Evolving Ext4 for Shingled Disks

Abutalib Aghayev, ***Carnegie Mellon University;*** **Theodore Ts'o,** ***Google, Inc.;***
Garth Gibson, ***Carnegie Mellon University;*** **Peter Desnoyers,** ***Northeastern University***

https://www.usenix.org/conference/fast17/technical-sessions/presentation/aghayev

This paper is included in the Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

Open access to the Proceedings of the 15th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

Evolving Ext4 for Shingled Disks

Abutalib Aghayev — *Carnegie Mellon University*
Theodore Ts'o — *Google, Inc.*
Garth Gibson — *Carnegie Mellon University*
Peter Desnoyers — *Northeastern University*

Abstract

Drive-Managed SMR (Shingled Magnetic Recording) disks offer a plug-compatible higher-capacity replacement for conventional disks. For non-sequential workloads, these disks show bimodal behavior: After a short period of high throughput they enter a continuous period of low throughput.

We introduce *ext4-lazy*[1], a small change to the Linux ext4 file system that significantly improves the throughput in both modes. We present benchmarks on four different drive-managed SMR disks from two vendors, showing that ext4-lazy achieves 1.7-5.4× improvement over ext4 on a metadata-light file server benchmark. On metadata-heavy benchmarks it achieves 2-13× improvement over ext4 on drive-managed SMR disks as well as on conventional disks.

1 Introduction

Over 90% of all data in the world has been generated over the last two years [14]. To cope with the exponential growth of data, as well as to stay competitive with NAND flash-based solid state drives (SSDs), hard disk vendors are researching capacity-increasing technologies like Shingled Magnetic Recording (SMR) [20, 60], Heat Assisted Magnetic Recording (HAMR) [29], and Bit-Patterned Magnetic Recording (BPMR) [2, 13]. While HAMR and BPMR are still in the research stage, SMR allows disk manufacturers to increase areal density with existing fabrication methods. Unfortunately, this increase in density comes at the cost of increased complexity, resulting in a disk that has different behavior than Conventional Magnetic Recording (CMR) disks. Furthermore, since SMR can complement HAMR and BPMR to provide even higher growth in areal density, it is likely that all high-capacity disks in the near future will use SMR [42].

The industry has tried to address SMR adoption by introducing two kinds of SMR disks: Drive-Managed (DM-SMR) and Host-Managed (HM-SMR). DM-SMR disks are a drop-in replacement for conventional disks that offer higher capacity with the traditional block interface, but can suffer performance degradation when subjected to non-sequential write traffic. Unlike CMR disks that have a low but consistent throughput under random writes, DM-SMR disks offer high throughput for a short period followed by a precipitous drop, as shown in Figure 1. HM-SMR disks, on the other hand, offer a backward-incompatible interface that requires major changes to the I/O stacks to allow SMR-aware software to optimize their access pattern.

A new HM-SMR disk interface presents an interesting problem to storage researchers who have already proposed new file system designs based on it [10, 24, 32]. It also presents a challenge to the developers of existing file systems [12, 15, 16] who have been optimizing their code for CMR disks for years. There have been attempts to revamp mature Linux file systems like ext4 and XFS [11, 41, 42] to use the new interface, but these attempts have stalled due to the large amount of redesign required. The Log-Structured File System (LFS) [47], on the other hand, has an architecture that can be most easily adapted to an HM-SMR disk. However, although LFS has been influential, disk file systems based on it [28, 49] have not reached production quality in practice [34, 40, 48] .

We take an alternative approach to SMR adoption. Instead of redesigning for the HM-SMR disk interface, we make an incremental change to a mature, high performance file system, to optimize its performance on a DM-SMR disk. The systems community is no stranger to taking a revolutionary approach when faced with a new technology [5], only to discover that the existing system can be evolved to take the advantage of the new technology with a little effort [6]. Following a similar evolutionary approach, we take the first step to optimize ext4 file system for DM-SMR disks, observing that random writes are even more expensive on these disks, and that metadata writeback is a key generator of it.

We introduce ext4-lazy, a small change to ext4 that eliminates most metadata writeback. Like other journaling file systems [45], ext4 writes metadata twice; as Figure 2 (a) shows,

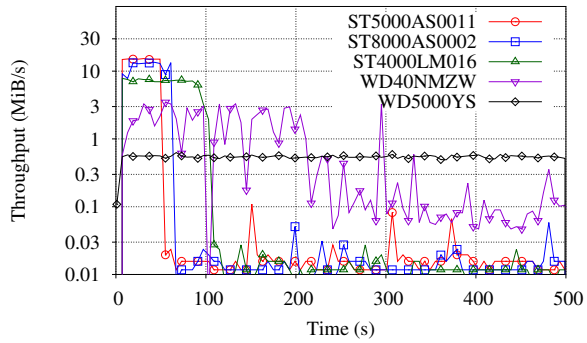


Figure 1: Throughput of CMR and DM-SMR disks from Table 1 under 4 KiB random write traffic. CMR disk has a stable but low throughput under random writes. DM-SMR disks, on the other hand, have a short period of high throughput followed by a continuous period of ultra-low throughput.

Type	Vendor	Model	Capacity	Form Factor
DM-SMR	Seagate	ST8000AS0002	8 TB	3.5 inch
DM-SMR	Seagate	ST5000AS0011	5 TB	3.5 inch
DM-SMR	Seagate	ST4000LM016	4 TB	2.5 inch
DM-SMR	Western Digital	WD40NMZW	4 TB	2.5 inch
CMR	Western Digital	WD5000YS	500 GB	3.5 inch

Table 1: CMR and DM-SMR disks from two vendors used for evaluation.

[1]The suffix *-lazy* is short for Lazy Writeback Journaling.

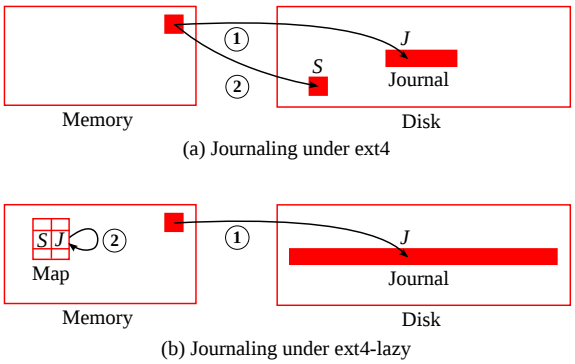


Figure 2: (a) Ext4 writes a metadata block to disk twice. It first writes the metadata block to the journal at some location J and marks it dirty in memory. Later, the writeback thread writes the same metadata block to its static location S on disk, resulting in a random write. (b) Ext4-lazy, writes the metadata block approximately once to the journal and inserts a mapping (S,J) to an in-memory map so that the file system can find the metadata block in the journal.

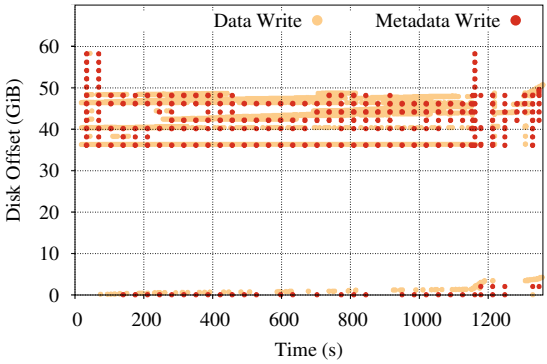


Figure 3: Offsets of data and metadata writes obtained with `blktrace` [4], when compiling Linux kernel 4.6 with all of its modules on a fresh ext4 file system. The workload writes 12 GiB of data, 185 MiB of journal (omitted from the graph), and only 98 MiB of metadata, making it 0.77% of total writes.

it first writes the metadata block to a temporary location J in the journal and then marks the block as *dirty* in memory. Once it has been in memory for long enough[2], the *writeback* (or *flusher*) thread writes the block to its *static location* S, resulting in a random write. Although metadata writeback is typically a small portion of a workload, it results in many random writes, as Figure 3 shows. Ext4-lazy, on the other hand, marks the block as *clean* after writing it to the journal, to prevent the writeback, and inserts a mapping (S,J) to an in-memory map allowing the file system to access the block in the journal, as seen in Figure 2 (b). Ext4-lazy uses a large journal so that it can continue writing updated blocks while reclaiming the space from the stale blocks. During mount, it reconstructs the in-memory map from the journal resulting in a modest increase in mount time. Our results show that ext4-lazy significantly improves performance on DM-SMR disks, as well as on CMR disks for metadata-heavy workloads.

Our key contribution in this paper is the design, implementation, and evaluation of ext4-lazy on DM-SMR and CMR disks. Our change is minimally invasive—we modify 80 lines of existing code and introduce the new functionality in additional files totaling 600 lines of C code. On a metadata-light ($\leq 1\%$ of total writes) file server benchmark, ext4-lazy increases DM-SMR disk throughput by 1.7-5.4×. For directory traversal and metadata-heavy workloads it achieves 2-13× improvement on both DM-SMR and CMR disks.

