File Systems Fated for Senescence? Nonsense, Says Science!

Abstract

Many file system implementors treat file system aging as an essentially solved problem. Traditional file systems employ heuristics to avoid aging, such as colocating related files and data blocks. We describe realistic workloads that can cause these heuristics to fail, inducing large performance declines due to aging. For example, on ext4 and zfs, a few hundred `git pull` operations can reduce read performance by a factor of 2, as compared to a defragmented copy of the same file system, and performing a thousand pulls can reduce performance by up to 30×. We present experimental results showing that age-related performance degradation occurs on rotating disks and SSDs, and across multiple realistic workloads. For example, aging in our mailserver workload reduced performance by up to 30× on rotating disks.

We present microbenchmarks demonstrating that the placement strategies employed by traditional file systems are extremely sensitive to file creation order. We show that when files are created in a random order, this can create a slowdown by a factor of up to 8 for HDDs and 2 for SSDs. The creation of a few thousand small files in a directory structure can cause a 15× slowdown for reads. In short, many file systems are exquisitely prone to read aging for a variety of write workloads.

We show, however, the aging is not inevitable. BetrFS, a file system based on write-optimized data structures, exhibited almost no aging in our experiments. In fact, BetrFS typically outperformed the other file systems in our benchmarks, even after those file systems had been defragmented. Finally, we present a framework for understanding and predicting aging, and identify the key features of BetrFS that enable it to avoid aging.

1 Introduction

File systems naturally tend to become fragmented, or ***age***, as files are created, deleted, moved, appended to, and truncated [16, 23].

Fragmentation occurs when logically contiguous file blocks become scattered on disk and files that are accessed together become stored in disparate disk locations. As a result, reading these files requires additional seeks, which harms performance. For example, on a disk with 100 MB/s bandwidth and 5 ms seek time, if there are only, say, 200 seeks interspersed in 100 MB of data, reading the data will take twice as long as reading it in an ideal layout. Even on SSDs, which do not perform mechanical seeks, a decline in logical block locality can harm performance [17].

The current state of the art in mitigating the effects of aging is, primarily, to apply best-effort heuristics to avoid fragmentation at allocation time. For example, file systems attempt to place related files close together on disk, while also leaving empty space for future files [8, 15, 16, 26]. Some file systems (including ext4, XFS, and btrfs among those tested here) also include defragmentation tools that attempt to reorganize files and file blocks into contiguous regions to counteract aging.

Over the past two decades, there have been differing options about the significance of aging. The seminal work of Smith and Seltzer [23] showed that file systems age under realistic workloads, and this affects performance. They also proposed methods, based on traces, for artificially aging a file system.

On the other hand, there is a widely held view in the developer community that aging is a solved problem in production file systems. For example, the Linux System Administrator's Guide [27] says:

> Modern Linux file system keep fragmentation at a minimum by keeping all blocks in a file close together, even if they can't be stored in consecutive sectors. Some file systems, like ext3, effectively allocate the free block that is nearest to other blocks in a file. Therefore it is not necessary to worry about fragmentation in a Linux system.

Similarly, the Oracle Linux Administrator's Solutions Guide [18] states, "As XFS is an extent-based file system, it is usually unnecessary to defragment a whole file system, and doing so is not recommended."

Over the past two decades, there have also been changes in storage technology and file system design that could substantially affect aging. For example, a back-of-the-envelope analysis suggests that aging should get worse as rotating disks get bigger. Seek times have been relatively stable, but bandwidth grows (approximately) as the square root of the capacity. Suppose that the disk in the example above has only 10MB/s bandwidth but with 5ms seek time. Then the 200 seeks introduce only a 10% slowdown rather than a two-fold slowdown. Thus, we expect fragmentation to become an increasingly significant problem, as the gap between random I/O and sequential I/O grows.

On the other hand, there is widespread belief that fragmentation is not an issue on SSDs. For example, PCWorld measured the performance gains from defragmenting an NTFS file system on SSD [1], and concluded that, "From my limited tests, I'm firmly convinced that the tiny difference that even the best SSD defragger makes is not worth reducing the life span of your SSD."

