other write operations, updating the root node is again a sequential bottleneck. In fact, between 40% and 60% of execution time is spent contending to update the root node.

In `F2FS`, performance characteristics of file creation and deletion are similar to those of appending and truncating data blocks in §5.1.3. The reason for this, in the case of deletion, is the contention in updating the SIT (segment info table), which keeps track of blocks in active use. In fact, up to 85% of execution time is spent on contending for updating the SIT. During create, contention within the NAT (node address table) is the main reason for the performance collapse.

When creating and deleting files in a shared directory, additional contention updating the shared directory is noticeable (see `MWCM` and `MWUM` in Figure 8). Like `MRDM`, Linux `VFS` holds a per-directory mutex while creating and deleting files.

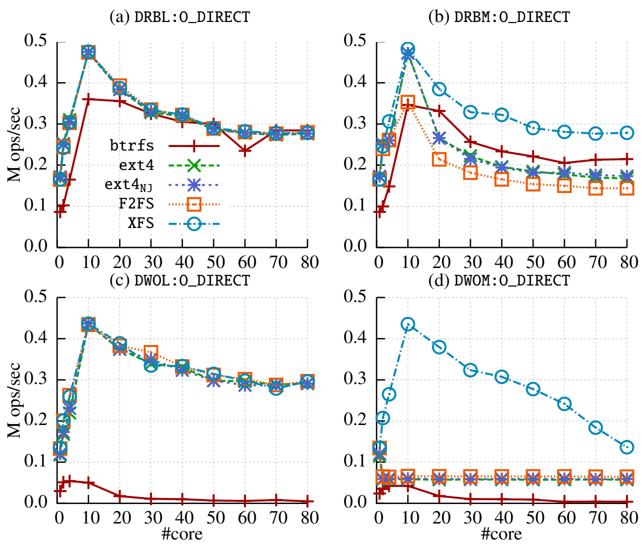


Figure 10: Performance of file data operations in a direct I/O mode on a single `RAMDISK`. Performance of all tested file systems are saturated or declining after 10-core, at which a single `RAMDISK` becomes a bottleneck. For write operations, `btrfs` suffers from its heavy b-tree operations the same as in the buffered mode. For `DWOM:O_DIRECT`, only `XFS` scales because it holds a shared lock for an `inode` while the others hold an inode mutex (`inode->i_mutex`).

5.2.4 File Rename

As Figure 9 shows, rename is a system-wide sequential operation regardless of the sharing level. In `VFS`, multiple readers optimistically access the `dcache` through `rename_lock`, concurrently with multiple writers, and then later, each reader checks if the sequence number is the same as the one at the beginning of the operation. If sequence numbers do not match (i.e., there were changes in dentries), the reader simply retries the operation. Therefore, a rename operation needs to hold a write lock (i.e., `write_seqlock()` on `rename_lock`), which turns out to be the bottleneck in our benchmark. In `MWRL`, on average, 84.7% of execution time is spent waiting to acquire the `rename_lock`. This scalability bottleneck is a serious limitation for applications that concurrently perform renaming of multiple files, like Exim.

5.3 Scalability in a Direct I/O Mode

Performance in direct I/O mode is critical for many I/O-intensive applications. To understand the scalability behavior, we ran microbenchmarks on file data operations from §5.1 in direct I/O mode (`O_DIRECT`).

When each test process reads a block in its respective private file (`DRBL:O_DIRECT`), there is no apparent contention at file system so the storage device (i.e., a single `RAMDISK`) becomes the bottleneck. When more than 10 cores are used (i.e., more than two sockets are involved in our experimental setup), performance gradually degrades due to the NUMA effect. When reading a private block

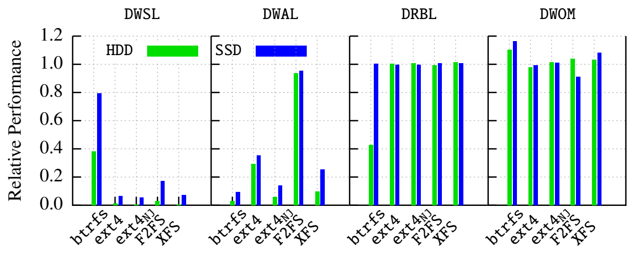


Figure 11: Relative performance of SSD and HDD to RAMDISK at 80-core. For buffered reads (e.g., DRBL) and contending operations (e.g., DWOM), the performance of the storage medium is not the dominant factor of applications' I/O performance.

of the shared file (DRBM:O_DIRECT), XFS shows around 20-50% higher performance than the other file systems. The performance difference comes from the different locking mechanism of the shared file for *w*riting. As discussed in §5.1.2, the file systems should lock the shared file before reading the disk blocks as the dirty pages of a file should be consistently written to that shared file. While writing dirty pages of a file in a direct I/O mode, XFS holds the shared lock of a file but others holds the inode mutex (inode->i_mutex). Thus, read operations of the other file systems are serialized by the inode mutex.