In addition, we make two contributions that are applicable beyond our proposed approach:

- For purely sequential write workloads, DM-SMR disks perform at full throughput and do not suffer performance degradation. We identify the minimal sequential I/O size to trigger this behavior for a popular DM-SMR disk.
- We show that for physical journaling [45], a small journal is a bottleneck for metadata-heavy workloads. Based on our result, ext4 developers have increased the default journal size from 128 MiB to 1 GiB for file systems over 128 GiB [54].

In the rest of the paper, we first give background on SMR technology, describe why random writes are expensive in DM-SMR disks, and show why metadata writeback in ext4 is causing more random writes (§ 2). Next, we motivate ext4-lazy and describe its design and implementation (§ 3). Finally, we evaluate our implementation (§ 4), cover related work (§ 5) and present our conclusions (§ 6). Source code and other artifacts to reproduce our results are available at `http://www.pdl.cmu.edu/Publications/downloads.shtml`.

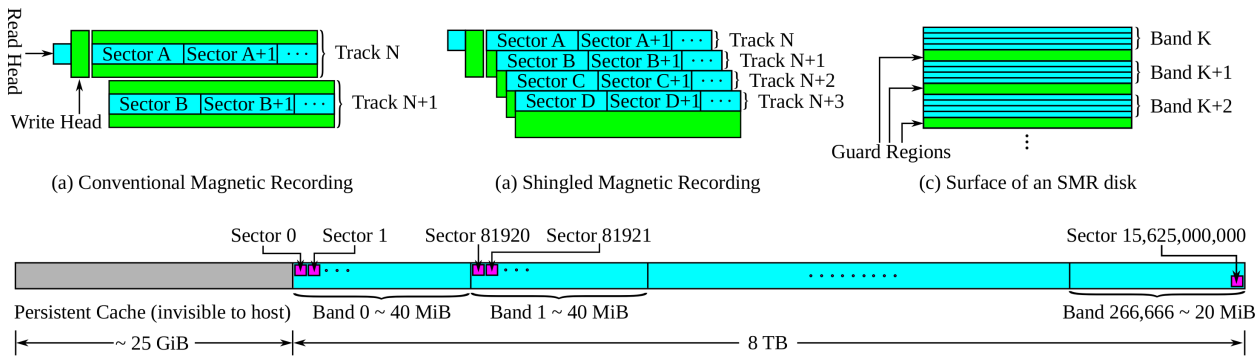
2 Background

We introduce SMR technology in general and describe how DM-SMR disks work. We then describe how ext4 lays out data on a disk and how it uses a generic layer in the kernel to enable journaling.

2.1 DM-SMR Internals

SMR leverages the difference in the width of the disk's *write head* and *read head* to squeeze more tracks into the same area than CMR. In CMR, tracks have the width of the write head even though they are read with a narrow read head, as seen in Figure 4 (a). In SMR, however, the tracks are written on top of each other, leaving just enough space for the read head to distinguish them, increasing track density, as seen in Figure 4 (b). Unlike CMR, however, overlapping writes cause the sector updates to corrupt data in adjacent tracks. Therefore, the surface of an SMR disk is divided into *bands* that are collections of narrow tracks divided by wide tracks called *guard regions*, as seen in Figure 4 (c). A band in an SMR disk represents a unit that can be safely overwritten sequentially, beginning at the first track and ending at the last. A write to any sector in a band—except to sectors in the last track of the band—will require read-modify-write (RMW) of all the tracks forming the band.

[2] Controlled by /proc/sys/vm/dirty_expire_centisecs in Linux.



(d) Logical view of an 8 TB statically mapped drive-managed SMR disk

Figure 4: (a) In conventional recording the tracks have the width of the write head. (b) In shingled recording the tracks are laid partially on top of each other reducing the track width to the read head. This allows for more tracks, however, unlike with conventional recording, overwriting a sector corrupts other sectors. (c) Therefore, the surface of an SMR disk is divided into bands made up of multiple tracks separated by guard regions. (d) An SMR disk also contains a persistent cache for absorbing random writes, in addition to a sequence of bands to whom a group of sectors are mapped.

HM-SMR disks provide an interface that exposes the band information and let the host manage data on disk. One such interface is going through the standardization [23] and researchers are coming up with others [17,25,34]. DM-SMR disks, on the other hand, implement a Shingle Translation Layer (STL)—similar to the Flash Translation Layer (FTL) in SSDs—that manages data in firmware while exposing the block interface to the host.

All STLs proposed in the literature and found in actual DM-SMR disks to this date [1,9,21,22,56] contain one or more *persistent caches* for absorbing random writes, to avoid expensive RMW operation for each write. Consequently, when a write operation updates some part of a band, the STL writes the update to the persistent cache, and the band becomes dirty. An STL *cleans* a dirty band in the background or during idle times, by merging updates for the band from the persistent cache with unmodified data from the band, and writing back to the band, freeing space used by the updates in the persistent cache.

The cost of cleaning a band may vary based on the type of the *block mapping* used. With *dynamic mapping* an STL can read a band, update it in memory, write the updated band to a different band, and fix the mapping, resulting in a read and a write of a band. With *static mapping*, however, an STL needs to persist the updated band to a scratch space first—directly overwriting the band can corrupt it in case of a power failure—resulting in a read and two writes of a band.

As a concrete example, Figure 4 (d) shows the logical view of Seagate ST8000AS0002 DM-SMR disk that was recently studied in detail [1]. With an average band size of 30 MiB, the disk has over 260,000 bands with sectors statically mapped to the bands, and a $\approx$25 GiB persistent cache that is not visible to the host. The STL in this disk detects sequential writes and starts *streaming* them directly to the bands, bypassing the persistent cache. Random writes, however, end up in the persistent cache, dirtying bands. Cleaning a single band typically takes 1-2 seconds, but can take up to 45 seconds in extreme cases.

STLs also differ in their cleaning strategies. Some STLs constantly clean in small amounts, while others clean during idle times. If the persistent cache fills before the workload completes, the STL is forced to interleave cleaning with work, reducing throughput. Figure 1 shows the behavior of DM-SMR disks from two vendors. Seagate disks are known to clean during idle times and to have static mapping [1]. Therefore, they have high throughput while the persistent cache is not full, and ultra-low throughput after it fills. The difference in the time when the throughput drops suggests that the persistent cache size varies among the disks. Western Digital disks, on the other hand, are likely to clean constantly and have dynamic mapping [9]. Therefore, they have lower throughput than Seagate disks while the persistent cache is not full, but higher throughput after it fills.

2.2 Ext4 and Journaling

The ext4 file system evolved [30,36] from ext2 [8], which was influenced by Fast File System (FFS) [37]. Similar to FFS, ext2 divides the disk into *cylinder groups*—or as ext2 calls them, *block groups*—and tries to put all blocks of a file in the same block group. To further increase locality, the metadata blocks (*inode bitmap*, *block bitmap*, and *inode table*) representing the files in a block group are also placed within the same block group, as Figure 5 (a) shows. *Group descriptor blocks*, whose location is fixed within the block group, identify the location of these metadata blocks that are typically located in the first megabyte of the block group.

In ext2 the size of a block group was limited to 128 MiB—the maximum number of 4 KiB data blocks that a 4 KiB block bitmap can represent. Ext4 introduced *flexible block groups* or *flex_bgs* [30], a set of contiguous block groups[3] whose metadata is consolidated in the first 16 MiB of the first block group within the set, as shown in Figure 5 (b).

[3] We assume the default size of 16 block groups per flex_bg.

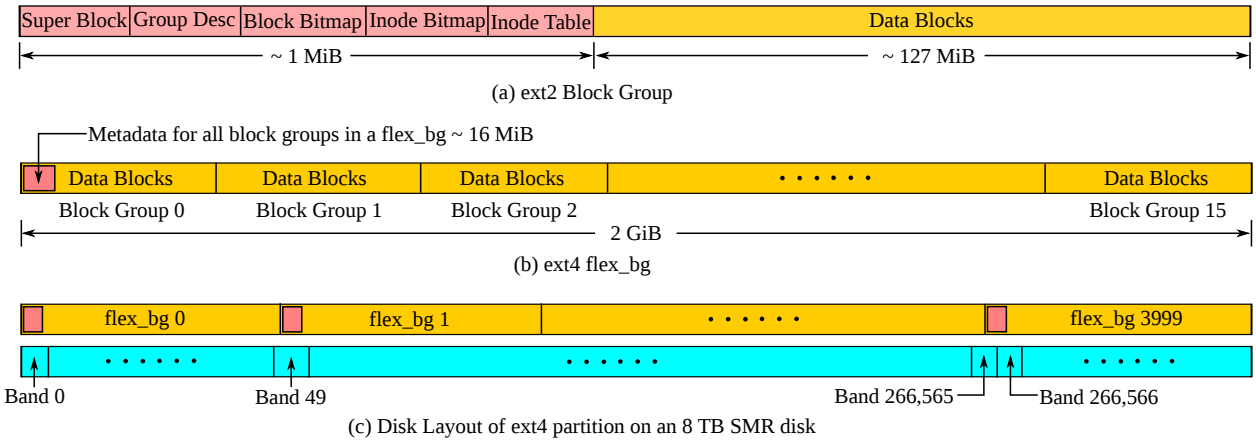


Figure 5: (a) In ext2, the first megabyte of a 128 MiB block group contains the metadata blocks describing the block group, and the rest is data blocks. (b) In ext4, a single flex_bg concatenates multiple (16 in this example) block groups into one giant block group and puts all of the metadata in the first block group. (c) Modifying data in a flex_bg will result in a metadata write that may dirty one or two bands, seen at the boundary of bands 266,565 and 266,566.

