Filesystem Aging: It's more Usage than Fullness

Alex Conway[1], Eric Knorr[1], Yizheng Jiao[2], Michael A. Bender[3], William Jannen[4], Rob Johnson[5], Donald Porter[2], and Martin Farach-Colton[1]

[1]Rutgers University, [2]UNC, [3]Stony Brook University, [4]Williams College, [5]VMware Research

Abstract

Filesystem fragmentation is a first-order performance problem that has been the target of many heuristic and algorithmic approaches. Real-world application benchmarks show that common filesystem operations cause many filesystems to fragment over time, a phenomenon known as ***filesystem aging***.

This paper examines the common assumption that space pressure will exacerbate fragmentation. Our microbenchmarks show that space pressure can cause a substantial amount of inter-file and intra-file fragmentation. However, on a "real-world" application benchmark, space pressure causes fragmentation that slows subsequent reads by only 20% on ext4, relative to the amount of fragmentation that would occur on a file system with abundant space. The other file systems show negligible additional degradation under space pressure.

Our results suggest that the effect of free-space fragmentation on read performance is best described as accelerating the filesystem aging process. The effect on write performance is non-existent in some cases, and, in most cases, an order of magnitude smaller than the read degradation from fragmentation cause by normal usage.

1 Introduction

Over the course of ordinary use, filesystems tend to become fragmented, or ***age***. Fragmentation causes performance declines in many filesystem operations, including file reads, writes, and recursive directory traversals. Aging tends to manifest as a pervasive performance decline.

Because aging is sensitive to usage patterns and can happen slowly over time, it is notoriously difficult to study. Smith and Seltzer [15] studied the real-world performance of the Fast Filesystem (FFS) over the course of 3 years and showed substantial filesystem aging. However, the cost and time of repeating such a study on each new filesystem, storage device and user workload render this approach infeasible. However, their findings inspired work on tools to synthetically age file systems [6, 17].

Because aging is difficult to repeatably measure, a certain amount of folklore has emerged. Many practitioners believe that filesystem aging is already solved and claim that aging only occurs under adversarial workloads or when the disk is full. For example Wikipedia claims that, as time progresses and the free space fragments "...the filesystem is no longer able to allocate new files contiguously, and has to break them into fragments. This is especially true when the filesystem becomes full and large contiguous regions of free space are unavailable." [3]

Recent work by Conway et al. [8] and Kadekodi et al. [12] showed that this optimism was unfounded. They showed that modern filesystems do age, often severely, even on SSDs, even under realistic workloads, even when the disk is far from full. For example, Conway et al. showed that ext4 could exhibit slowdowns of over 20× on a realistic git-based workload on a nearly empty hard drive, and slowdowns of $2-4\times$ on an SSD.

The goal of this paper is to tease out the effect of disk fullness on filesystem fragmentation. We first use a synthetic benchmark designed to stress the worst-case, full-disk behavior of the filesystem. We find that, on an HDD, the synthetic benchmark can cause ext4 to fragment far worse on a full disk than on a nearly empty one. For the other filesystems, a full disk roughly doubles the read performance degradation from fragmentation. On SSDs, disk fullness has a modest effect on read degradation from fragmentation (typically less than 20%), except on BTRFS.

We then measure a more realistic, git-based application workload, in which disk fullness degrades read performance by only about 20% for ext4 on HDD compared to normal fragmentation. It has a negligible impact for BTRFS and XFS on HDD, and for all filesystems on SSD.

In summary, our results show that file and directory allocations on modern filesystems can be severely fragmented in the course of normal usage, even with plenty of contiguous free space. Although the fragmentation is worsened under space pressure in a few stress tests, the behavior is inconsistent across file systems. Our results suggest that normal usage will erode performance before space pressure and the resulting free-space fragmentation has the opportunity. An interesting question for future work is whether space pressure is more of a problem on a filesystem which resists use-aging, such asx BεtrFS [8, 10, 16].

2 Related Work

Prior work can be broadly classified into two categories: aging studies and anti-aging techniques.