For write operations, btrfs suffers from its heavy b-tree operations regardless of the contention level. When writing private files (DWOL:O_DIRECT), the storage device is the bottleneck as same as DRBL:O_DIRECT. When writing a private block of the shared file (DWOM:O_DIRECT), only XFS scales up to 10-core. File systems other than XFS serialize concurrent write operations by holding the inode mutex. In contrast, since XFS holds the shared lock while writing disk blocks, write operations for the shared file can be concurrently issued.

The scalability bottleneck in accessing the shared file is a serious limitation for applications such as database systems and virtual machines, where large files (e.g., database table or virtual disk) are accessed in a direct I/O mode by multiple I/O threads.

5.4 Impact of Storage Medium

To understand how the performance of the storage medium affects the scalability behavior, we ran FxMark on SSD and HDD, and compared their performance at 80-core in Figure 11. For synchronous write operations (e.g., DWSL) or operations incurring frequent page cache misses (e.g., DWAL), the bandwidth of the storage medium is a dominant factor. However, for buffered reads (e.g., DRBL) or contending operations (e.g., DWOM), the impact of the storage medium is not dominant. With larger memory devices, faster storage mediums (e.g., NVMe), and increasing core counts in modern computers, it is important to understand, measure, and thus improve the scalability behavior of file systems.

6 Application Benchmarks Analysis

In this section, we explain the scalability behavior of three applications on various file systems backed by memory.

Exim. After removing the scalability bottleneck in Exim (see §3.2), it linearly scales up to 80-core (tmpfs in Figure 12(a)). With the optimized Exim, ext4 scales the most, followed by $\texttt{ext4}_{\texttt{NJ}}$, but it is still 10× slower than tmpfs. Since Exim creates and deletes small files in partitioned spool directories, performance bottlenecks in each file system are equivalent to both MWCL and MWUL (see §5.2.3).

RocksDB. As Figure 12(b) illustrates, RocksDB scales fairly well for all file systems up to 10 cores but either flattens out or collapses after that. The main bottleneck can be found in RocksDB itself, synchronizing compactor threads among each other. Since multiple compactor threads concurrently write new merged files to disk, the behavior and performance bottleneck in each file system is analogous to DWAL (see §5.1.2).

DBENCH. Figure 12(c) illustrates the DBENCH results, which do not scale linearly with increasing core count for any of the file systems. This happens because DBENCH reads, writes, and deletes a large number of files in a *shared directory*. This is similar to our microbenchmarks MWCM and MWUM (§5.1.3). tmpfs suffers for two reasons: look-ups and insertions in the page cache and reference counting for the dentry of the directory.

7 Summary of Benchmarks

We found 25 scalability bottlenecks in five widely-used file systems, as summarized in Figure 13 and Table 3. Some of the bottlenecks (e.g., inode list lock) are also found in recent literature [21, 56, 57]. In our opinion, this is the most important first step to scale file systems. We draw the following observations, to which I/O-intensive application developers must pay close attention.

High locality can cause performance collapse. Maintaining high locality is believed to be the golden rule to improve performance in I/O-intensive applications because it increases the cache hit ratio. But when the cache hit is dominant, the *scalability of cache hits* does matter. We found such performance collapses in the page cache and dentry cache in Linux file systems. [§5.1.1, §5.2.1]

Renaming is system-wide sequential. rename() is commonly used in many applications for transactional updates [46, 71, 75]. However, we found that rename() operations are completely serialized at a system level in Linux for consistent updates of the dentry cache. [§5.2.4]

Even read operations on a directory are sequential. All operations (e.g., readdir(), create(), and unlink()) on a shared directory are serialized through a per-directory mutex (inode->i_mutex) in Linux. [§5.2.2, §5.2.3, §5.2.4]