Ext4 ensures metadata consistency via journaling, however, it does not implement journaling itself; rather, it uses a generic kernel layer called the *Journaling Block Device* [55] that runs in a separate kernel thread called *jbd2*. In response to file system operations, ext4 reads metadata blocks from disk, updates them in memory, and exposes them to jbd2 for journaling. For increased performance, jbd2 batches metadata updates from multiple file system operations (by default, for 5 seconds) into a *transaction* buffer and atomically *commits* the transaction to the journal—a *circular log* of transactions with a head and tail pointer. A transaction may commit early if the buffer reaches maximum size, or if a synchronous write is requested. In addition to metadata blocks, a committed transaction contains *descriptor blocks* that record the static locations of the metadata blocks within the transaction. After a commit, jbd2 marks the in-memory copies of metadata blocks as dirty so that the writeback threads would write them to their static locations. If a file system operation updates an in-memory metadata block before its dirty timer expires, jbd2 writes the block to the journal as part of a new transaction and delays the writeback of the block by resetting its timer.

On DM-SMR disks, when the metadata blocks are eventually written back, they dirty the bands that are mapped to the metadata regions in a flex_bg, as seen in Figure 5 (c). Since a metadata region is not aligned with a band, metadata writes to it may dirty zero, one, or two extra bands, depending on whether the metadata region spans one or two bands and whether the data around the metadata region has been written.

3 Design and Implementation of ext4-lazy

We start by motivating ext4-lazy, follow with a high-level view of our design, and finish with the implementation details.

3.1 Motivation

The motivation for ext4-lazy comes from two observations: (1) metadata writeback in ext4 results in random writes that cause a significant cleaning load on a DM-SMR disk, and (2) file system metadata comprises a small set of blocks, and *hot* (frequently updated) metadata is an even smaller set. The corollary of the latter observation is that managing hot metadata in a circular log several times the size of hot metadata turns random writes into purely sequential writes, reducing the cleaning load on a DM-SMR disk. We first give calculated evidence supporting the first observation and follow with empirical evidence for the second observation.

On an 8 TB partition, there are about 4,000 flex_bgs, the first 16 MiB of each containing the metadata region, as shown in Figure 5 (c). With a 30 MiB band size, updating every flex_bg would dirty 4,000 bands on average, requiring cleaning of 120 GiB worth of bands, generating 360 GiB of disk traffic. A workload touching 1/16 of the whole disk, that is 500 GiB of files, would dirty at least 250 bands requiring 22.5 GiB of cleaning work. The cleaning load increases further if we consider floating metadata like extent tree blocks and directory blocks.

To measure the hot metadata ratio, we emulated the I/O workload of a build server on ext4, by running 128 parallel Compilebench [35] instances, and categorized all of the writes completed by disk. Out of 433 GiB total writes, 388 GiB were data writes, 34 GiB were journal writes, and 11 GiB were metadata writes. The total size of unique metadata blocks was 3.5 GiB, showing that it was only 0.8% of total writes, and that 90% of journal writes were overwrites.

3.2 Design

At a high level, ext4-lazy adds the following components to ext4 and jbd2:

Map: Ext4-lazy tracks the location of metadata blocks in the journal with *jmap*—an in-memory map that associates the static location S of a metadata block with its location J in the journal. The mapping is updated whenever a metadata block is written to the journal, as shown in Figure 2 (b).

Indirection: In ext4-lazy all accesses to metadata blocks go through jmap. If the most recent version of a block is in the journal, there will be an entry in jmap pointing to it; if no entry is found, then the copy at the static location is up-to-date.
Cleaner: The cleaner in ext4-lazy reclaims space from locations in the journal which have become *stale*, that is, invalidated by the writes of new copies of the same metadata block.
Map reconstruction on mount: On every mount, ext4-lazy reads the descriptor blocks from the transactions between the tail and the head pointer of the journal and populates jmap.

3.3 Implementation

We now detail our implementation of the above components and the trade-offs we make during the implementation. We implement jmap as a standard Linux red-black tree [31] in jbd2. After jbd2 commits a transaction, it updates jmap with each metadata block in the transaction and marks the in-memory copies of those blocks as clean so they will not be written back. We add indirect lookup of metadata blocks to ext4 by changing the call sites that read metadata blocks to use a function which looks up the metadata block location in jmap, as shown in Listing 1, modifying 40 lines of ext4 code in total.

```
−  submit_bh (READ | REQ_META | REQ_PRIO, bh);
+  jbd2_submit_bh ( journal ,  READ | REQ_META | REQ_PRIO, bh);
```

Listing 1: Adding indirection to a call site reading a metadata block.

The indirection allows ext4-lazy to be backward-compatible and gradually move metadata blocks to the journal. However, the primary reason for indirection is to be able to migrate *cold* (not recently updated) metadata back to its static location during cleaning, leaving only hot metadata in the journal.

We implement the cleaner in jbd2 in just 400 lines of C, leveraging the existing functionality. In particular, the cleaner merely reads live metadata blocks from the tail of the journal and adds them to the transaction buffer using the same interface used by ext4. For each transaction it keeps a doubly-linked list that links jmap entries containing live blocks of the transaction. Updating a jmap entry invalidates a block and removes it from the corresponding list. To clean a transaction, the cleaner identifies the live blocks of a transaction in constant time using the transaction's list, reads them, and adds them to the transaction buffer. The beauty of this cleaner is that it does not "stop-the-world", but transparently mixes cleaning with regular file system operations causing no interruptions to them, as if cleaning was just another operation. We use a simple cleaning policy—after committing a fixed number of transactions, clean a fixed number of transactions—and leave sophisticated policy development, such as hot and cold separation, for future work.

Map reconstruction is a small change to the recovery code in jbd2. Stock ext4 resets the journal on a normal shutdown; finding a non-empty journal on mount is a sign of crash and triggers the recovery process. With ext4-lazy, the state of the journal represents the persistent image of jmap, therefore, ext4-lazy never resets the journal and always "recovers". In our prototype, ext4-lazy reconstructs the jmap by reading descriptor blocks from the transactions between the tail and head pointer of the journal, which takes 5-6 seconds when the space between the head and tail pointer is ≈ 1 GiB.

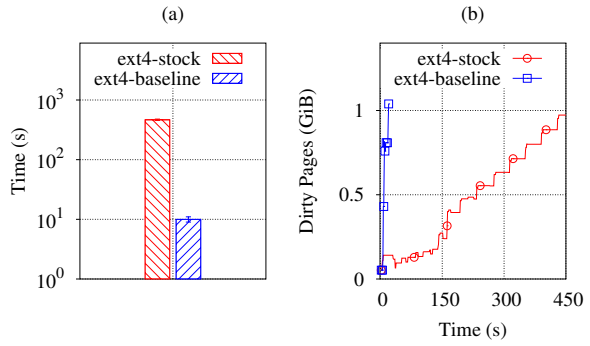


Figure 6: (a) Completion time for a benchmark creating 100,000 files on ext4-stock (ext4 with 128 MiB journal) and on ext4-baseline (ext4 with 10 GiB journal). (b) The volume of dirty pages during benchmark runs obtained by sampling `/proc/meminfo` every second.

4 Evaluation

We run all experiments on a system with a quad-core Intel i7-3820 (Sandy Bridge) 3.6 GHz CPU, 16 GB of RAM running Linux kernel 4.6 on the Ubuntu 14.04 distribution, using the disks listed in Table 1. To reduce the variance between runs, we unmount the file system between runs, always start with the same file system state, disable lazy initialization[4] when formatting ext4 partitions, and fix the writeback cache ratio [62] for our disks to 50% of the total—by default, this ratio is computed dynamically from the writeback throughput [53]. We repeat every experiment at least five times and report the average and standard deviation of the runtime.

4.1 Journal Bottleneck

Since it affects our choice of baseline, we start by showing that for metadata-heavy workloads, the default 128 MiB journal of ext4 is a bottleneck. We demonstrate the bottleneck on the CMR disk WD5000YS from Table 1 by creating 100,000 small files in over 60,000 directories, using *CreateFiles* microbenchmark from Filebench [52]. The workload size is ≈ 1 GiB and fits in memory.

Although ext4-lazy uses a large journal by definition, since enabling a large journal on ext4 is a command-line option to `mkfs`, we choose ext4 with a 10 GiB journal[5] as our baseline. In the rest of this paper, we refer to ext4 with the default journal size of 128 MiB as *ext4-stock*, and we refer to ext4 with 10 GiB journal as *ext4-baseline*.

[4] mkfs.ext4 -E lazy_itable_init=0,lazy_journal_init=0 /dev/<dev>
[5] Created by passing "-J size=10240" to mkfs.ext4.

We measure how fast ext4 can create the files in memory and do not consider the writeback time. Figure 6 (a) shows that on ext4-stock the benchmark completes in ≈460 seconds, whereas on ext4-baseline it completes 46× faster, in ≈10 seconds. Next we show how a small journal becomes a bottleneck.

