Evolving Ext4 for Shingled Disks

ABUTALIB AGHAYEV, THEODORE TS'O, GARTH GIBSON, AND PETER DESNOYERS

Abutalib Aghayev is a PhD student in the Computer Science Department at Carnegie Mellon University. His research interests include operating systems, file and storage systems, and, recently, distributed machine learning systems. agayev@cs.cmu.edu

Theodore Ts'o started working with Linux in September 1991 and is the first North American Linux kernel developer. He also served as the tech lead for the MIT Kerberos V5 development team and as a chair of the IP Security working group at the IETF. He previously was CTO for the Linux Foundation and is currently employed at Google. Theodore is a Debian Developer, and maintains the ext4 file system in the Linux kernel. He is also the maintainer and original author of the e2fsprogs userspace utilities for the ext2, ext3, and ext4 file systems. tytso@thunk.org

Garth Gibson is a Professor of Computer Science and an Associate Dean in the School of Computer Science at Carnegie Mellon University. Garth's research is split between scalable storage systems and distributed machine learning systems, and he has had his hand in the creation of the RAID taxonomy, the Panasas PanFS parallel file system, the IETF NFS v4.1 parallel NFS extensions, and the USENIX Conference on File and Storage Technologies (FAST). garth@cs.cmu.edu

Multi-terabyte hard disks today use Shingled Magnetic Recording (SMR), a technique that increases capacity at the expense of more costly random writes. We introduce ext4-lazy, a small change to the popular Linux ext4 file system that eliminates a major source of random writes—the metadata writeback—significantly improving performance on SMR disks in general, as well as on conventional disks for metadata-heavy workloads in particular. In this article, we briefly explain why SMR disks suffer under random writes and how ext4-lazy helps.

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. Shingled Magnetic Recording (SMR) is one such technique that allows disk manufacturers to increase areal density with existing fabrication methods. So far, the industry has introduced two kinds of SMR disks: Drive-Managed (DM-SMR) and Host-Managed (HM-SMR). HM-SMR disks have a novel backward-incompatible interface that requires changes to the I/O stack and, therefore, are not widely deployed. DM-SMR disks, on the other hand, are a drop-in replacement for Conventional Magnetic Recording (CMR) disks that offer high capacity with the traditional block interface. Millions of DM-SMR disks have been shipped; in the rest of the article, therefore, we will use SMR disk as a shorthand for DM-SMR disk.

If you buy a multi-terabyte disk today, there is a good chance that it is an SMR disk in disguise, which is easy to tell: unlike CMR disks, SMR disks suffer performance degradation when subjected to continuous random write traffic, as Figure 1 shows.

One approach to adopting SMR disks is to develop a file system from scratch based on their performance characteristics. But file systems are complex and critical pieces of code that take years to mature. Therefore, we take an evolutionary approach to adopting these disks: we make a small change to the popular Linux file system, ext4, that significantly improves its performance on SMR disks by avoiding random metadata writes.

We introduce a simple technique that we call *lazy writeback journaling,* and we call a version of ext4 using our journaling technique *ext4-lazy*. Like other journaling file systems, by default ext4 writes metadata twice; as Figure 2a shows, it first writes the metadata block to a temporary location *J* in the journal and then marks the block as *dirty* in memory. Once the block has been in memory for long enough, a *writeback* 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. 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 2b. Since the journal is written sequentially to a *circular* log, overwriting a metadata block is not possible. Therefore, ext4-lazy writes an updated block to the head of the log, updating the map and invalidating the old copy of the block. Ext4-lazy uses a large journal so that it can continue writing updated blocks while reclaiming the space from the

Peter Desnoyers is an Associate Professor at Northeastern University. He worked for Apple, Motorola, and a number of startups for 15 years before getting his PhD at the University of Massachusetts, Amherst, in 2007. He received BS and MS degrees from MIT. His main focuses are storage, particularly the integration of emerging storage technologies, and cloud computing. pjd@ccs.neu.edu