In this paper, we revisit the problem of file-system aging, in light of changes in storage hardware, file-system design, and data-structure theory. We make several contributions: (1) We give a simple, fast, and portable method for aging file systems. (2) We show that fragmentation over time, that is, aging, is a first-order performance concern, and this is true even on modern hardware, such as SSDs, and on modern file systems. (3) Furthermore, we show that aging is not inevitable. We present several techniques for avoiding aging. We show that BetrFS, a research prototype that includes several of these design techniques, is much more resistant to aging than the other file systems we tested. In fact, BetrFS essentially did not age in our experiments, showing that aging is a solvable problem.

Results. We benchmark four widely used file systems—btrfs [20], ext4 [8, 15, 26], XFS [24], and zfs [7]—and one research file system—BetrFS—by aging them using realistic application workloads. One workload ages the file system by performing successive git checkouts of the Linux kernel source, emulating the aging that a developer might experience on his or her workstation. A second workload ages the file system by running a mail server benchmark, emulating aging that a server might experience over continued use.

We evaluate the impact of aging as follows. We periodically stop the aging workload and measure the overall read throughput of the file system—greater fragmentation will result in slower read throughput. To isolate the impact of aging, as opposed to performance changes due to changes in, say the distribution of file sizes, we then copy the file system onto a fresh partition, essentially producing a defragmented or "clean" version of the file system, and perform the same read throughput measurement. Any difference in read throughput between the aged and clean copies is the result of aging.

We find that

- All the production file systems aged on both rotating disks and SSDs. For example, under our git workload, we observe over 50× slowdowns on hard disks and 2–5× slowdowns on SSD. Similarly, our mail server slows $4--30\times$ on a HDD due to aging.
- Aging can happen quickly. For example, ext4 shows over a 2× slowdown after 100 git pulls, and btrfs and zfs slow down similarly after 300 pulls.
- BetrFS exhibited essentially no aging. In almost all cases, BetrFS's aged performance was better than the other file systems' clean performance. For instance, on our mail server workload, aged ext4 is 28× slower than an aged BetrFS. We note that, although BetrFS has been described in previous work [11, 28], to our knowledge, this is the first analysis of the design properties of this system that counteract aging.
- The costs of aging can be staggering in concrete terms. For example, at the end of our git workload on an HDD, all four production file systems took over 8 minutes to grep through 1 GB of data. Two of the four took over 15 minutes. BetrFS took 10 seconds.

We performed several microbenchmarks to dive into the causes of aging, and found that performance in the production file systems was sensitive to numerous factors:

- If files are created out of order relative to the directory structure (and therefore relative to a depth-first search of the directory tree) ext4 and zfs cannot achieve a throughput of 1 MB/s scanning only 2000 files, and the other standard file systems surveyed fall below 2 MB/s at 3000 files. This need not be the case; BetrFS maintains throughput of $25 - 60$ MB/s.
- If an application writes to a file in small chunks, then the file's blocks can end up scattered on disk, harming performance when reading the file back. For example, in a benchmark in which we appended 16KB chunks to 10 files in a round-robin fashion, aging reduced read throughput in btrfs, ext4, and XFS on a hard drive by a factor of 15–25×. zfs throughput aged by about 2×. BetrFS throughput was, on average, roughly the same in the clean and aged file systems.
- Some file systems slowed down during the course of our git benchmark *even after defragmentation*. For example, read throughput in a clean btrfs, ext4, or zfs partition was about 50% slower at the end of the benchmark than at the beginning. BetrFS and XFS do not exhibit this trend. This shows that file system heuristics to avoid defragmentation can fail, even in the ideal case of a fresh copy onto an empty partition.

2 Related Work

Prior work on file-system aging falls into three broad categories: techniques for artificially inducing aging, techniques for measuring aging, and techniques for mitigating aging.