The ext4 journal is a circular log of transactions with a head and tail pointer (§ 2.2). As the file system performs operations, jbd2 commits transactions to the journal, moving the head forward. A committed transaction becomes *checkpointed* when every metadata block in it is either written back to its static location due to a dirty timer expiration, or it is written to the journal as part of a newer transaction. To recover space, at the end of every commit jbd2 checks for transactions at the tail that have been checkpointed, and when possible moves the tail forward. On a metadata-light workload with a small journal and default dirty timer, jbd2 always finds checkpointed transactions at the tail and recovers the space without doing work. However, on a metadata-heavy workload, incoming transactions fill the journal before the transactions at the tail have been checkpointed. This results in a *forced checkpoint*, where jbd2 synchronously writes metadata blocks at the tail transaction to their static locations and then moves the tail forward, so that a new transaction can start [55].

We observe the file system behavior while running the benchmark by enabling tracepoints in the jbd2 code[6]. On ext4-stock, the journal fills in 3 seconds, and from then on until the end of the run, jbd2 moves the tail by performing forced checkpoints. On ext4-baseline the journal never becomes full and no forced checkpoints happen during the run.

Figure 6 (b) shows the volume of dirtied pages during the benchmark runs. On ext4-baseline, the benchmark creates over 60,000 directories and 100,000 files, dirtying about 1 GiB worth of pages in 10 seconds. On ext4-stock, directories are created in the first 140 seconds. Forced checkpoints still happen during this period, but they complete fast, as the small steps in the first 140 seconds show. Once the benchmark starts filling directories with files, the block groups fill and writes spread out to a larger number of block groups across the disk. Therefore, forced checkpoints start taking as long as 30 seconds, as indicated by the large steps, during which the file system stalls, no writes to files happen, and the volume of dirtied pages stays fixed.

This result shows that for disks, a small journal is a bottleneck for metadata-heavy buffered I/O workloads, as the journal wraps before metadata blocks are written to disk, and file system operations are stalled until the journal advances via synchronous writeback of metadata blocks. With a sufficiently large journal, all transactions will be written back before the journal wraps. For example, for a 190 MiB/s disk and a 30 second dirty timer, a journal size of 30s × 190 MiB/s = 5,700 MiB will guarantee that when the journal wraps, the transactions at the tail will be checkpointed. Having established our baseline, we move on to evaluation of ext4-lazy.

[6] /sys/kernel/debug/tracing/events/jbd2/

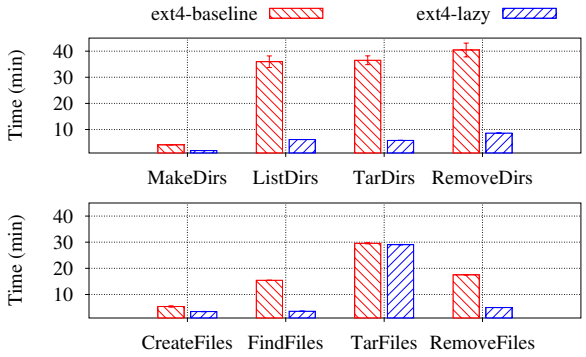


Figure 7: Microbenchmark runtimes on ext4-baseline and ext4-lazy.

4.2 Ext4-lazy on a CMR disk

We first evaluate ext4-lazy on the CMR disk WD5000YS from Table 1 via a series of microbenchmarks and a file server macrobenchmark. We show that on a CMR disk, ext4-lazy provides a significant speedup for metadata-heavy workloads, and specifically for massive directory traversal workloads. On metadata-light workloads, however, ext4-lazy does not have much impact.

4.2.1 Microbenchmarks

We evaluate directory traversal and file/directory create operations using the following benchmarks. *MakeDirs* creates 800,000 directories in a directory tree of depth 10. *ListDirs* runs `ls -lR` on the directory tree. *TarDirs* creates a tarball of the directory tree, and *RemoveDirs* removes the directory tree. *CreateFiles* creates 600,000 4 KiB files in a directory tree of depth 20. *FindFiles* runs `find` on the directory tree. *TarFiles* creates a tarball of the directory tree, and *RemoveFiles* removes the directory tree. MakeDirs and CreateFiles—microbenchmarks from Filebench—run with 8 threads and execute sync at the end. All benchmarks start with a cold cache[7].

Benchmarks that are in the file/directory create category (MakeDirs, CreateFiles) complete 1.5-2× faster on ext4-lazy than on ext4-baseline, while the remaining benchmarks that are in the directory traversal category, except TarFiles, complete 3-5× faster, as seen in Figure 7. We choose MakeDirs and RemoveDirs as a representative of each category and analyze their performance in detail.

MakeDirs on ext4-baseline results in ≈4,735 MiB of journal writes that are transaction commits containing metadata blocks, as seen in the first row of Table 2 and at the center in Figure 8 (a); as the dirty timer on the metadata blocks expires, they are written to their static locations, resulting in a similar amount of metadata writeback. The block allocator is able to allocate large contiguous blocks for

[7] `echo 3 > /proc/sys/vm/drop_caches`

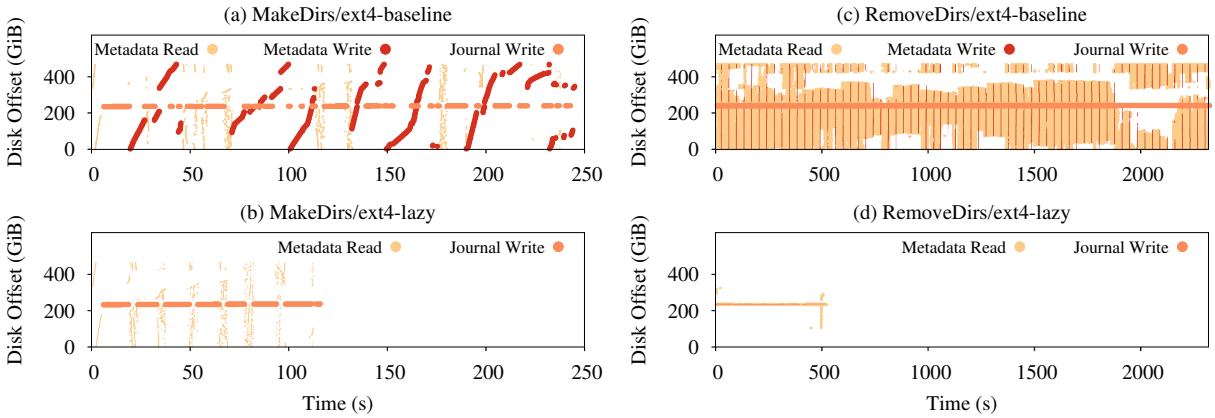


Figure 8: Disk offsets of I/O operations during MakeDirs and RemoveDirs microbenchmarks on ext4-baseline and ext4-lazy. Metadata reads and writes are spread out while journal writes are at the center. The dots have been scaled based on the I/O size. In part (d), journal writes are not visible due to low resolution. These are pure metadata workloads with no data writes.

	Metadata Reads (MiB)	Metadata Writes (MiB)	Journal Writes (MiB)
MakeDirs/ext4-baseline	143.7±2.8	4,631±33.8	4,735±0.1
MakeDirs/ext4-lazy	144±4	0	4,707±1.8
RemoveDirs/ext4-baseline	4,066.4±0.1	322.4±11.9	1,119±88.6
RemoveDirs/ext4-lazy	4,066.4±0.1	0	472±3.9

Table 2: Distribution of the I/O types with MakeDirs and RemoveDirs benchmarks running on ext4-baseline and ext4-lazy.

the directories, because the file system is fresh. Therefore, in addition to journal writes, metadata writeback is sequential as well. The write time dominates the runtime in this workload, hence, by avoiding metadata writeback and writing only to the journal, ext4-lazy halves the writes as well as the runtime, as seen in the second row of Table 2 and Figure 8 (b). On an aged file system, the metadata writeback is more likely to be random, resulting in even higher improvement on ext4-lazy.

An interesting observation about Figure 8 (b) is that although the total volume of metadata reads—shown as periodic vertical spreads—is ≈ 140 MiB (3% of total I/O in the second row of Table 2), they consume over 30% of runtime due to long seeks across the disk. In this benchmark, the metadata blocks are read from their static locations because we run the benchmark on a fresh file system, and the metadata blocks are still at their static locations. As we show next, once the metadata blocks migrate to the journal, reading them is much faster since no long seeks are involved.

In RemoveDirs benchmark, on both ext4-baseline and ext4-lazy, the disk reads ≈ 4,066 MiB of metadata, as seen in the last two rows of Table 2. However, on ext4-baseline the metadata blocks are scattered all over the disk, resulting in long seeks as indicated by the vertical spread in Figure 8 (c), while on ext4-lazy they are within the 10 GiB region in the journal, resulting in only short seeks, as Figure 8 (d) shows. Ext4-lazy also benefits from skipping metadata writeback, but most of the improvement comes from eliminating long seeks for metadata reads. The significant difference in the volume of journal writes between ext4-baseline and ext4-lazy seen in Table 2 is caused by metadata write coalescing: since ext4-lazy completes faster, there are more operations in each transaction, with many modifying the same metadata blocks, each of which is only written once to the journal.