Reproducing and studying aging. It takes years to collect years of traces from live systems. Moreover, traces are large, idiosyncratic, and may contain sensitive data. Consequently, researchers have created synthetic benchmarks to simulate aging. These benchmarks create files and directories and perform normal filesystem operations to induce aging [5]. Once aged, a filesystem can be profiled using other benchmarking tools to understand how an initial aged state affects *future* operations.

Ji et al. [11] studied filesystem fragmentation on mobile devices, confirming that fragmentation causes performance degradation on mobile devices and that existing defragmentation techniques are ineffective.

Anti-aging strategies. The perception that aging is a solved problem is likely due to the abysmal aging of the once ubiquitous FAT filesystem. FAT tracks file contents using a linked list, so a cold-cache random read is particularly expensive when compared to modern tree-structured filesystems. Moreover, most FAT implementations have no heuristics to combat aging; lack read-ahead or other latency-hiding heuristics; and, on earlier PCs with limited DRAM, suffered frequent cache misses [9]. Even on more modern systems, write performance on an aged FAT filesystem degrades by two orders of magnitude [14]. As a result, users were trained to defragment hard drives using an offline utility, which rendered a noticeable performance improvement. Although the works above show that modern filesystems also age, the aging is not as extreme as users experienced with FAT.

Block groups are an anti-aging strategy introduced in FFS [13] and adopted by modern filesystems like ext4. Another common strategy is to pre-allocate contiguous file regions, either by inference or via explicit interfaces like `fallocate()`. Finally, BetrFS uses large (2–4 MB) nodes to group related data, and it incrementally rewrites data to preserve locality as the filesystem evolves [8].

3 Free-Space Fragmentation

This section explains how disk fullness can impact free-space fragmentation, how free-space fragmentation can impact filesystem performance, and how free-space fragmentation can lead to other kinds of fragmentation. The observations in this section will be used in the next section to design benchmarks for measuring the impact of disk fullness on filesystem aging.

Filesystem fragmentation. Several types of disk blocks should be kept together to improve filesystem performance. ***Fragmentation*** occurs when related blocks become scattered. We categorize fragmentation by the types of blocks and their relationships:

Intrafile: fragmentation of allocated blocks in a file.
Interfile: fragmentation of the blocks of small files in the same directory.
Free-space: fragmentation of unused disk blocks.

The first two types of fragmentation directly impact the read performance of a filesystem and together can be referred to as ***read aging***. During a scan of filesystem data, fragmented blocks will incur non-sequential reads, which on most modern storage hardware are considerably slower than sequential reads.

Free-space fragmentation, disk fullness, and write performance. This paper focuses on disk fullness and any resulting free-space fragmentation, which can have a direct effect on write performance and an indirect effect on read performance. When free space is fragmented, the filesystem must choose between scattering new data among the existing free-space fragments (fewer writes, slower future reads) or migrating old data to coalesce free-space fragments (more writes, faster future reads). If a filesystem fragments incoming writes, then free-space fragmentation gets turned into regular intra- and inter-file fragmentation, *i.e.,* read aging. A fragmented write is also slower than when free space is unfragmented, as one write is split into discrete I/Os. If the filesystem compacts the free space by moving data, the compaction slows the write operation. In either case, free-space fragmentation degrades write performance.

Note that intra- and inter-file fragmentation can exacerbate free-space fragmentation, and vice versa: fragmented files, when deleted, produce fragmented free space.

As the disk becomes full, free-space fragmentation tends to worsen. If the filesystem coalesces free-space fragments, combining several small fragments into one large fragments may involve copying already-allocated data multiple times to avoid fragmenting it [7]. This cost is inversely proportional to the fraction of free space available on the disk. Even on systems that don't coalesce free-space fragments, fuller disks simply have more allocated objects and less free space, so it becomes more difficult to co-locate related data.

Free-space fragmentation and read performance. Free-space fragmentation differs from the other types of fragmentation in that it doesn't immediately impact read performance. Because HDDs and most types of SSDs have faster sequential reads than random reads, inter- and intra-file fragmentation causes scans to be slower [4]. Free-space fragmentation, on the other hand, doesn't affect read performance, since free space is not accessed during a scan.