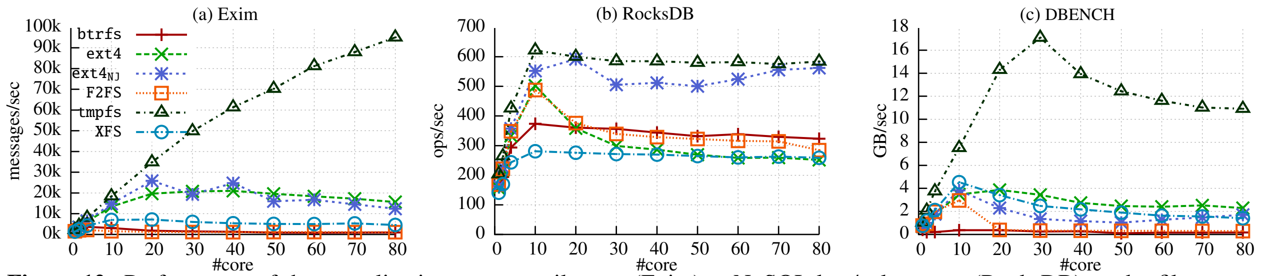


Figure 12: Performance of three applications—an email sever (Exim), a NoSQL key/value store (RocksDB), and a file server (DBENCH)—on various file systems backed by memory.

FS	Bottleneck	Sync. object	Scope	Operation
VFS	Rename lock	`rename_lock`	System	`rename()`
	`inode` list lock	`inode_sb_list_lock`	System	File creation and deletion
	Directory access lock	`inode->i_mutex`	Directory	All directory operations
	Page reference counter	`page->_count`	Page	Page cache access
	`dentry lockref` [38]	`dentry->d_lockref`	`dentry`	Path name resolution
`btrfs`	Acquiring a write lock for a B-tree node	`btrfs_tree_lock()`	File system	All write operations
	Acquiring a read lock for a B-tree node	`btrfs_set_lock_blocking_rw()`	File system	All read operations
	Checking data free space	`data_info->lock`	File system	File append
	Reserving data blocks	`delalloc_block_rsv->lock`	File system	File append
	File write lock	`inode->i_mutex`	File	File write
`ext4`	Acquiring a read lock for a journal	`journal->j_state_lock`	Journal	Heavy metadata update operations
	Atomic increment of a transaction counter	`t_handle_count`	Journal	Heavy metadata update operations
	Orphan `inode` list	`sbi->s_orphan`	File system	File deletion
	Block group lock	`bgl->locks`	Block group	File creation and deletion
	File write lock	`inode->i_mutex`	File	File write
`F2FS`	Single-threaded writing	`sbi->cp_rwsem`	File system	File write
	SIT (segment information table)	`sit_i->sentry_lock`	File system	File write
	NAT (node address table)	`nm_i->nat_tree_lock`	File system	File creation, deletion, and write
	File write lock	`inode->i_mutex`	File	File write
`tmpfs`	Cgroup page reference counter	`counter->count`	cgroup	File truncate
	Capacity limit check (per-CPU counter)	`sbinfo->used_blocks`	File system	File write near disk full
	File write lock	`inode->i_mutex`	File	File write
`XFS`	Journal writing	`log->l_icloglock`	File system	Heavy metadata update operations
	Acquiring a read lock of `XFS inode`	`ip->i_iolock`	File	File read
	File write lock	`inode->i_mutex`	File	File write

Table 3: The identified scalability bottlenecks in tested file systems with FxMark.

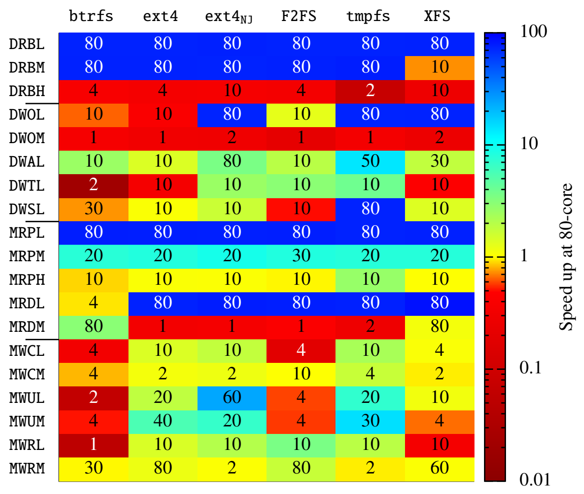
A file cannot be concurrently updated. All of the tested file systems hold an exclusive lock for a file (`inode->i_mutex`) during a write operation. This is a critical bottleneck for high-performance database systems allocating a large file but not maintaining the page cache by themselves (e.g., PostgreSQL). Even in the case of using direct I/O operations, this is the critical bottleneck for both read and write operations as we discussed in §5.3. To the best of our knowledge, the only way to concurrently update a file in Linux is to update it on a `XFS` partition with `O_DIRECT` mode, because `XFS` holds a shared lock of a file for all IO operations in direct I/O mode [26]. [§5.1.2]