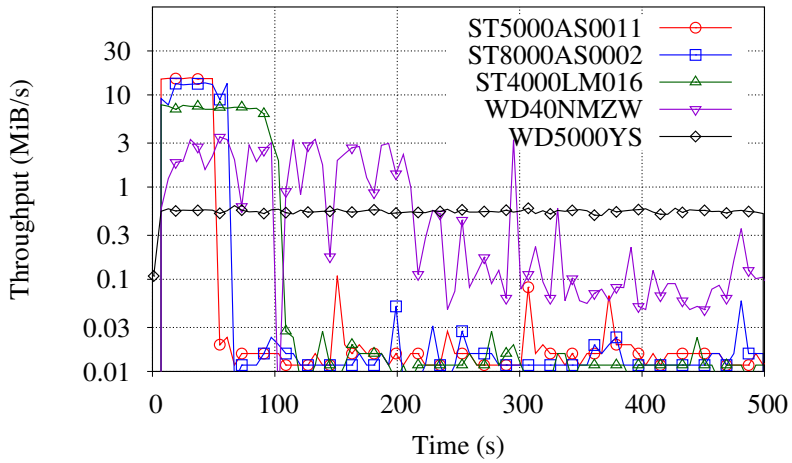

Figure 1: Throughput of CMR and SMR disks from Table 1 under 4 KiB random write traffic. The CMR disk (WD500YS) has a stable but low throughput under random writes. SMR disks, on the other hand, have a short period of high throughput followed by a continuous period of ultra-low throughput.

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

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

invalidated blocks. During mount, it reconstructs the in-memory map from the journal resulting in a modest increase in mount time. Results show that ext4-lazy significantly improves performance on SMR disks in general, as well as on CMR disks for metadata-heavy workloads in particular.

Our main contribution to ext4 includes the design, implementation, and evaluation of ext4-lazy on SMR and CMR disks. The change we make is minimally invasive—we modify 80 lines of existing code and introduce the new functionality in additional files totaling 600 lines of C code. As we show in the evaluation section, even on a metadata-light file server benchmark where the metadata writes make up less than 1% of total writes, with stock ext4 the SMR disk appears unresponsive for almost an hour with near-zero throughput. With ext4-lazy, on the other hand, the SMR disk does not suffer such a behavior and completes 1.7–5.4x faster. For directory traversal and metadata-heavy workloads, ext4-lazy achieves 2–13x improvement on both SMR and CMR disks.

Background

A high-level introduction to SMR technology has been previously presented in *;login:* [3]. Readers interested in nitty-gritty details of how an SMR disk works and why it suffers under random writes may refer to the detailed study [1] of one such disk. Here, we give just enough background on SMR disks and ext4 journaling to make the rest of the article understandable.

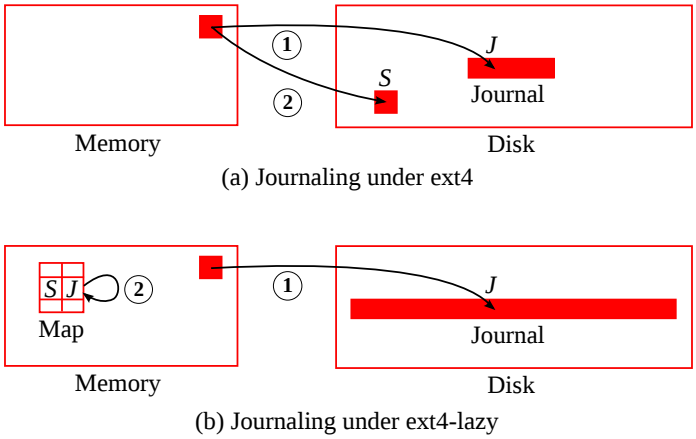

Figure 2: (a) Ext4 writes a metadata block to disk twice. It first writes the metadata block to the journal at some location *J* and marks it dirty in memory. Later, the writeback thread writes the same metadata block to its static location *S* on disk, resulting in a random write. (b) Ext4-lazy writes 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.

SMR

As a concrete example, one SMR disk used in our evaluation consists of ≈ 30 MiB bands that are the smallest units that must be written sequentially. Overwriting a random block in a band requires read-modify-write (RMW) of the whole band. This results in reading a band, modifying it in memory, writing the updated band to a temporary band (since overwriting the original band is not atomic and could corrupt the old data if power is lost), and finally overwriting the original band, generating ≈ 90 MiB disk I/O. To hide the cost of random writes, the disk uses a *persistent cache* for handling bursts of random writes—incoming random writes are written to the persistent cache, and the bands are updated using RMW during the idle times, emptying the persistent cache. If the burst of random writes is large enough to fill the persistent cache, the throughput of the disk drops because every incoming write requires RMW of the corresponding band. Sequential writes, on the other hand, are detected and written directly to bands, bypassing the persistent cache.