2.1 Creating Aged File Systems

The seminal work of Smith and Seltzer [23] created a methodology for simulating and measuring aging on a file system—leading to more representative benchmark results than running on a new, empty file system. The study is based on data collected from daily snapshots of over fifty real file systems from five servers over durations ranging from one to three years. An overarching goal of Smith and Selter's work was to evaluate file systems with representative levels of aging.

A number of other tools have been subsequently developed for synthetically aging a file system. In order to realistically measure NFS performance, TBBT [29] was designed to synthetically age a disk image to create a realistic initial state for an NFS trace replay. TBBT first creates a namespace hierarchy, then interleaves synthetic operations so that allocations are more fragmented.

The Impressions framework [2] was designed so that users can synthetically age a file system by setting a small number of parameters, such as the organization of the directory hierarchy. Impressions also lets users select a target layout score for the resulting image.

Both TBBT and Impression focus on creating an aged file system with a specific level of fragmentation, while our study explores realistic workloads to see if they lead to fragmentation.

2.2 Measuring Aged File Systems

Smith and Seltzer also introduced a **layout score** for studying aging, which was used by subsequent studies [2, 4]. Their layout score is the fraction of file blocks that are placed in consecutive physical locations on the disk. In this paper, we use a *dynamic* version of this layout score that performs a recursive scan through the file system and tracks what fraction of blocks are requested from the driver in sequential order. Thus this dynamic layout score both incorporates metadata traversal and allows for the OS to perform some collalescing of requests.

The degree of fragmentation (DoF) is used in the study of fragmentation in mobile devices [12]. DoF is the ratio of the actual number of extents to the ideal number of extents. As with the layout score, DoF measures how one file is fragmented.

Several studies have reported on file-system statistics such as numbers of files, distributions of file sizes and types, and organization of file system namespaces [3, 10, 21], and these statistics can be used to reason about or age a file system [2, 29]. For example, as we describe later on, file-size distribution can have an impact on fragmentation.

2.3 Existing Techniques to Mitigate Aging

Most modern file systems employ heuristics designed to avoid free-space fragmentation and to place logically related objects in close proximity—and thus avoid data fragmentation. We briefly outline some well known approaches below.

Allocation heuristics. When files are created or extended, blocks must be allocated to store the new data. Especially for update-in-place file systems, such as ext4, where data is rarely, if ever relocated, initial block allocation decisions are essential to performance over the life of the file system. Most file systems do not move data specifically to counteract fragmentation.

FFS [16] and the ext family divide the logical block address space into cylinder groups and attempt to locate file objects from the same directory in the same cylinder group, since these files are often accessed together. They also try to place a file's data blocks in the same cylinder group and to lay them out contiguously in order to minimize the number and distance of seeks.

Most modern file systems, including ext4, XFS, btrfs, and zfs, implement delayed allocation. Delayed allocation delays the physical allocation of data until buffers are written to disk, thus reducing fragmentation.

Additionally, application developers can request persistent preallocation through `fallocate`. However, developers need to know how much space they will use in order to exploit this interface to reduce fragmentation.

Efficient metadata. Extents are often a compact way to index file data. Compared to file systems that use direct and indirect pointers, extent-based file systems [15, 24] can often locate data by reading fewer metadata blocks, reducing the number of disk seeks required to read a file.

Defragmentation. Many of the file systems in this study provide online or offline defragmentation utilities [5, 18, 25]. Defragmenters can be used to gather each file's blocks and group related data and metadata. This type of defragmentation may include two effects: increasing the proximity of logical objects and creating contiguous regions of free space. After such defragmentation, reads and allocations can become more efficient.

Log-structured file systems [22] garbage collect data to free up space and to defragment free space. However, this may harm read performance because related blocks can be moved farther from each other. A recently proposed defragmentation scheme for log-structure file systems [19] reorders blocks in inode order before writing back to disk which improves matters but does not resolve all types of fragmentation.

Copying into a fresh partition. An in-order copy of a file system directory tree onto a newly formatted disk partition should create an optimal disk layout. However, a file systems may also choose to leave space or lay out files in some order for perceived efficiency or convenience.