Metadata changes are not scalable. For example, creating, deleting, growing, and shrinking a file are not scalable in Linux. The key reason is that existing consistency mechanisms (i.e., journaling in `ext4` and `XFS`, copy-on-write in `btrfs`, and log-structured writing in `F2FS`) are not designed with manycore scalability in mind. [§5.1.3, §5.2.3]

Overwriting could be as expensive as appending. Many I/O-optimized applications such as MySQL and PostgreSQL pre-allocate file blocks of heavily updated files (e.g., log files) and overwrite them instead of growing files dynamically to reduce the overhead of changing its metadata [64, 71, 84]. But in non-in-place update file systems such as `btrfs` and `F2FS`, overwrite is written in a new place; it involves freeing and allocating disk blocks, updating the inode block map, etc. and thus is as expensive as append operations. [§5.1.2]

Scalability is not portable. Some file systems have peculiar scalability characteristics for some operations; a single file read of multiple threads is not scalable in `XFS` due to the scalability limitation of Linux's read/write semaphore; in `F2FS`, a checkpointing operation caused by segment cleaning freezes entire file system operations so scalability will be seriously hampered under write-heavy workloads with low free space; enumerating directories in `btrfs` is not scalable because of too frequent atomic operations. The scalability of an I/O-intensive application is very file system-specific. [§5.1.1, §5.1.2, §5.2.2]

Non-scalability often means wasting CPU cycles. In file systems, concurrent operations are coordinated mainly by locks. Because of spinning of spinlock and optimistic spinning of blocking locks, non-scalable file systems tend



	btrfs	ext4	ext4$_{NJ}$	F2FS	tmpfs	XFS
DRBL	80	80	80	80	80	80
DRBM	80	80	80	80	80	10
DRBH	4	4	10	4	2	10
DWOL	10	10	80	10	80	80
DWOM	1	1	2	1	1	2
DWAL	10	10	80	10	50	30
DWTL	2	10	10	10	10	10
DWSL	30	10	10	10	80	10
MRPL	80	80	80	80	80	80
MRPM	20	20	20	30	20	20
MRPH	10	10	10	10	10	10
MRDL	4	80	80	80	80	80
MRDM	80	1	1	1	2	80
MWCL	4	10	10	4	10	4
MWCM	4	2	2	10	4	2
MWUL	2	20	60	4	20	10
MWUM	4	40	20	4	30	4
MWRL	1	10	10	10	10	10
MWRM	30	80	2	80	2	60

Figure 13: Summary of manycore scalability of tested file systems. The color represents the relative speed over a single-core performance of a file system with a specific microbenchmark. The number in a cell denotes the core count at peak performance.

to consume more CPU cycles to alleviate the contention. In our benchmarks, about 30-90% of CPU cycles are consumed for synchronization. [§5.1.3, §5.2.2]

8 Discussion

The next question is whether traditional file system designs can be used and implemented in a scalable way. It is difficult to answer conclusively. At a high level, the root causes of the scalability problems of file systems are not different from those of OS scalability in previous studies [21, 22, 32–34]: shared cache line contention, reference counting, coarse-grained locking, etc. But the devil is in the details; some are difficult to fix with known techniques and lead us to revisit the core design of file systems.

Consistency mechanisms need to be scalable. All three consistency mechanisms, journaling (`ext4` and `XFS`), log-structured writing (`F2FS`), and copy-on-write (`btrfs`), are scalability bottlenecks. We think these are caused by their inherent designs so that our community needs to revisit consistency mechanisms from a scalability perspective.

In journaling file systems, committing a journal transaction guarantees a filesystem-wide consistency. To this end, `ext4`, for example, maintains a single running journal transaction, so accessing the journal transaction becomes a scalability bottleneck. Scalable journaling is still an unexplored area in the context of file systems, though there are some studies in the database field [54, 66, 90, 91].

In the case when copy-on-write techniques are combined with self-balancing index structures (e.g., B-tree) like `btrfs`, such file systems are very fragile to scalability; a leaf node update triggers updates of all interim nodes to the root so that write locks of all nodes should be acquired. Moreover, two independent updates should contend for acquiring locks of common ancestors. Besides locking overhead, this could result in a deadlock if two updates should be coordinated by other locks. We suspect this is the reason why `btrfs` uses the retry-based locking protocol to acquire a node lock (`btrfs_tree_lock()`). Parallelizing a CoW file system by extending the current B-tree scheme (e.g., LSM tree) or using non-self-balancing index structures (e.g., radix tree or hash table) is worth further research.