Ext4 and Journaling

The ext4 file system evolved from ext2, which was influenced by Fast File System (FFS). 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 3a shows. 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*—a set of contiguous block groups whose metadata is consolidated in the first 16 MiB of the first block group within the set, as shown in Figure 3b.

Ext4 ensures metadata consistency via journaling, but it does not implement journaling itself; rather, it uses a generic kernel layer called the *Journaling Block Device* 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 five seconds) into a *transaction* buffer and atomically *commits* the transaction to the journal—a circular log of transactions. After a commit, jbd2 marks the in-memory copies of metadata blocks as dirty so that the writeback thread would write them to their static locations.

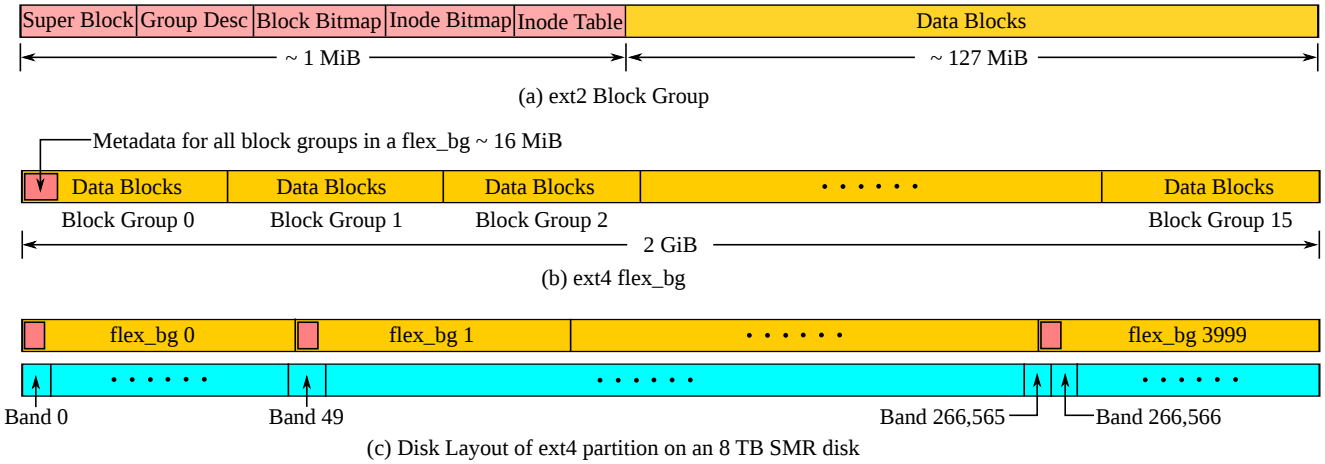

Figure 3: (a) In ext2, the first megabyte of a 128 MiB block group contains the metadata blocks describing the block group, and the rest is data blocks. (b) In ext4, a single flex bg (flexible block group) concatenates multiple (16 in this example) block groups into one giant block group and puts all 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.

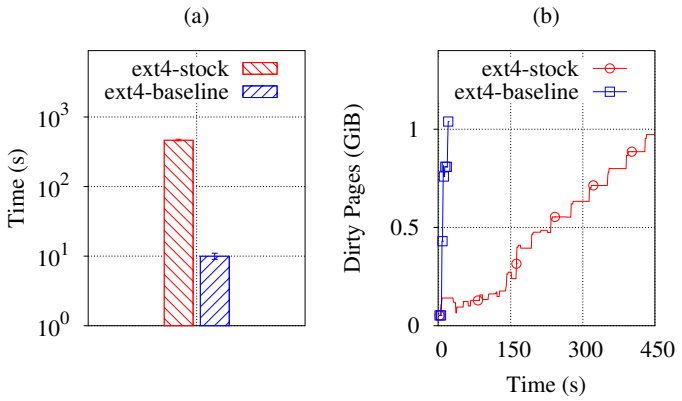

Figure 4: (a) Completion time for a benchmark creating 100,000 files on ext4-stock (ext4 with 128 MiB journal) and on ext4-baseline (ext4 with 10 GiB journal). (b) The volume of dirty pages during benchmark runs obtained by sampling `/proc/meminfo` every second.