For example, ext4 places the contents of a directory in hash rather than logical order. Moreover, it attempts to leave some space between files and in directories for future file creation and extension. We will show that under certain conditions these heuristics have the perverse effect of greatly reducing read performance on in-order copied data.

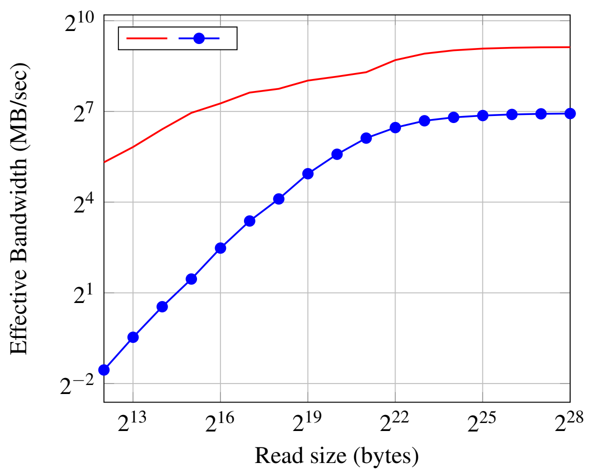


Figure 1: Effective bandwidth vs. read size (higher is better). Even on SSDs, large I/Os can yield an order of magnitude more bandwidth than small I/Os.

3 A Framework for Aging

This section provides a collection of design principles that can provide strong guarantees against aging.

Our model of aging is based on the observation that better logical block layout leads to better read performance. This is generally true for hard drives, where large discontinuities in the LBA space lead to seek overheads. SSDs can also realize higher read performance when requesting contiguous logical blocks [13].

Seek times offer a simple explanation for the gap between sequential and random reads on disks. On SSDs, this gap are hard to explain conclusively without vendor support, but common theories include: sequential accesses are easier to stripe across internal banks, better leveraging parallelism [13]; some FTL translation data structure have nonuniform search times [14]; and fragmented SSDs are not able to prefetch data [9] or metadata [12]. For whatever reason, SSDs show a gap between sequential and random reads, though not as great as on disks.

Define the *natural transfer size* to be the amount of sequential data that must be read per seek in order to obtain some fixed fraction of maximum throughput, say 50% or 90%. Figure 1 shows measurements of SSD and HDD bandwidth as a function of read size. On a hard drive, a reasonable natural transfer size would be 4 MB.

The major file systems currently in use can be roughly categorized as B-tree-based, such as XFS, zfs, and btrfs, and update-in-place, such as ext4. The research file system that we consider, BetrFS, is based on B$^{\varepsilon}$-trees. Each of these fundamental designs creates different aging considerations, discussed in turn below. In later sections, we present experimental validation for the design principles presented in this section.

B-trees. The aging profile of a B-tree depends on the leaf size. If the leaves are much smaller than the natural transfer size, then the B-tree will age as the leaves are split and merged, and thus moved around on the storage device.

Making leaves as large as the natural transfer size can affect the *write amplification* of a B-tree, which is the ratio between the changes to a B-tree and the amount written to storage. In the extreme case, a single-bit change to a B-tree leaf can cause the entire leaf to be rewritten. Thus, B-trees are usually implemented with small leaves. Consequently we expect them to age until a wide variety of workloads.

In Section 7.1, we show that the aging of btrfs is inversely related to the size of the leaves, as predicted. There are, in theory, ways to mitigate the aging due to B-tree leaf movements. For example, the leaves could be stored in a so-called packed memory array [6]. However, such an arrangement might well incur an unacceptable performance overhead to keep the leaves arranged in logical order.

Write-Once or Update-in-Place Filesystems. Both the above techniques for using large blocks without write-amplification generally involve writing each piece of data multiple times. In a write-optimized data structure, data must percolate down the tree being written into each subsequent block. Likewise, a PMA constantly shifts around data in order to maintain both sequentiality and gaps. In a data structure where data is written and never again moved, such as an update-in-place file system like ext4, sequential order is very difficult to maintain. In the worst case, this is easy to see: imagine two files written to disk, and then create a bunch of files which logically should occur between them. Without moving one or the other, the data cannot be maintained sequentially. This process is extremely brittle; if we imagine the current state of a write-once file system as a sort of sieve with holes of free space, and start putting files into those holes without altering the sieve, there is an obvious tension between the size and number of the holes—the current sequentiality of the file system—, and the availability of space to put new files while maintaining order—the potential future sequentiality.