To our best knowledge, all log-structured file systems, including `F2FS`, `NILFS2` [3], and `UBIFS` [4], adopt single-threaded writing. By nature, log-structured file systems are designed to create a large sequential write stream, while metadata updates should be issued after writing file data for consistency guarantee. Multi-headed log-structured writing schemes are an unexplored area in the context of file systems, while some techniques are proposed at the storage device level [28, 58].

Spatial locality still needs to be considered. One potential solution to parallelizing file systems is partitioning. To see its feasibility, we modified Exim and RocksDB to run on multiple file system partitions for spool directories and database files, respectively. We set up 60 file system partitions, the maximum allowed on a disk, to spread files in 60 ways. Our results on `RAMDISK` and `HDD` show its potential and limitations at the same time. We see a significant performance improvement on `RAMDISK` (Figure 14). It confirms that the reduced contentions in a file system can improve the scalability. However, the RocksDB results on `HDD` also show its limitation (Figure 15). In all file systems except for `F2FS`, the partitioned case performs worse, as partitioning ruins spatial locality. But `F2FS` performs equally well in both cases; because the log-structured writing scheme of `F2FS` always issues bulk sequential write for file data, the impact of partitioning is negligible. The above example shows the unique challenges in designing scalable file systems. Optimizing for storage devices based on their performance characteristics and achieving consistency guarantees in a scalable fashion will be critical in file systems.

File system-specific locking schemes in VFS. The scalable performance of locking strategies, such as granularity and types of lock, is dependent on data organization and management. The current locking schemes enforced in `VFS` will become obsolete as storage devices and file systems change. For example, the inode mutex for directory accesses, currently enforced by the `VFS`, should be flexible enough for each file system to choose proper, finer-grained locking.

Reference counting still does matter. We found scalability problems in the reference counters of various file system objects (e.g., `page`, `dentry`, and `XFS inode` struc-

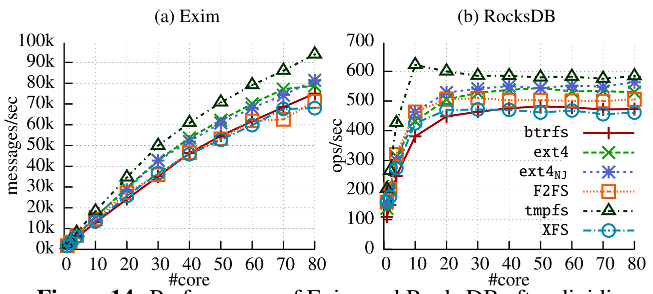


Figure 14: Performance of Exim and RocksDB after dividing each file system to 60 partitions on RAMDISK.

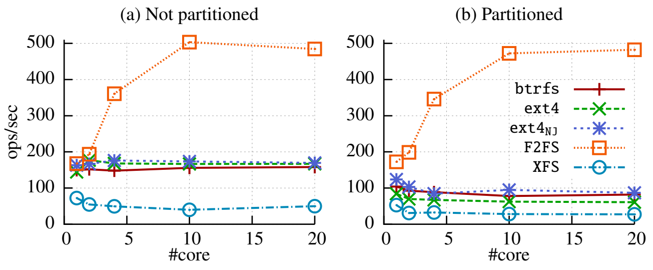


Figure 15: Performance of RocksDB with and without partitioning HDD. For clarity, results up to 20-core are presented because the performance of all file systems is saturated after that. Except for F2FS, all other file systems fail to scale after partitioning and perform around 50% of the original performance. The impact of partitioning in F2FS is negligible because the log-structured writing in F2FS always generates large sequential write for file data.

tures). Many previous studies proposed scalable reference counting mechanisms, including clustered object [61], SNIZ [40], sloppy counter [21], Refcache [33], and Linux per-CPU counter [39], but we identified that they are not adequate for file system objects for two reasons. First, their space overhead is proportional to the number of cores since they speed up by separating cache lines per-core. Such space overhead is especially problematic to file system objects, which are many in number and small in size (typically a few tens or hundreds of bytes). For example, in our 80-core machine, the space required for per-core counters per `page` is 5× larger than the `page` structure itself (320 bytes vs. 64 bytes). Second, getting the true value is not scalable but immediate. Recall that we found the `reader` counter problem at the R/W semaphore used in the XFS `inode` (§5.1.1). If getting the number of `reader` is not immediate, writers waiting for readers can starve.