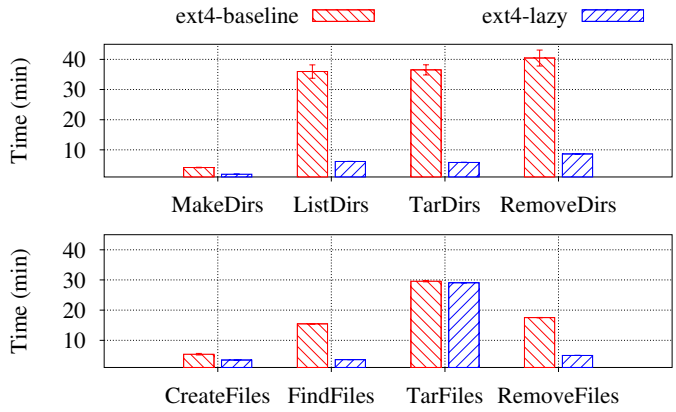

Figure 5: Microbenchmark runtimes on ext4-baseline and ext4-lazy

On 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 3c. 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.

Design of Ext4-lazy

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 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 2b.

Indirection: In ext4-lazy, all accesses to metadata blocks go through the map. If the most recent version of a block is in the journal, there will be an entry in the map 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 that have become 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 map.

Evaluation

We evaluate ext4-lazy 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, using the disks listed in Table 1. One surprising finding of our work was that the default journal size on ext4 is a bottleneck for metadata-heavy workloads. Figure 4 shows that just by increasing the journal size, a metadata-heavy workload completes over 40x faster. As a result, the recent version of e2fsprogs has increased the default journal size from 128 MiB to 1 GiB for file systems over 128 GiB. Readers interested in the details may refer to our paper [2]. Since enabling a large journal on ext4 is a command-line option to `mkfs`, we choose ext4 with a 10 GiB journal as our baseline.

Next, we first show that ext4-lazy achieves significant speedup on the CMR disk WD5000YS from Table 1 for metadata-heavy workloads, and specifically for massive directory traversal workloads. We then show that on SMR disks, ext4-lazy provides significant improvement on both metadata-heavy and metadata-light workloads.

Ext4-lazy on a CMR Disk

For metadata-heavy workloads we use the following benchmarks. *MakeDirs* creates 800,000 directories in a directory tree of depth 10. The directory tree is also used by the following benchmarks: *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 files each of size 4 KiB in a new directory tree of depth 20. The directory tree is also used by the following benchmarks: *FindFiles* runs find on the directory tree, *TarFiles* creates a tarball of the directory tree, and *RemoveFiles* removes the directory tree. All of the benchmarks start with a cold cache, set up by echoing "3" to /proc/sys/vm/drop_caches.

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

	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

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 6a; 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 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 6b. 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 6b 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 6c, while on ext4-lazy they are within the 10 GiB region in the journal, resulting in only short seeks, as Figure 6d 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 is 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. One of the biggest benefits of consolidating metadata in a small region is an order-of-magnitude improvement in such workloads.

Ext4-lazy on SMR Disks

An additional critical factor for file systems when running on 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 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.

Microbenchmarks

Figure 7 shows results of the microbenchmarks (see section "Ext4-lazy on a CMR Disk") repeated on ST8000AS0002 with a 2 TB partition, on ext4-baseline and ext4-lazy. MakeDirs and CreateFiles do not fill the persistent cache, and, therefore, they typically complete 2–3x faster than on CMR disk. Similar to CMR disk, MakeDirs and CreateFiles are 1.5–2.5x faster on ext4-lazy. On the other hand, ListDir, for example, one of the remaining directory traversal benchmarks, completes 13x faster on ext4-lazy, as compared to 5x faster on CMR disk.