The improvement in the remaining benchmarks, are also due to reducing seeks to a small region and avoiding metadata writeback. We do not observe a dramatic improvement in TarFiles, because unlike the rest of the benchmarks that read only metadata from the journal, TarFiles also reads data blocks of files that are scattered across the disk.

Massive directory traversal workloads are a constant source of frustration for users of most file systems [3, 18, 33, 43, 50]. One of the biggest benefits of consolidating metadata in a small region is an order of magnitude improvement in such workloads, which to our surprise was not noticed by previous work [44, 46, 61]. On the other hand, the above results are obtainable in the ideal case that all of the directory blocks are hot and therefore kept in the journal. If, for example, some part of the directory is cold and the policy decides to move those blocks to their static locations, removing such a directory will incur an expensive traversal.

4.2.2 File Server Macrobenchmark

We first show that ext4-lazy slightly improves the throughput of a metadata-light file server workload. Next we try to reproduce a result from previous work without success.

To emulate a file server workload, we started with the *Fileserver* macrobenchmark from Filebench but encountered bugs for large configurations. The development on Filebench has been recently restarted and the recommended version is still in alpha stage. Therefore, we decided to use Postmark [27], with some modifications.

Like the Fileserver macrobenchmark from Filebench, Postmark first creates a *working set* of files and directories

	Data Writes (MiB)	Metadata Writes (MiB)	Journal Writes (MiB)
ext4-baseline	34,185±10.3	480±0.2	1,890±18.6
ext4-lazy	33,878±9.8	0	1,855±15.4

Table 3: Distribution of write types completed by the disk during Postmark run on ext4-baseline and ext4-lazy. Metadata writes make 1.3% of total writes in ext4-baseline, only ⅓ of which is unique.

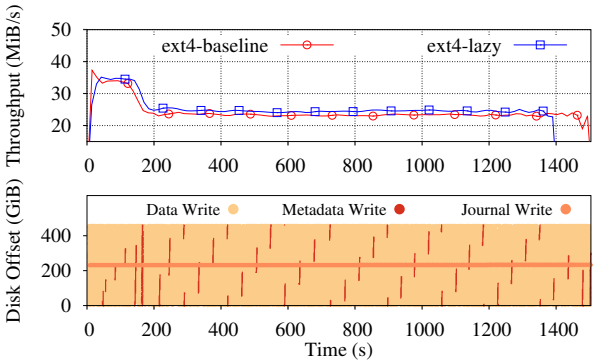


Figure 9: The top graph shows the throughput of the disk during a Postmark run on ext4-baseline and ext4-lazy. The bottom graph shows the offsets of write types during ext4-baseline run. The graph does not reflect sizes of the writes, but only their offsets.

and then executes *transactions* like reading, writing, appending, deleting, and creating files on the working set. We modify Postmark to execute `sync` after creating the working set, so that the writeback of the working set does not interfere with transactions. We also modify Postmark not to delete the working set at the end, but to run `sync`, to avoid high variance in runtime due to the race between deletion and writeback of data.

Our Postmark configuration creates a working set of 10,000 files spread sparsely across 25,000 directories with file sizes ranging from 512 bytes to 1 MiB, and then executes 100,000 transactions with the I/O size of 1 MiB. During the run, Postmark writes 37.89 GiB of data and reads 31.54 GiB of data from user space. Because ext4-lazy reduces the amount of writes, to measure its effect, we focus on writes.

Table 3 shows the distribution of data writes completed by the disk while the benchmark is running on ext4-baseline and on ext4-lazy. On ext4-baseline, metadata writes comprise 1.3% of total writes, all of which ext4-lazy avoids. As a result, the disk sees 5% increase in throughput on ext4-lazy from 24.24 MiB/s to 25.47 MiB/s and the benchmark completes 100 seconds faster on ext4-lazy, as the throughput graph in Figure 9 shows. The increase in throughput is modest because the workload spreads out the files across the disk resulting in traffic that is highly non-sequential, as data writes in the bottom graph of Figure 9 show. Therefore, it is not surprising that reducing random writes of a non-sequential write traffic by 1.3% results in a 5% throughput improvement. However, the same random writes result in extra cleaning work for DM-SMR disks (§ 2).

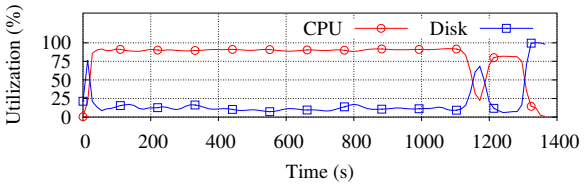


Figure 10: Disk and CPU utilization sampled from `iostat` output every second, while compiling Linux kernel 4.6 including all its modules, with 16 parallel jobs (`make -j16`) on a quad-core Intel i7-3820 (Sandy Bridge) CPU with 8 hardware threads.

Previous work [44] that writes metadata only once reports performance improvements even in a metadata-light workloads, like kernel compile. This has not been our experience. We compiled Linux kernel 4.6 with all its modules on ext4-baseline and observed that it generated 12 GiB of data writes and 185 MiB of journal writes. At 98 MiB, metadata writes comprised only 0.77% of total writes completed by the disk. This is expected, since metadata blocks are cached in memory, and because they are journaled, unlike data pages their dirty timer is reset whenever they are modified (§ 3), delaying their writeback. Furthermore, even on a system with 8 hardware threads running 16 parallel jobs, we found kernel compile to be CPU-bound rather than disk-bound, as Figure 10 shows. Given that reducing writes by 1.3% on a workload that utilized the disk 100% resulted in only 5% increase in throughput (Figure 9), it is not surprising that reducing writes by 0.77% on such a low-utilized disk does not cause improvement.

4.3 Ext4-lazy on DM-SMR disks

We show that unlike CMR disks, where ext4-lazy had a big impact on just metadata-heavy workloads, on DM-SMR disks it provides significant improvement on both, metadata-heavy and metadata-light workloads. We also identify the minimal sequential I/O size to trigger streaming writes on a popular DM-SMR disk.

An additional critical factor for file systems when running on DM-SMR disks is the cleaning time after a workload. A file system resulting in a short cleaning time gives the disk a better chance of emptying the persistent cache during idle times of a bursty I/O workload, and has a higher chance of continuously performing at the persistent cache speed, whereas a file system resulting in a long cleaning time is more likely to force the disk to interleave cleaning with file system user work.

In the next section we show microbenchmark results on just one DM-SMR disk—ST8000AS0002 from Table 1. At the end of every benchmark, we run a vendor provided script that polls the disk until it has completed background cleaning and reports the total cleaning time, which we report in addition to the benchmark runtime. We achieve similar normalized results for the remaining disks, which we skip to save space.

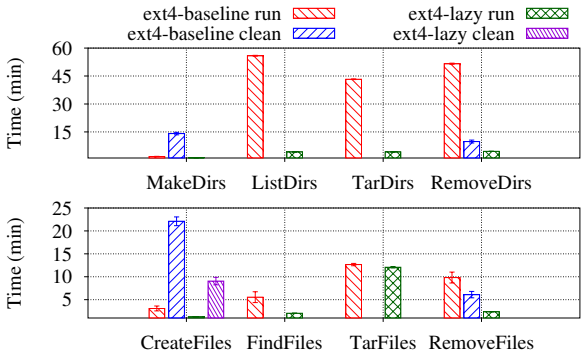


Figure 11: Microbenchmark runtimes and cleaning times on ext4-baseline and ext4-lazy running on an DM-SMR disk. Cleaning time is the additional time after the benchmark run that the DM-SMR disk was busy cleaning.

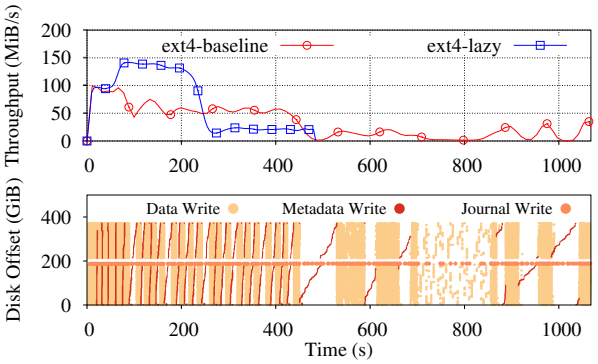


Figure 12: The top graph shows the throughput of a ST8000AS0002 DM-SMR disk with a 400 GB partition during a Postmark run on ext4-baseline and ext4-lazy. The bottom graph shows the offsets of write types during the run on ext4-baseline. The graph does not reflect sizes of the writes, but only their offsets.

4.3.1 Microbenchmarks

Figure 11 shows results of the microbenchmarks (§ 4.2.1) repeated on ST8000AS0002 with a 2 TB partition, on ext4-baseline and ext4-lazy. MakeDirs and CreateFiles do not fill the persistent cache, therefore, they typically complete 2-3× faster than on CMR disk. Similar to CMR disk, MakeDirs and CreateFiles are 1.5-2.5× faster on ext4-lazy. On the other hand, the remaining directory traversal benchmarks, ListDir for example, completes 13× faster on ext4-lazy, compared to being 5× faster on CMR disk.