9 Related Work

Benchmarking file systems. Due to the complexity of file systems and interactions among multiple factors (e.g., page cache, on-disk fragmentation, and device characteristics), file system benchmarks have been criticized for decades [12, 87–89]. Popular file system benchmarks, such as FIO [1] and iozone [2], mostly focus on measuring bandwidth and IOPS of file system operations varying I/O patterns. In contrast, recently developed benchmarks focus on a specific system (e.g., smartphones [59]) or a particular component in file systems (e.g., block allocation [47]). Along this line, FxMark focuses only on manycore scalability of file systems.

Scaling operating systems. To improve the scalability of OS, researchers have been optimizing existing OSes [20–22, 32, 33] or have been building new OSes based on new design principles [16, 34]. However, previous studies used memory-based file systems to opt out of the effect of I/O operations. In Arrakis [74], since file system service is a part of applications, its manycore scalability solely depends on each application.

Scaling file systems. The Linux kernel community has made a steady effort to improve the scalability of the file system by mostly reducing lock contentions [35, 36, 65]. Hare [45] is a scalable file system for non-cache-coherent systems. But it does not provide durability and crash consistency, which were significant performance bottlenecks in our evaluation. ScaleFS [41] extends a scalable in-memory file system to support consistency by using operation log on an on-disk file system. SpanFS [57] adopts partitioning techniques to reduce lock contentions. But how partitioning affects performance in a physical storage medium such as SSD and HDD is not explored.

Optimizing the storage stack for fast storage. As storage devices become dramatically faster, there are research efforts to make storage stacks more scalable. Many researchers made efforts to reduce the overhead and latency of interrupt handling in the storage device driver layer [13, 82, 92, 93]. At the block layer, Bjørling et al. [18] address the scalability of the Linux block layer and propose a new Linux block layer, which maintains a per-core request queue.

10 Conclusion

We performed a comprehensive analysis of the manycore scalability of five widely-deployed file systems using our FxMark benchmark suite. We observed many unexpected scalability behaviors of file systems. Some of them lead us to revisit the core design of traditional file systems; in addition to well-known scalability techniques, scalable consistency guarantee mechanisms and optimizing for storage devices based on their performance characteristics will be critical. We believe that our analysis results and insights can be a starting point toward designing scalable file systems for manycore systems.

11 Acknowledgment

We thank the anonymous reviewers, and our shepherd, Angela Demke Brown, for their helpful feedback. This research was supported by the NSF award DGE-1500084, ONR under grant N000141512162, DARPA Transparent Computing program under contract No. DARPA-15-15-TC-FP-006, ETRI MSIP/IITP[B0101-15-0644], and NRF BSRP/MOE[2015R1A6A3A03019983].

References

[1] Flexible I/O Tester. https://github.com/axboe/fio.

[2] IOzone Filesystem Benchmark. http://www.iozone.org/.

[3] NILFS – Continuous Snapshotting Filesystem. http://nilfs.sourceforge.net/en/.

[4] UBIFS – UBI File-System. http://www.linux-mtd.infradead.org/doc/ubifs.html.

[5] DBENCH, 2008. https://dbench.samba.org/.

[6] MongoDB, 2009. https://www.mongodb.org/.

[7] perf: Linux profiling with performance counters , 2014. https://perf.wiki.kernel.org/index.php/Main_Page.

[8] Exim Internet Mailer, 2015. http://www.exim.org/.

[9] SAP HANA, 2015. http://hana.sap.com/abouthana.html/.

[10] MariaDB, 2015. https://mariadb.org/.

[11] VoltDB, 2015. https://voltdb.com/.

[12] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating Realistic Impressions for File-system Benchmarking. *Trans. Storage*, 5(4):16:1–16:30, Dec. 2009.

[13] I. Ahmad, A. Gulati, and A. Mashtizadeh. vIC: Interrupt coalescing for virtual machine storage device IO. In *Proceedings of the 2011 ATC Annual Technical Conference (ATC)*, Portland, OR, June 2011.

[14] P. Alcorn. Samsung Releases New 12 Gb/s SAS, M.2, AIC And 2.5" NVMe SSDs: 1 Million IOPS, Up To 15.63 TB, 2013. http://www.tomsitpro.com/articles/samsung-sm953-pm1725-pm1633-pm1633a,1-2805.html.

[15] Artem B. Bityutskiy. JFFS3 design issues. http://linux-mtd.infradead.org/tech/JFFS3design.pdf.

[16] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[17] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium*, Ottawa, Canada, June 2003.

[18] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)*, June 2013.

[19] S. Borkar. Thousand Core Chips: A Technology Perspective. In *Proceedings of the 44th Annual Design Automation Conference (DAC)*, June 2007.

[20] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[21] 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)*, Vancouver, Canada, Oct. 2010.