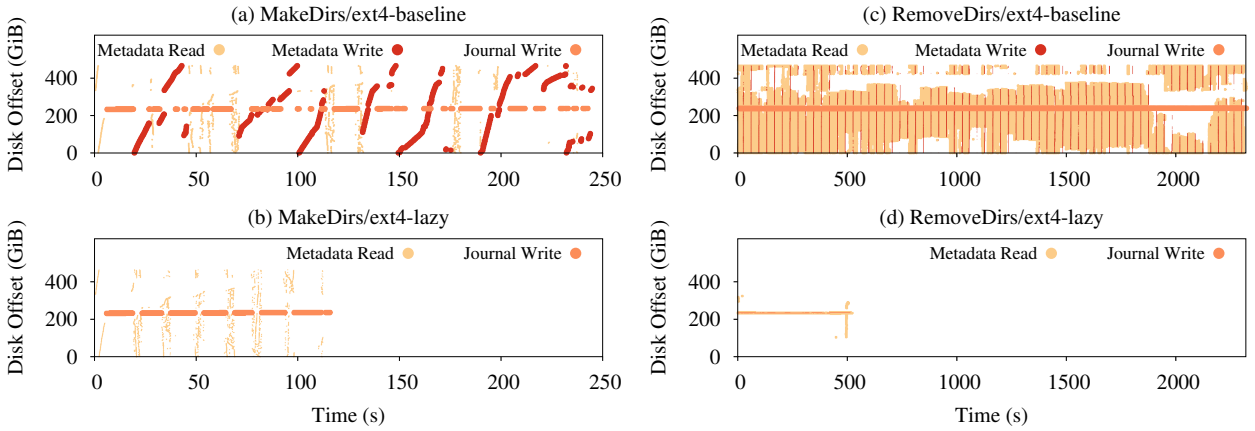

Figure 6: Disk offsets of I/O operations during MakeDirs and RemoveDirs microbenchmarks on ext4-baseline and ext4-lazy. Metadata reads and writes are spread out while journal writes are at the center. The dots have been scaled based on the I/O size. In part (d), journal writes are not visible due to low resolution. These are pure metadata workloads with no data writes.

The cleaning times for ListDirs, FindFiles, TarDirs, and TarFiles are zero because they do not write to disk—TarDirs and TarFiles write their output to a different disk. 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 6b 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 and 20 seconds, respectively, on ext4-lazy compared to 590 and 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.

File Server Macrobenchmark

Our file server benchmark 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. In total, the benchmark writes 37.89 GiB and reads 31.54 GiB of data from user space.

Table 3 shows the distribution of write types completed by a ST8000AS0002 SMR disk with a 400 GB partition during the benchmark. 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.

The benchmark completes more than 2x faster on ext4-lazy, in 461 seconds, as seen in Figure 8. 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 reduces 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, as seen, for example, between seconds 450 and 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 more slowly on a full 8 TB ext4 partition, as seen in Figure 9, 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

	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 3: Distribution of write types completed by a ST8000AS0002 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 20% of which is unique.

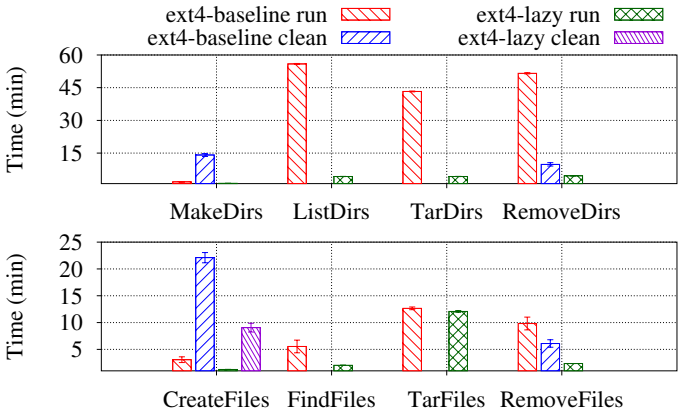

Figure 7: Microbenchmark runtimes and cleaning times on ext4-baseline and ext4-lazy running on an SMR disk. Cleaning time is the additional time after the benchmark run that the SMR disk was busy cleaning.