Summary. Though the different forms of fragmentation are interdependent, we can cleanly measure each type at any single moment in time. We can measure free-space fragmentation directly on ext4 using e2freefrag. For the

other file systems, the allocated and free space can be inferred by scanning the data with a cold cache and using a tool such as blktrace [1] to see which blocks are read, but we do not pursue this approach here. We can measure intra- and inter-file fragmentation by measuring read performance. Due to the complex feedback described above, we might expect that disk fullness will affect both free-space and intra- and inter-file fragmentation, and hence will affect read and write performance.

4 Measuring Fragmented Performance

The goal of our benchmarks is to understand how disk fullness affects filesystem fragmentation, and the subsequent read and write performance. In particular:

- Does fullness make fragmentation worse? If so, how much? In other words, is disk fullness a main driver of fragmentation, a second-order driver, or completely overwhelmed by other fragmentation factors?
- Do full disks age faster than empty ones? Do full disks age more overall?
- Does disk fullness affect fragmentation under realistic workloads? If not, can synthetic workloads demonstrate a significant connection between fullness and subsequent read or write degradation?

Empty, full, and unaged disks. We measure the effects of fragmentation and full disks as follows. We first run a workload generator on a small partition (the "full disk" case). This generator may create, delete, rename, write, etc., files. It measures the disk fullness and ensures that, after initial setup, the partition is always above a certain level of fullness. We record the sequence of operations (such as git pulls or file deletions) performed and then replay them on a much larger partition (the "empty disk" case). Thus the empty and full partitions go through the exact same sequence of logical filesystem states.

We run the test on the full partition, the empty partition, and on a fresh (large) partition to which we have copied the current state (the "unaged disk" case). The unaged partition thus provides the baseline performance of an unaged version of the same filesystem state.

Measuring read fragmentation: The Grep Test. To measure fragmentation, we periodically pause the workload generator and run a ***grep test***. This is the wall-clock time it takes to perform a recursive grep on the root directory of the filesystem. This performs a recursive scan through the data and metadata. Fragmentation will cause this scan to be less sequential. Because the filesystems change over time, we report this time normalized by the filesystem size as reported by `du -s`.

Measuring write fragmentation. To measure write fragmentation, we use the wall-clock latency of new writes. We check that the workload is not CPU-bound.

Measuring free-space fragmentation We measure free-space fragmentation directly on ext4 using the e2freefrag tool [2]. This tool reports a histogram of the sizes of free extents. We do not directly measure the free-space fragmentation on XFS, BTRFS or F2FS.

Experimental setup. Each experiment compares three filesystems on HDD: BTRFS, ext4 and XFS, and additionally F2FS on SSD, using the versions in Ubuntu 14.04.5 LTS, kernel version 3.11.10 with default settings. The benchmarks do not use O_DIRECT, and therefore the filesystems may use the page cache.

We use Dell PowerEdge T130 machines with 4-core 3.00 GHz Intel(R) Xeon(R) E3-1220 v6 CPU, 16 GB RAM, two 500 GB, 7,200 RPM TOSHIBA HDDs and a 250 GB Samsung SSD 860.

5 Experimental Results

In this section we describe the benchmarks used to generate free-space fragmentation and the results of running them on several popular filesystems.

Free-space fragmentation microbenchmark (FSFB). FSFB is a worst-case microbenchmark, designed to induce severe free-space fragmentation. FSFB first fills a filesystem with many small files. Next, it randomly selects files for deletion and creates a new directory with the same total size as the deleted files. Deleting small files creates fragmented free space, across which the new directory will need to be allocated.