[22] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[23] N. Brown. An f2fs teardown, 2010. https://lwn.net/Articles/518988/.

[24] M. Callaghan. Bug #55004: async fuzzy checkpoint constraint isn't really async, 2010. http://bugs.mysql.com/bug.php?id=55004.

[25] M. Callaghan. InnoDB fuzzy checkpoints, 2010. https://www.facebook.com/notes/mysqlfacebook/innodb-fuzzy-checkpoints/408059000932.

[26] M. Callaghan. XFS, ext and per-inode mutexes, 2011. https://www.facebook.com/notes/mysql-at-facebook/xfs-ext-and-per-inode-mutexes/10150210901610933.

[27] M. Cao, J. R. Santos, and A. Dilger. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium*, 2008.

[28] M.-L. Chiang, P. C. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for flash memory. *Software-Practice & Experience*, 29(3):267–290, 1999.

[29] 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)*, San Jose, California, USA, Feb. 2012.

[30] D. Chinner. XFS Delayed Logging Design, 2010. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/xfs-delayed-logging-design.txt.

[31] D. Chinner. Improving Metadata Performance By Reducing Journal Overhead, 2010. http://xfs.org/index.php/Improving_Metadata_Performance_By_Reducing_Journal_Overhead.

[32] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

[33] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the ACM EuroSys Conference*, Prague, Czech Republic, Apr. 2013.

[34] 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)*, Farmington, PA, Nov. 2013.

[35] J. Corbet. JLS: Increasing VFS scalability, 2009. https://lwn.net/Articles/360199/.

[36] J. Corbet. Dcache scalability and RCU-walk, 2010. https://lwn.net/Articles/419811/.

[37] J. Corbet. XFS: the filesystem of the future?, 2012. https://lwn.net/Articles/476263/.

[38] J. Corbet. Introducing lockrefs, 2013. https://lwn.net/Articles/565734/.

[39] J. Corbet. Per-CPU reference counts, 2013. https://lwn.net/Articles/557478/.

[40] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Portland, OR, Aug. 2007.

[41] R. Eqbal. ScaleFS: A Multicore-Scalable File System. Master's thesis, Massachusetts Institute of Technology, 2014.

[42] Ext4 Wiki. Ext4 Disk Layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[43] Facebook. RocksDB, 2013. http://rocksdb.org/.

[44] M. J. Foley. Microsoft SQL Server 2014 released to manufacturing, 2014. http://www.zdnet.com/article/microsoft-sql-server-2014-released-to-manufacturing/.

[45] C. Gruenwald III, F. Sironi, M. F. Kaashoek, and N. Zeldovich.

Hare: a file system for non-cache-coherent multicores. In *Proceedings of the ACM EuroSys Conference*, Bordeaux, France, Apr. 2015.

[46] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[47] J. He, D. Nguyen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Reducing file system tail latencies with Chopper. In *Proceedings of the 13th Usenix Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, Feb. 2015.

[48] HP. Performance improvements using Concurrent I/O on HP-UX 11i v3 with OnlineJFS 5.0.1 and the HP-UX 11i Logical Volume Manager, 2015. `http://www.filibeto.org/unix/hp-ux/lib/os/volume-manager/perf-hpux-11.31-cio-onlinejfs-4AA1-5719ENW.pdf`.

[49] *Converged System for SAP HANA Scale-out Configurations*. HP, 2015. `http://www8.hp.com/h20195/v2/GetPDF.aspx%2F4AA5-1488ENN.pdf`.

[50] IBM. Use concurrent I/O to improve DB2 database performance, 2012. `http://www.ibm.com/developerworks/data/library/techarticle/dm-1204concurrent/`.

[51] Intel. Performance Benchmarking for PCIe and NVMe Enterprise Solid-State Drives, 2015. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-pcie-nvme-enterprise-ssds-white-paper.pdf`.

[52] D. Jeong, Y. Lee, and J.-S. Kim. Boosting quasi-asynchronous I/O for better responsiveness in mobile devices. In *Proceedings of the 13th Usenix Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, Feb. 2015.

[53] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 24–35. ACM, 2009.

[54] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A Scalable Approach to Logging. *Proc. VLDB Endow.*, 3(1-2):681–692, Sept. 2010.

[55] D. H. Kang, C. Min, and Y. I. Eom. An Efficient Buffer Replacement Algorithm for NAND Flash Storage Devices. In *Proceedings of the 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, Paris, France, Sept. 2014.

[56] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai. MultiLanes: providing virtualized storage for OS-level virtualization on many cores. In *Proceedings of the 12th Usenix Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, Feb. 2014.