A heuristic which attempts to overcome this brittleness is to batch writes and updates before committing them to the overall structure, similar to how we suggest that file systems batch small writes into large blocks above. This is implemented via the zfs intent log (ZIL) in zfs for example. Of course this creates an unsorted area on disk which will not be sequential. Moreover, it still does not guarantee that the structure when it is committed will be much improved.

Write-Optimization. In order for small writes and updates to be performed without writing entire blocks, these operations can be batched together into larger writes. This issue then becomes how to make sure these writes maintain some sort of overarching logical order to ensure efficient searches of the written data. Here a write-optimized dictionary, such as a B^{ε}-tree or LSM-tree, provides a theoretical solution.

4 Measuring File System Fragmentation

This section explains the two measures for file system fragmentation used in our evaluation: recursive scan latency and dynamic layout score, a modified form of Smith and Seltzer's layout score [23]. These measures are designed to capture both intra-file fragmentation and inter-file fragmentation.

Recursive scan latency. One measure we present in the following sections is the wall-clock time required to perform a recursive `grep` in the root directory of the filesystem. This workload captures the effects of both inter- and intra-file locality, as it searches both large files and large directories containing many small files. We report search time per unit of data, normalizing for how the filesystem reports this figure—we always use ext4's `du` output. Through the rest of the paper, we refer to this as the `grep` test.

Dynamic layout score. Smith and Seltzer's layout score [23] measures the fraction of blocks in a file or (in aggregate) a filesystem that are allocated in a contiguous sequence in the logical block space. We apply the core idea of Seltzer and Smith's score to the dynamic I/O patterns of a file system. During a given workload, we capture the logical block requests made by the file system, using `blktrace`, and measure the fraction that are contiguous. This approach captures the impact of placement decisions on a file system's access patterns, including the impact of metadata accesses or accesses that span files. A high dynamic layout score indicates good data and metadata locality, and an efficient on-disk organization for a given workload.

A potential shortcoming of this measure is that it does not distinguish between small and large discontiguities. Thus, we also sometimes present a histogram of the sizes of the measured discontiguities. For example, small discontiguities on a hard drive should induce less expensive mechanical seeks than large discontiguities, which require longer seeks.

5 Experimental Setup

To understand the ways that file systems' read performance can change over time, we exercise five local file systems using a series of microbenchmarks and application workloads. After each experiment, we measure both the recursive scan latency and the dynamic layout score.

Our evaluation is organized around the following questions:

- How sensitive is file system locality to the order that files are created?
- How sensitive is file system locality to write patterns within existing file system objects?
- Under realistic application workloads, how does file system locality change over time?

Each experiment compares several file systems: BetrFS, btrfs, ext4, XFS, and zfs. We use the versions of XFS, btrfs, ext4 that are part of the 3.11.10 kernel, and zfs 0.6.5-234_ge0ab3ab, downloaded from the zfsonlinux repository on `www.github.com`. We downloaded BetrFS 0.2 from the BetrFS git repository [1]. We applied several patches provided to us by the BetrFS authors in response to bugs uncovered during our benchmarking. We use default recommended file system settings unless otherwise noted. Lazy inode table and journal initialization were turned off on ext4, pushing more work onto file system creation time and reducing experimental noise.

All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, a 500 GB, 7200 RPM ATA Seagate Barracuda ST500DM002 disk with a 4096 B block size, and a 240 GB Sandisk Extreme Pro—both disks used SATA 3.0. Each file system's block size is set to 4096 B.