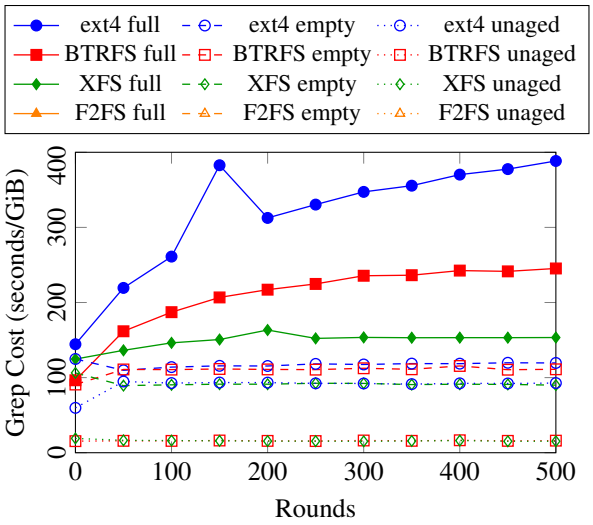
FSFB starts by creating a random directory structure with 1000 directories. Then it creates files by randomly selecting a directory and creating a file there with size chosen randomly between 1KiB and 150KiB. This process creates the files out-of-directory-order, so that the initial layout is "pre-aged." This process repeats until the filesystem reaches the target level of fullness.

FSFB then ages the filesystem through a series of ***replacement rounds***. In a replacement round, 5% of the files, by size, are removed at random and then replaced by new files of equivalent total size in a newly created directory in a random location.

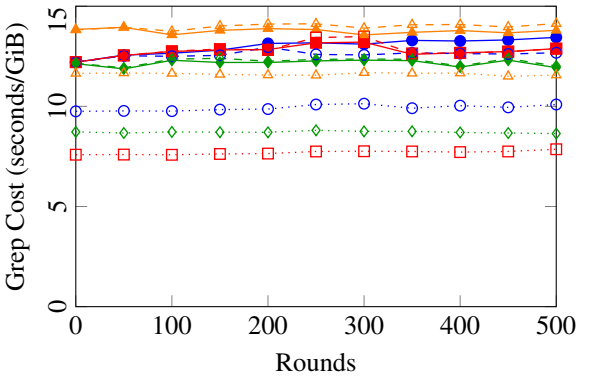
FSFB read aging. We run the microbenchmark with a target fullness of 95% on a 5GB partition. We then age the filesystem for 500 replacement rounds, performing a grep test every 50 rounds. We then replay the benchmark on a 50GB partition, so that it is at most 10% full ("empty"). We also create an "unaged" version by copying the data to a fresh partition.

Figure 1a shows the HDD results. All filesystems are slower in the full-disk case than the empty-disk case. However, BTRFS and XFS slow-down far more from unaged to aged than from empty to full. ext4, in contrast, only loses read performance under space pressure.

Figure 1b shows the SSD results. The additional read aging from disk fullness is negligible.



(a) Grep performance on HDD under FSFB. By the end, all full file systems are slower than empty by $1.5-4\times$, XFS and BTRFS are $7\times$ slower empty than unaged.



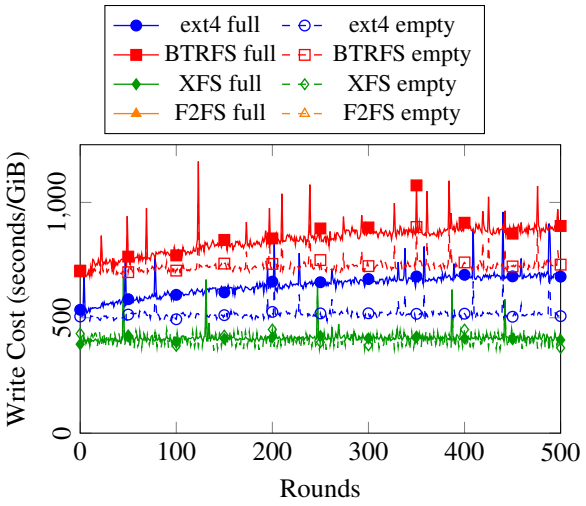
(b) Grep performance on SSD under FSFB. The full filesystems show no discernable slowdown compared to empty, however the empty ones are 25-50% slower than unaged.