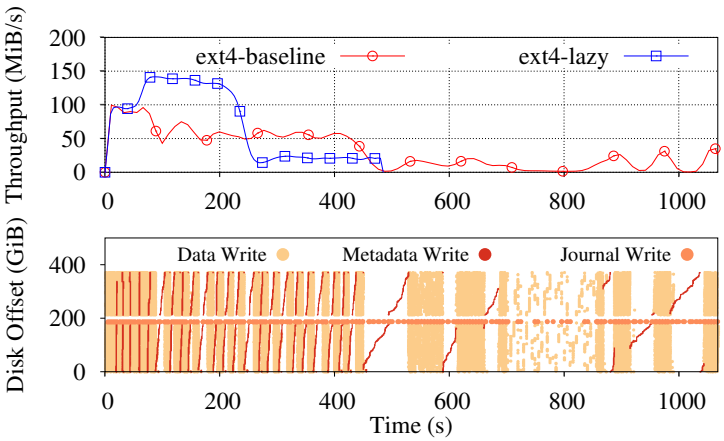

Figure 8: The top graph shows the throughput of a ST8000AS0002 SMR disk with a 400 GB partition during a file server benchmark run on ext4-baseline and ext4-lazy. The bottom graph shows the offsets of write types during the run on ext4-baseline. The graph does not reflect sizes of the writes, only their offsets.

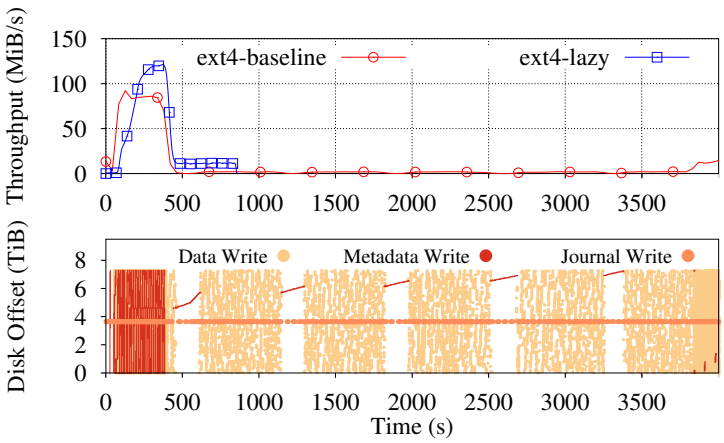

Figure 9: The top graph shows the throughput of a ST8000AS0002 SMR disk with a full 8 TB partition during a file server benchmark run on ext4-baseline and ext4-lazy. The bottom graph shows the offsets of write types during the run on ext4-baseline. The graph does not reflect sizes of the writes, only their offsets.

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 5.4x slower on ext4-baseline. Interested readers may refer to our paper [2] for the evaluations of all disks from Table 1.

Conclusion

Previous work has explored separating metadata from data and managing it as a log by designing a file system from scratch [4–6]. Our work, however, is the first that leverages the metadata separation idea for adapting a legacy file system to 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.

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.

References

[1] A. Aghayev and P. Desnoyers, "Skylight—A Window on Shingled Disk Operation," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pp. 135–149: https://www.usenix.org/system/files/conference/fast15/fast15-paper-aghayev.pdf.

[2] A. Aghayev, T. Ts'o, G. Gibson, and P. Desnoyers, "Evolving Ext4 for Shingled Disks," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pp. 105–120: https://www.usenix.org/system/files/conference/fast17/fast17-aghayev.pdf.

[3] T. Feldman and G. Gibson, "Shingled Magnetic Recording: Areal Density Increase Requires New Data Management," *;login*, vol. 38, no. 2 (June 2013), pp. 22–30: http://www.pdl.cmu.edu/PDL-FTP/HECStorage/05_feldman_022-030.pdf.

[4] J. Piernas, T. Cortes, and J. Garcia, "DualFS: A New Journaling File System without Meta-Data Duplication," in *Proceedings of the 16th International Conference on Supercomputing*, 2002, pp. 137–146: http://ditec.um.es/web-ditec/ficheros/publicaciones/publicacion95.pdf.

[5] K. Ren and G. Gibson, "TableFS: Enhancing Metadata Efficiency in the Local File System," in *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pp. 145–156: http://www.pdl.cmu.edu/PDL-FTP/FS/CMU-PDL-13-102.pdf.

[6] 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 (EuroSys '07)*, pp. 175–187: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.160.507&rep=rep1&type=pdf.