The cleaning times for ListDirs, FindFiles, TarDirs, and TarFiles are zero because they do not write to disk[8]. However, cleaning time for MakeDirs on ext4-lazy is zero as well, compared to ext4-baseline's 846 seconds, despite having written over 4 GB of metadata, as Table 2 shows. Being a pure metadata workload, MakeDirs on ext4-lazy consists of journal writes only, as Figure 8 (b) shows, all of which are streamed, bypassing the persistent cache and resulting in zero cleaning time. Similarly, cleaning time for RemoveDirs and RemoveFiles are 10-20 seconds on ext4-lazy compared to 590-366 seconds on ext4-baseline, because these too are pure metadata workloads resulting in only journal writes for ext4-lazy. During deletion, however, some journal writes are small and end up in persistent cache, resulting in short cleaning times.

We confirmed that the disk was streaming journal writes in previous benchmarks by repeating the MakeDirs benchmark on the DM-SMR disk with an observation window from Skylight [1] and observing the head movement. We observed that shortly after starting the benchmark, the head moved to the physical location of the journal on the platter[9] and remained there until the end of the benchmark. This observation lead to Test 1 for identifying the minimal sequential write size that triggers streaming. Using this test, we found that sequential writes of at least 8 MiB in size are streamed. We also observed that a single 4 KiB random write in the middle of a sequential write disrupted streaming and moved the head to the persistent cache; soon the head moved back and continued streaming.

Test 1: Identify the minimal sequential write size for streaming

1 Choose identifiable location L on the platter
2 Start with a large sequential write size S
3 **do**
  Write S bytes sequentially at L
  $S = S$ - 1 MiB
  while Head moves to L and stays there until the end of the write
4 $S = S + 1$ MiB
5 Minimal sequential write size for streaming is S

4.3.2 File Server Macrobenchmark

We show that on DM-SMR disks the benefit of ext4-lazy increases with the partition size, and that ext4-lazy achieves a significant speedup on a variety of DM-SMR disks with different STLs and persistent cache sizes.

Table 4 shows the distribution of write types completed by a ST8000AS0002 DM-SMR disk with a 400 GB partition during the file server macrobenchmark (§ 4.2.2). On ext4-baseline, metadata writes make up 1.6% of total writes. Although the unique amount of metadata is only ≈ 120 MiB, as the storage slows down, metadata writeback increases slightly, because each operation takes a long time to complete and the writeback of a metadata block occurs before the dirty timer is reset.

Unlike the CMR disk, the effect is profound on a ST8000AS0002 DM-SMR disk. The benchmark completes more than 2× faster on ext4-lazy, in 461 seconds, as seen in Figure 12. On ext4-lazy, the disk sustains 140 MiB/s throughput and fills the persistent cache in 250 seconds, and then drops to a steady 20 MiB/s until the end of the run. On ext4-baseline, however, the large number of small metadata writes reduce throughput to 50 MiB/s taking the disk 450 seconds to fill the persistent cache. Once the persistent cache fills, the disk interleaves cleaning and file system user work, and small metadata writes become prohibitively expensive,

[8] TarDirs and TarFiles write their output to a different disk.
[9] Identified by observing the head while reading the journal blocks.

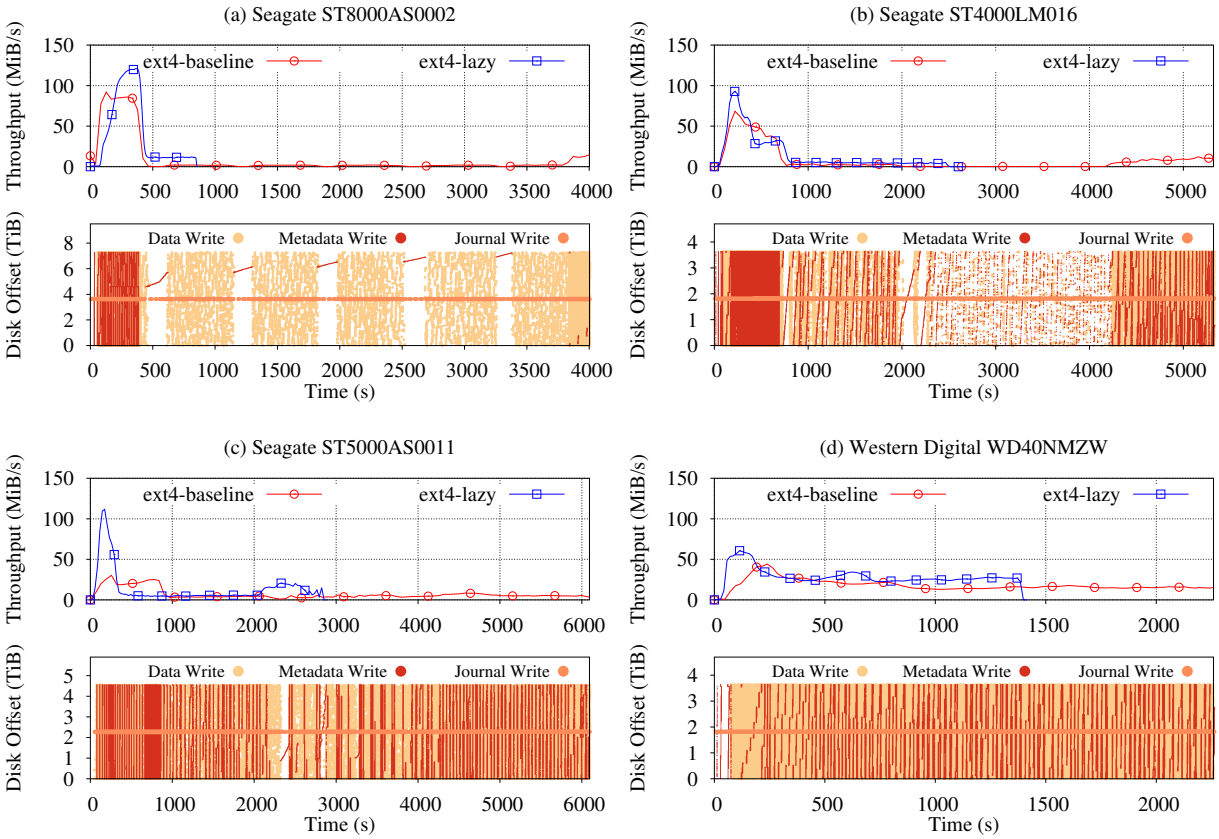


Figure 13: The top graphs show the throughput of four DM-SMR disks on a full disk partition during a Postmark run on ext4-baseline and ext4-lazy. Ext4-lazy provides a speedup of 5.4× 2×, 2×, 1.7× in parts (a), (b), (c), and (d), respectively. The bottom graphs show the offsets of write types during ext4-baseline run. The graphs do not reflect sizes of the writes, but only their offsets.

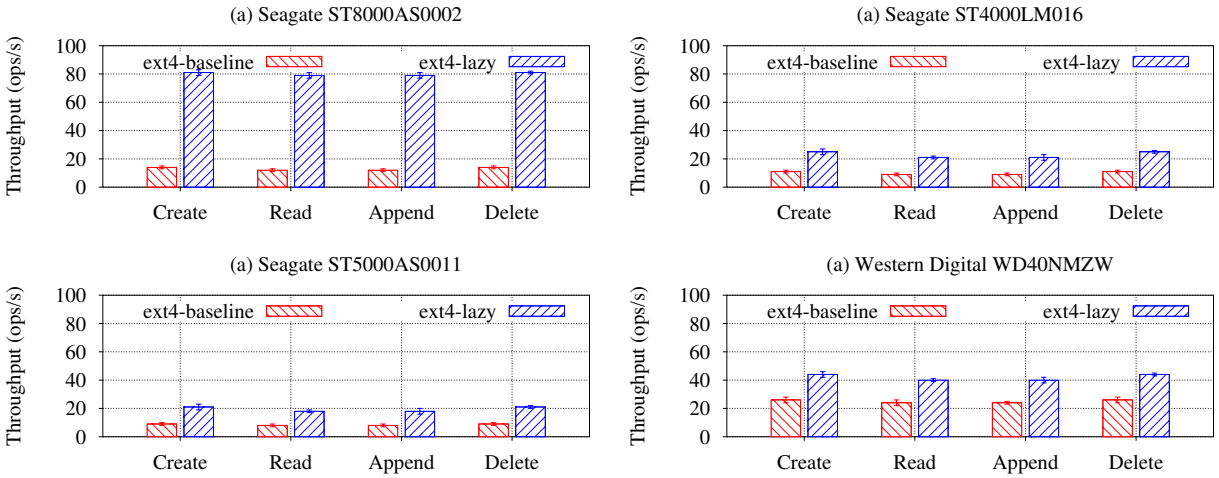


Figure 14: Postmark reported transaction throughput numbers for ext4-baseline and ext4-lazy running on four DM-SMR disks with on a full disk partition. Only includes numbers from the transaction phase of the benchmark.