[57] 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 ATC Annual Technical Conference (ATC)*, Santa Clara, CA, July 2015.

[58] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX conference on Hot Topics in Storage and File Systems*, pages 13–13. USENIX Association, 2014.

[59] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. *Trans. Storage*, 8(4):14:1–14:25, Dec. 2012.

[60] J. Kim. f2fs: introduce flash-friendly file system , 2012. `https://lwn.net/Articles/518718/`.

[61] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a Complete Operating System. In *Proceedings of the ACM EuroSys Conference*, Leuven, Belgium, Apr. 2006.

[62] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal. ATAC: A 1000-core Cache-coherent Processor with On-chip Optical Network. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Vienna, Austria, Sept. 2010.

[63] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th Usenix Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, Feb. 2015.

[64] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In *Proceedings of the 2015 ATC Annual Technical Conference (ATC)*, Santa Clara, CA, July 2015.

[65] W. Long. [PATCH] dcache: Translating dentry into pathname without taking rename_lock, 2013. `https://lkml.org/lkml/2013/9/4/471`.

[66] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *Proceedings of the 30th IEEE International Conference on Data Engineering Workshop*, Chicago, IL, Mar. 2014.

[67] 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*, 2007.

[68] R. McDougall. Solaris Internals and Performance FAQ: Direct I/O, 2012. `http://www.solarisinternals.com/wiki/index.php/Direct_I/O`.

[69] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Trans. Comput. Syst.*, 2(3):181–197, Aug. 1984. ISSN 0734-2071.

[70] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th Usenix Conference on File and Storage Technologies (FAST)*, San Jose, California, USA, Feb. 2012.

[71] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *Proceedings of the 2015 ATC Annual Technical Conference (ATC)*, Santa Clara, CA, July 2015.

[72] T. P. Morgan. Flashtec NVRAM Does 15 Million IOPS At Sub-Microsecond Latency, 2014. `http://www.enterprisetech.com/2014/08/06/flashtec-nvram-15-million-iops-sub-microsecond-latency/`.

[73] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[74] S. Peter, J. Li, I. Zhang, D. R. Ports, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[75] 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)*, Broomfield, Colorado, Oct. 2014.

[76] A. Raj. CPU hotplug Support in Linux(tm) Kernel, 2006. `https://www.kernel.org/doc/Documentation/cpu-hotplug.txt`.

[77] O. Rodeh. B-trees, Shadowing, and Clones. *Trans. Storage*, 3(4):

2:1–2:27, Feb. 2008. ISSN 1553-3077.

[78] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[79] C. Rohland. Tmpfs is a file system which keeps all files in virtual memory, 2001. git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/tmpfs.txt.

[80] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992. ISSN 0734-2071.

[81] F. B. Schmuck and R. L. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st Usenix Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[82] D. I. Shin, Y. J. Yu, H. S. Kim, J. W. Choi, D. Y. Jung, and H. Y. Yeom. Dynamic interval polling and pipelined post i/o processing for low-latency storage class memory. In *Proceedings of the 5th USENIX conference on Hot Topics in Storage and File Systems*, pages 5–5. USENIX Association, 2013.

[83] Silicon Graphics Inc. XFS Filesystem Structure, 2006. http://xfs.org/docs/xfsdocs-xml-dev/XFS_Filesystem_Structure/tmp/en-US/html/index.html.

[84] J. Swanhart. An Introduction to InnoDB Internals, 2011. https://www.percona.com/files/percona-live/justin-innodb-internals.pdf.

[85] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the 1996 ATC Annual Technical Conference (ATC)*, Jan. 1996.

[86] B. Tallis. Intel Announces SSD DC P3608 Series, 2015. http://www.anandtech.com/show/9646/intel-announces-ssd-dc-p3608-series.

[87] D. Tang and M. Seltzer. Lies, damned lies, and file system benchmarks. Technical report, Technical Report TR-34-94, Harvard University, 1994.

[88] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking file system benchmarking: It* is* rocket science. *HotOS XIII*, 2011.

[89] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright. A Nine Year Study of File System and Storage Benchmarking. *Trans. Storage*, 4(2):5:1–5:56, May 2008.

[90] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[91] T. Wang and R. Johnson. Scalable Logging Through Emerging Non-volatile Memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[92] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th Usenix Conference on File and Storage Technologies (FAST)*, San Jose, California, USA, Feb. 2012.

[93] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, H. Eom, and H. Y. Yeom. Exploiting peak device throughput from random access workload. In *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems*. USENIX Association, 2012.