Figure 1: Read performance under FSBS on a 95% full "full" disk, a 10% full "empty" disk, and an "unaged" copy. Lower is better.

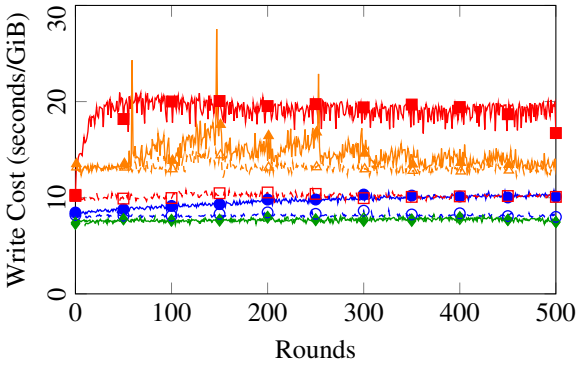
FSFB write aging. We measure write aging by measuring the wall-clock time to create each new directory of files during a replacement round.

Figure 2a shows that, on an empty hard drive, none of the filesystems exhibit any write aging beyond the initial filesystem construction. When the disk is full, ext4 has 40% higher write costs, BTRFS has 25% higher write costs, and XFS has essentially the same costs. Thus disk fullness does induce some write aging, but it is an order of magnitude less than read aging on an empty disk.

On SSDs (Figure 2b), XFS is slightly faster when the disk is full, ext4 exhibits a modest 25% slowdown between the empty an full cases, BTRFS rapidly loses half its performance in the full-disk case, and F2FS has erratic but generally only slightly slower performance. Again,



(a) Write performance on HDD under FSBS. ext4 slows by 40% and BTRFS by 25%. XFS performance is almost unchanged.



(b) Write performance on SSD under FSFB. The four filesystems exhibit different behaviors.

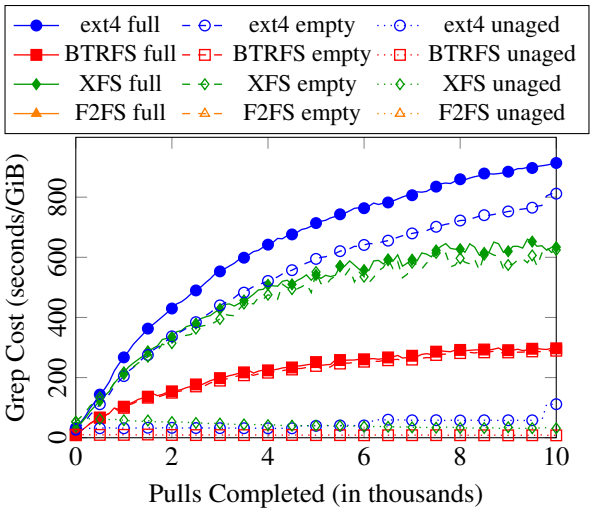
Figure 2: Write performance under FSFB on a 95% full "full" disk and a 10% full "empty" disk. Lower is better.

except possibly for BTRFS, the performance differences between an empty and full SSD are smaller than the read aging performance losses on an empty disk.

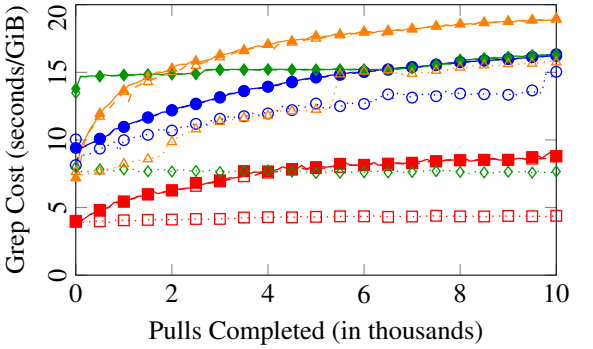
As with the read aging effect of disk fullness, space pressure induces a significant write aging effect, but it is an order of magnitude smaller than read aging. The two outlier points were ext4 full-disk aging on an HDD and BTRFS write aging on an SSD. It might be worth investigating the design decisions that make these filesystems vulnerable to this workload on a full disk.