	Data Writes (MiB)	Metadata Writes (MiB)	Journal Writes (MiB)
ext4-baseline	32,917±9.7	563±0.9	1,212±12.6
ext4-lazy	32,847±9.3	0	1,069±11.4

Table 4: Distribution of write types completed by a ST8000AS0002 DM-SMR disk during a Postmark run on ext4-baseline and ext4-lazy. Metadata writes make up 1.6% of total writes in ext4-baseline, only 1/5 of which is unique.

as seen, for example, between seconds 450-530. During this period we do not see any data writes, because the writeback thread alternates between page cache and buffer cache when writing dirty blocks, and it is the buffer cache's turn. We do, however, see journal writes because jbd2 runs as a separate thread and continues to commit transactions.

The benchmark completes even slower on a full 8 TB partition, as seen in Figure 13 (a), because ext4 spreads the same workload over more bands. With a small partition, updates to different files are likely to update the same metadata region. Therefore, cleaning a single band frees more space in the persistent cache, allowing it to accept more random writes. With a full partition, however, updates to different files are likely to update different metadata regions; now the cleaner has to clean a whole band to free a space for a single block in the persistent cache. Hence, after an hour of ultra-low throughput due to cleaning, it recovers slightly towards the end, and the benchmark completes 5.4× slower on ext4-baseline.

On the ST4000LM016 DM-SMR disk, the benchmark completes 2× faster on ext4-lazy, as seen in Figure 13 (b), because the disk throughput is almost always higher than on ext4-baseline. With ext4-baseline, the disk enters a long period of cleaning with ultra-low throughput starting at second 2000, and recovers around second 4200 completing the benchmark with higher throughput.

We observe a similar phenomenon on the ST5000AS0011 DM-SMR disk, as shown in Figure 13 (c). Unlike with ext4-baseline that continues with a low throughput until the end of the run, with ext4-lazy the cleaning cycle eventually completes and the workload finishes 2× faster.

The last DM-SMR disk in our list, WD40NMZW model found in My Passport Ultra from Western Digital [57], shows a different behavior from previous disks, suggesting a different STL design. We think it is using an S-blocks-like architecture [9] with dynamic mapping that enables cheaper cleaning (§ 2.1). Unlike previous disks that clean only when idle or when the persistent cache is full, WD40NMZW seems to regularly mix cleaning with file system user work. Therefore, its throughput is not as high as the Seagate disks initially, but after the persistent cache becomes full, it does not suffer as sharp of a drop, and its steady-state throughput is higher. Nevertheless, with ext4-lazy the disk achieves 1.4-2.5× increase in throughput over ext4-baseline, depending on the state of the persistent cache, and the benchmark completes 1.7× faster.

Figure 14 shows Postmark transaction throughput numbers for the runs. All of the disks show a significant improvement with ext4-lazy. An interesting observation is that, while with ext4-baseline WD40NMZW is 2× faster than ST8000AS0002, with ext4-lazy the situation is reversed and ST8000AS0002 is 2× faster than WD40NMZW, and fastest overall.

4.4 Performance Overhead

Indirection Overhead: To determine the overhead of in-memory jmap lookup, we populated jmap with 10,000 mappings pointing to random blocks in the journal, and measured the total time to read all of the blocks in a fixed random order. We then measured the time to read the same random blocks directly, skipping the jmap lookup, in the same order. We repeated each experiment five times, starting with a cold cache every time, and found no difference in total time read time—reading from disk dominated the total time of the operation.

Memory Overhead: A single jmap entry consists of a red-black tree node (3×8 bytes), a doubly-linked list node (2×8 bytes), a mapping (12 bytes), and a transaction id (4 bytes), occupying 66 bytes in memory. Hence, for example, a million-entry jmap that can map 3.8 GiB of hot metadata, requires 63 MiB of memory. Although this is a modest overhead for today's systems, it can further be reduced with memory-efficient data structures.

Seek Overhead: The rationale for introducing cylinder groups in FFS, which manifest themselves as block groups in ext4, was to create clusters of inodes that are spread over the disk close to the blocks that they reference, to avoid long seeks between an inode and its associated data [38]. Ext4-lazy, however, puts hot metadata in the journal located at the center of the disk, requiring a half-seek to read a file in the worst case. The TarFiles benchmark (§ 4.2.1) shows that when reading files from a large and deep directory tree, where directory traversal time dominates, putting the metadata at the center wins slightly over spreading it out. To measure the seek overhead on a shallow directory, we created a directory with 10,000 small files located at the outer diameter of the disk on ext4-lazy, and starting with a cold cache creating the tarball of the directory. We observed that since files were created at the same time, their metadata was written sequentially to the journal. The code for reading metadata blocks in ext4 uses readahead since the introduction of flex_bgs. As a result, the metadata of all files was brought into the buffer cache in just 3 seeks. After five repetitions of the experiment on ext4-baseline an ext4-lazy, the average times were 103 seconds and 101 seconds, respectively.

Cleaning Overhead: In our benchmarks, the 10 GiB journal always contained less than 10% live metadata. Therefore, most of the time the cleaner reclaimed space simply by advancing the tail. We kept reducing the journal size and the first noticeable slowdown occurred with a journal size of 1.4 GiB, that is, when the live metadata was ≈ 70% of the journal.

5 Related Work

Researchers have tinkered with the idea of separating metadata from data and managing it differently in local file systems before. Like many other good ideas, it may have been ahead of its time because the technology that would benefit most from it did not exist yet, preventing adoption.

The Multi-Structured File System [39] (MFS) is the first file system proposing the separation of data and metadata. It was motivated by the observation that the file system I/O is becoming a bottleneck because data and metadata exert different access patterns on storage, and a single storage system cannot respond to these demands efficiently. Therefore, MFS puts data and metadata on isolated disk arrays, and for each data type it introduces on-disk structures optimized for the respective access pattern. Ext4-lazy differs from MFS in two ways: (1) it writes metadata as a log, whereas MFS overwrites metadata in-place; (2) facilitated by (1), ext4-lazy does not require a separate device for storing metadata in order to achieve performance improvements.

DualFS [44] is a file system influenced by MFS—it also separates data and metadata. Unlike MFS, however, DualFS uses well known data structures for managing each data type. Specifically, it combines an FFS-like [37] file system for managing data, and LFS-like [47] file system for managing metadata. hFS [61] improves on DualFS by also storing small files in a log along with metadata, thus exploiting disk bandwidth for small files. Similar to these file systems ext4-lazy separates metadata and data, but unlike them it does not confine metadata to a log—it uses a hybrid design where metadata can migrate back and forth between file system and log as needed. However, what really sets ext4-lazy apart is that it is not a new prototype file system; it is an evolution of a production file system, showing that a journaling file system can benefit from the metadata separation idea with a small set of changes that does not require on-disk format changes.

ESB [26] separates data and metadata on ext2, and puts them on CMR disk and SSD, respectively, to explore the effect of speeding up metadata operations on I/O performance. It is a virtual block device that sits below ext2 and leverages the fixed location of static metadata to forward metadata block requests to an SSD. The downside of this approach is that unlike ext4-lazy, it cannot handle floating metadata, like directory blocks. ESB authors conclude that for metadata-light workloads speeding up metadata operations will not improve I/O performance on a CMR disk, which aligns with our findings (§ 4.2.2).

A separate metadata server is the norm in distributed object-based file systems like Lustre [7], Panasas [59], and Ceph [58]. TableFS [46] extends the idea to a local file system: it is a FUSE-based [51] file system that stores metadata in LevelDB [19] and uses ext4 as an object store for large files. Unlike ext4-lazy, TableFS is disadvantaged by FUSE overhead, but still it achieves substantial speedup against production file systems on metadata-heavy workloads.

In conclusion, although it is likely that the above file systems could have taken a good advantage of DM-SMR disks, they could not have shown it because all of them predate the hardware. We reevaluate the metadata separation idea in the context of a technological change and demonstrate its amplified advantage.

6 Conclusion

Our work is the first step in adapting a legacy file system to DM-SMR disks. It shows how effective a well-chosen small change can be. It also suggests that while three decades ago it was wise for file systems depending on the block interface to scatter the metadata across the disk, today, with large memory sizes that cache metadata and with changing recording technology, putting metadata at the center of the disk and managing it as a log looks like a better choice. Our work also rekindles an interesting question: How far can we push a legacy file system to be SMR friendly?

We conclude with the following general takeaways:

- We think modern disks are going to practice more extensive "lying" about their geometry and perform deferred cleaning when exposed to random writes; therefore, file systems should work to eliminate structures that induce small isolated writes, especially if the user workload is not forcing them.
- With modern disks operation costs are asymmetric: Random writes have a higher ultimate cost than random reads, and furthermore, not all random writes are equally costly. When random writes are unavoidable, file systems can reduce their cost by confining them to the smallest perimeter possible.

7 Acknowledgements

We thank anonymous reviewers, Sage Weil (our shepherd), Phil Gibbons, and Greg Ganger for their feedback; Tim Feldman and Andy Kowles for the SMR disks and for their help with understanding the SMR disk behavior; Lin Ma, Prashanth Menon, and Saurabh Kadekodi for their help with experiments; Jan Kara for reviewing our code and for explaining the details of Linux virtual memory. We thank the member companies of the PDL Consortium (Broadcom, Citadel, Dell EMC, Facebook, Google, HewlettPackard Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung, Seagate Technology, Tintri, Two Sigma, Uber, Veritas, Western Digital) for their interest, insights, feedback, and support. This research is supported in part by National Science Foundation under award CNS-1149232.