Git-Benchmark full-disk read aging. We also use git as a more representative application benchmark. We modify the git aging benchmark [8], so that it can be used to keep a disk in a nearly-full steady state. The git benchmark replays the commit history of the Linux kernel from `github.com`. The benchmark pulls each commit, running a grep test every 100 commits.

The challenge to performing the git test on a full disk is that the repository grows over time. The disk starts empty



(a) Grep performance on HDD under git benchmark. ext4 on a full disk is ˜20% slower than on empty. For XFS and BTRFS, the full and empty performance is barely distinguishable.



(b) Grep performance on SSD under git benchmark. The performance between full and empty for all filesystems is negligible.

Figure 3: Read performance under the git benchmark. Lower is better.

and eventually becomes full, at which time we cannot pull newer commits. We overcome this challenge by maintaining multiple copies of the repository. We initially fill the disk to 75% by creating multiple copies of the initial commit. Then we update the repositories round-robin by pulling one more commit, until a pull fails due to disk fullness. At that point the repository to which the pull failed is deleted, freeing up space. Then the process continues.

Every operation is also mirrored on an "empty" filesystem and an "unaged" version (see section 4). Because this workload is generally CPU-bound during the pulls, we do not present the effect on write aging.

On an HDD, there is a big difference between the empty and unaged versions (Figure 3a), commensurate with prior results [8]. For XFS and BTRFS, the full and empty versions are barely distinguishable. The read cost for ext4 on a full disk is about 20% greater than on an empty disk.

On SSD, the full and empty lines of all three filesystems

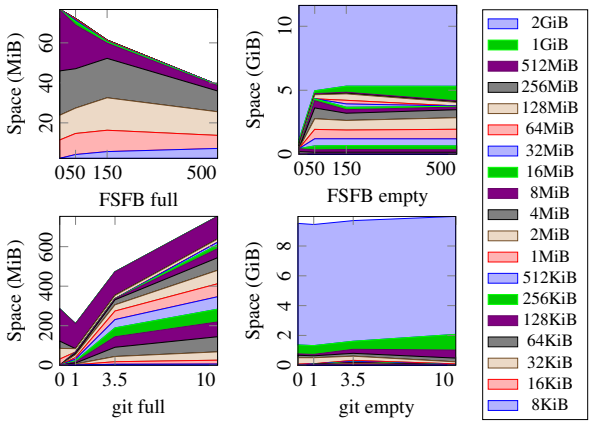


Figure 4: Free space by extent size on ext4 for snapshots under FSFB (at 0, 50, 150 and 500 rounds) and git (at 0, 1, 3.5 and 10 thousand pulls). Each bar represents the total free space in extents of the given size.

are essentially indistinguishable, shown in Figure 3b. On ext4, F2FS and, to a lesser extent on BTRFS, the read costs of the unaged versions drift higher as the benchmark progresses. This is due to a smaller average file size.

If free-space aging were a first-order consideration, we would expect it to consistently create performance degradation in all of these experiments. In the git workload, disk fullness has at most a lower-order effect on read aging than the workload itself. Its biggest impact was on ext4 on HDD, which added 20% to the read cost, compared to a 1,200% increase from the baseline fragmentation caused by usage with an abunance of space.

Free-Space Fragmentation on ext4 figure 4 shows the distribution of free-space among different extent sizes (bucketed into powers of 2), as reported by e2freefrag [2], on ext4 during our benchmarks.

Both benchmarks create many small free fragments. However, FSFB on a full disk immediately uses all the large free extents, whereas git on a full disk and both benchmarks on a empty disk have large free extents available throughout. Because ext4 saw a large performance impact from fullness under FSFB (figure 1), but not under git (figure 3), this suggests that the availability of large free extents is more important for ext4 performance than the existence of many small free fragments.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Ethan Miller, for their helpful feedback on the paper. We gratefully acknowledge support from a NetApp Faculty Fellowship, NSF grants CCF 805476, CCF 822388, CNS 1408695, CNS 1755615, CCF 1439084, CCF 1725543, CSR 1763680, CCF 1716252, CCF 1617618, IIS 1247726, and from VMware.