References

[1] A. Aghayev and P. Desnoyers. Skylight—A Window on Shingled Disk Operation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 135–149, Santa Clara, CA, USA, Feb. 2015. USENIX Association.

[2] T. R. Albrecht, H. Arora, V. Ayanoor-Vitikkate, J.-M. Beaujour, D. Bedau, D. Berman, A. L. Bogdanov, Y.-A. Chapuis, J. Cushen, E. E. Dobisz, et al. Bit-Patterned Magnetic Recording: Theory, Media Fabrication, and Recording Performance. *IEEE Transactions on Magnetics*, 51(5):1–42, 2015.

[3] AskUbuntu. Is there any faster way to remove a directory than rm -rf? http://askubuntu.com/questions/114969.

[4] J. Axboe et al. blktrace. http://git.kernel.org/cgit/linux/kernel/git/axboe/blktrace.git.

[5] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, 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 USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[6] 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 USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[7] P. J. Braam et al. The Lustre Storage Architecture. http://lustre.org/.

[8] R. Card, T. Tso, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the first Dutch International Symposium on Linux*, volume 1, 1994.

[9] Y. Cassuto, M. A. A. Sanvido, C. Guyot, D. R. Hall, and Z. Z. Bandic. Indirection Systems for Shingled-recording Disk Drives. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–14, Washington, DC, USA, 2010. IEEE Computer Society.

[10] S. P. M. Chi-Young Ku. An SMR-aware Append-only File System. In *Storage Developer Conference*, Santa Clara, CA, USA, Sept. 2015.

[11] D. Chinner. SMR Layout Optimization for XFS. http://xfs.org/images/f/f6/Xfs-smr-structure-0.2.pdf, Mar. 2015.

[12] D. Chinner. XFS: There and Back... ...and There Again? In *Vault Linux Storage and File System Conference*, Boston, MA, USA, Apr. 2016.

[13] E. A. Dobisz, Z. Bandic, T.-W. Wu, and T. Albrecht. Patterned Media: Nanofabrication Challenges of Future Disk Drives. *Proceedings of the IEEE*, 96(11):1836–1846, Nov. 2008.

[14] Å. Dragland. Big Data – for better or worse. http://www.sintef.no/en/latest-news/big-data--for-better-or-worse/, May 2013.

[15] J. Edge. Ideas for supporting shingled magnetic recording (SMR). https://lwn.net/Articles/592091/, Apr 2014.

[16] J. Edge. Filesystem support for SMR devices. https://lwn.net/Articles/637035/, Mar 2015.

[17] T. Feldman and G. Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *USENIX ;login issue*, 38(3), 2013.

[18] FreeNAS. ZFS and lots of files. https://forums.freenas.org/index.php?threads/zfs-and-lots-of-files.7925/.

[19] S. Ghemawat and J. Dean. LevelDB. https://github.com/google/leveldb.

[20] G. Gibson and G. Ganger. Principles of Operation for Shingled Disk Devices. Technical Report CMU-PDL-11-107, CMU Parallel Data Laboratory, Apr. 2011.

[21] D. Hall, J. H. Marcos, and J. D. Coker. Data Handling Algorithms For Autonomous Shingled Magnetic Recording HDDs. *IEEE Transactions on Magnetics*, 48(5):1777–1781, 2012.

[22] W. He and D. H. C. Du. Novel Address Mappings for Shingled Write Disks. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, pages 5–5, Berkeley, CA, USA, 2014. USENIX Association.

[23] INCITS T10 Technical Committee. Information technology - Zoned Block Commands (ZBC). Draft Standard T10/BSR INCITS 536, American National Standards Institute, Inc., Sept. 2014. Available from http://www.t10.org/drafts.htm.

[24] C. Jin, W.-Y. Xi, Z.-Y. Ching, F. Huo, and C.-T. Lim. HiSMRfs: a High Performance File System for Shingled Storage Array. In *Proceedings of the 2014 IEEE 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, June 2014.

[25] S. Kadekodi, S. Pimpale, and G. A. Gibson. Caveat-Scriptor: Write Anywhere Shingled Disks. In *7th USENIX Workshop on Hot Topics in Storage and File*

Systems (HotStorage 15), Santa Clara, CA, USA, July 2015. USENIX Association.

[26] J. Kaiser, D. Meister, T. Hartung, and A. Brinkmann. ESB: Ext2 Split Block Device. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, ICPADS '12, pages 181–188, Washington, DC, USA, 2012. IEEE Computer Society.

[27] J. Katcher. Postmark: A New File System Benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997.

[28] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux Implementation of a Log-structured File System. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.

[29] M. Kryder, E. Gage, T. McDaniel, W. Challener, R. Rottmayer, G. Ju, Y.-T. Hsia, and M. Erden. Heat Assisted Magnetic Recording. *Proceedings of the IEEE*, 96(11):1810–1835, Nov. 2008.

[30] A. K. KV, M. Cao, J. R. Santos, and A. Dilger. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium*, volume 1, 2008.

[31] R. Landley. Red-black Trees (rbtree) in Linux. https://www.kernel.org/doc/Documentation/rbtree.txt, Jan. 2007.

[32] D. Le Moal, Z. Bandic, and C. Guyot. Shingled file system host-side management of Shingled Magnetic Recording disks. In *Proceedings of the 2012 IEEE International Conference on Consumer Electronics (ICCE)*, pages 425–426, Jan. 2012.

[33] C. MacCárthaigh. Scaling Apache 2.x beyond 20,000 concurrent downloads. In *ApacheCon EU*, July 2005.

[34] A. Manzanares, N. Watkins, C. Guyot, D. LeMoal, C. Maltzahn, and Z. Bandic. ZEA, A Data Management Approach for SMR. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, USA, June 2016. USENIX Association.

[35] C. Mason. Compilebench. https://oss.oracle.com/~mason/compilebench/.

[36] 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*, volume 2, 2007.

[37] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[38] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2014.

[39] K. Muller and J. Pasquale. A High Performance Multi-structured File System Design. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 56–67, New York, NY, USA, 1991. ACM.

[40] NetBSD-Wiki. How to install a server with a root LFS partition. https://wiki.netbsd.org/tutorials/how_to_install_a_server_with_a_root_lfs_partition/.

[41] A. Palmer. SMRFFS-EXT4—SMR Friendly File System. https://github.com/Seagate/SMR_FS-EXT4.

[42] A. Palmer. SMR in Linux Systems. In *Vault Linux Storage and File System Conference*, Boston, MA, USA, Apr. 2016.

[43] PerlMonks. Fastest way to recurse through VERY LARGE directory tree. http://www.perlmonks.org/?node_id=883444.

[44] J. Piernas. DualFS: A New Journaling File System without Meta-data Duplication. In *In Proceedings of the 16th International Conference on Supercomputing*, pages 137–146, 2002.

[45] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, USA, April 2005.

[46] K. Ren and G. Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156, San Jose, CA, USA, 2013. USENIX.

[47] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 1–15, New York, NY, USA, 1991. ACM.

[48] R. Santana, R. Rangaswami, V. Tarasov, and D. Hildebrand. A Fast and Slippery Slope for File Systems. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 5:1–5:8, New York, NY, USA, 2015. ACM.

[49] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin. An Implementation of a Log-structured File System for UNIX. In *Proceedings of the USENIX Winter 1993 Conference*, USENIX'93, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[50] ServerFault. Doing an rm -rf on a massive directory tree takes hours. http://serverfault.com/questions/46852.

[51] M. Szeredi et al. FUSE: Filesystem in userspace. https://github.com/libfuse/libfuse/.

[52] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A Flexible Framework for File System Benchmarking. *USENIX ;login issue*, 41(1), 2016.

[53] L. Torvalds and P. Zijlstra. __wb_calc_thresh. http://lxr.free-electrons.com/source/mm/page-writeback.c?v=4.6#L733.

[54] T. Ts'o. Release of e2fsprogs 1.43.2. http://www.spinics.net/lists/linux-ext4/msg53544.html, Sept. 2016.

[55] S. C. Tweedie. Journaling the Linux ext2fs Filesystem. In *The Fourth Annual Linux Expo*, Durham, NC, USA, May 1998.

[56] J. Wan, N. Zhao, Y. Zhu, J. Wang, Y. Mao, P. Chen, and C. Xie. High Performance and High Capacity Hybrid Shingled-Recording Disk System. In *2012 IEEE International Conference on Cluster Computing*, pages 173–181. IEEE, 2012.

[57] WDC. My Passport Ultra. https://www.wdc.com/products/portable-storage/my-passport-ultra-new.html, July 2016.

[58] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[59] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

[60] R. Wood, M. Williams, A. Kavcic, and J. Miles. The Feasibility of Magnetic Recording at 10 Terabits Per Square Inch on Conventional Media. *IEEE Transactions on Magnetics*, 45(2):917–923, Feb. 2009.

[61] Z. Zhang and K. Ghose. hFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 175–187, New York, NY, USA, 2007. ACM.

[62] P. Zijlstra. sysfs-class-bdi. https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-class-bdi, Jan. 2008.