6 Discussion Topics

Our position is that, although the community acknowledges that filesystem aging degrades performance, the causes and effects of filesystem aging are poorly understood. As a result both filesystem designers and filesystem users do not know how to prevent, treat or even work around this problem.

We would like to engage the storage community about their experience with filesystem aging and full-disk fragmentation.

- Are there additional experiments that the community would find compelling to tease out and isolate the root causes of aging?
- Are there realistic application workloads—beyond `git`—that would generate aged filesystem states? Specifically, are there application workloads that generate significant free space fragmentation that have been observed in the wild?
- What techniques can be/are being used to combat the different causes of filesystem aging?

This paper seeks to generate discussion by challenging the commonly held belief that disk fullness has a first-order performance impact on filesystem performance. If workshop attendees have lived experiences that run counter to our findings, we hope to discuss the conditions that led to the filesystem states that they observed and diagnose the root cause of their filesystem's performance degradation.

To the extent that disk fullness presents challenges to filesystem design, we would love to discuss the theoretical and systems-design approaches which could solve the problem.

References

[1] blktrace(8) - linux man page. https://linux.die.net/man/8/blktrace.

[2] e2freefrag(8) - linux man page. https://linux.die.net/man/8/e2freefrag.

[3] Wikipedia: File system fragmentation. https://en.wikipedia.org/wiki/File_system_fragmentation. Accessed: 2019-03-13.

[4] *31st ACM Symposium on Parallelism in Algorithms and Architectures, SPAA, 2019, Phoenix, AZ, USA, June 22-24, 2019*. ACM, 2019.

[5] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic *impressions* for file-system benchmarking. *TOS*, 5(4):16:1–16:30, 2009.

[6] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In Margo I. Seltzer and Richard Wheeler, editors, *USENIX FAST*, pages 125–138. USENIX, 2009.

[7] Michael A. Bender, Martin Farach-Colton, Sándor P. Fekete, Jeremy T. Fineman, and Seth Gilbert. Cost-oblivious storage reallocation. *ACM Trans. Algorithms*, 13(3):38:1–38:20, 2017.

[8] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File systems fated for senescence? nonsense, says science! In Geoff Kuenning and Carl A. Waldspurger, editors, *USENIX FAST*, pages 45–58. USENIX Association, 2017.

[9] Robert Kelley Cook. Design goals and implementation of the new high performance file system. http://cd.textfiles.com/megademo2/INFO/OS2_HPFS.TXT.

[10] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: A right-optimized write-optimized file system. In Jiri Schindler and Erez Zadok, editors, *USENIX FAST*, pages 301–315. USENIX Association, 2015.

[11] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. An empirical study of file-system fragmentation in mobile storage systems. In Nitin Agrawal and Sam H. Noh, editors, *HotStorage*. USENIX Association, 2016.

[12] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems. In Haryadi S. Gunawi and Benjamin Reed, editors, *USENIX ATC*, pages 691–704. USENIX Association, 2018.

[13] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.

[14] Han-Kwang Nienhuys. Flash drive fragmentation: does it affect performance? http://www.lagom.nl/misc/flash_fragmentation.html.

[15] Keith A. Smith and Margo I. Seltzer. File system aging - increasing the relevance of file system benchmarks. In John Zahorjan, Albert G. Greenberg, and Scott T. Leutenegger, editors, *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Seattle, Washington, USA, June 15-18, 1997*, pages 203–213. ACM, 1997.

[16] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Optimizing every operation in a write-optimized file system. In Angela Demke Brown and Florentina I. Popovici, editors, *USENIX FAST*, pages 1–14. USENIX Association, 2016.

[17] Ningning Zhu, Jiawu Chen, and Tzi-cker Chiueh. TBBT: scalable and accurate trace replay for file server evaluation. In Garth Gibson, editor, *Proceedings of the FAST '05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA*. USENIX, 2005.