The system runs 64-bit Ubuntu 13.10 server with Linux kernel version 3.11.10 on a bootable USB stick. All HDD tests were performed on two partitions located at the outermost region of the drive. On SSDs, we created three partitions: two partitions were formatted and used for tests, and one partition was created to artificially tie up blocks in the FTL by writing pseudo-random data to it. We found that, even for long-running tests, no SSD garbage collection occurred without filling this third partition. We use partitions of size 4GB and 20GB. For the Git Read-Aging workload we use both 4GB and 20GB partition sizes. The 4GB partition size results were not included due to space constraints. There is essentially no difference between the two. For the Microbenchmarks, the Btrfs Nodesize and the Mailserver Workload we use a 4GB partition size. Unless otherwise noted, all benchmarks are cold-cache tests.

6 Fragmentation Microbenchmarks

We present several simple microbechmarks, each designed around a write/update pattern for which it is difficult to ensure both fast writes in the moment and future locality. These microbenchmarks isolate and highlight the effects of intra-file fragmentation, inter-file fragmentation and sensitivity to creation order. These microbenchmarks show the performance impact aging can have on read performance in the worst cases.

[1] `github.com/oscarlab/betrfs`

Intrafile Fragmentation. When a file grows in size, there may not be room to store the new blocks immediately after the old blocks on disk, and a single file's data may become scattered. Modern file systems apply several heuristics to avoid intrafile fragmentation (Section 2.3).

Our benchmark first creates one hundred files in the root directory of an empty file system. We then iteratively take the following steps to incrementally age the file system:

1. Append 4 KiB of random data to each file in a round-robin fasion.
2. Flush all pending writes to disk and clear the caches, via unmounting and remounting the file system.
3. Do a cold cache `grep` of all files, measuring wall-clock time and block access locality.

We call the measurements as this test runs the "aged" performance of the file system.

To create an "unaged" file system, we copied all files to a newly initialized file system at the end of each round using `cp -a`. We again unmounted and remounted the file system to measure a cold-cache `grep`. The results of this test are shown in Figures 2a and 2c, for the HDD and SSD, respectively.

For unaged file systems on hard disk (Figure 2a), all the file systems except zfs are able to perform greps at a cost of about 10 s/GB, which corresponds to about 100 MB/s. The maximum throughput of the drive is about 124 MB/s according to `hdparm -t`.

In the aged setup, all the file systems except zfs and BetrFS show a clear trend of deteriorating performance. Although zfs is slower once it ages, the degradation plateaus much earlier than other file systems and stays flat. One likely explanation for ZFS's relatively flat aging is that the default data block size is 128 KB—much closer to the natural transfer size of the device. BetrFS is the only file system that keeps costs below 50 s/GB after 10 rounds of aging and, other than BetrFS and zfs, all take at least 100 s/GB after 50 rounds. With the exception of BetrFS and zfs, the slowdown from aging is roughly 10x.

These performance numbers are inversely correlated with the dynamic layout scores. Recall that this metric gives the fraction of blocks which are requested in logically sequential order—thus 1 means the blocks are requested perfectly sequentially, and a score such as .8 means that on average, there is 1 block requested non-sequentially out of every 5. All unaged file systems, as well as BetrFS have scores close to 1. zfs clean has a slightly lower layout score, which corresponds to it's slightly worse performance.. The file systems that exhibit poor aging have scores very close to 0 in the aged setting. zfs maintains a score around 0.96, likely because of it's larger block size (Note that $31/32 \approx 0.96$, and there are 32 4KB blocks in an 128KB block). btrfs maintains a layout scoore of 0.998.

On SSD, the performance curves are quite different (Figure 2c). These curves appear to be dominated by metadata lookup costs when files are small, and, as files grow, data read costs dominate. In some cases, such as XFS and btrfs, the costs of the clean file systems are lower than the aged variants. Strangely, on an SSD, ext4 and zfs aged versions are marginally better than the clean versions, but within a small margin. We note that these curves flatten out, whereas the HDD curves continue degrading; we suspect the difference is that HDDs are more sensitive to discontiguity size, whereas SSDs are sensitive primarily to whether the I/Os are contiguous in LBA space.

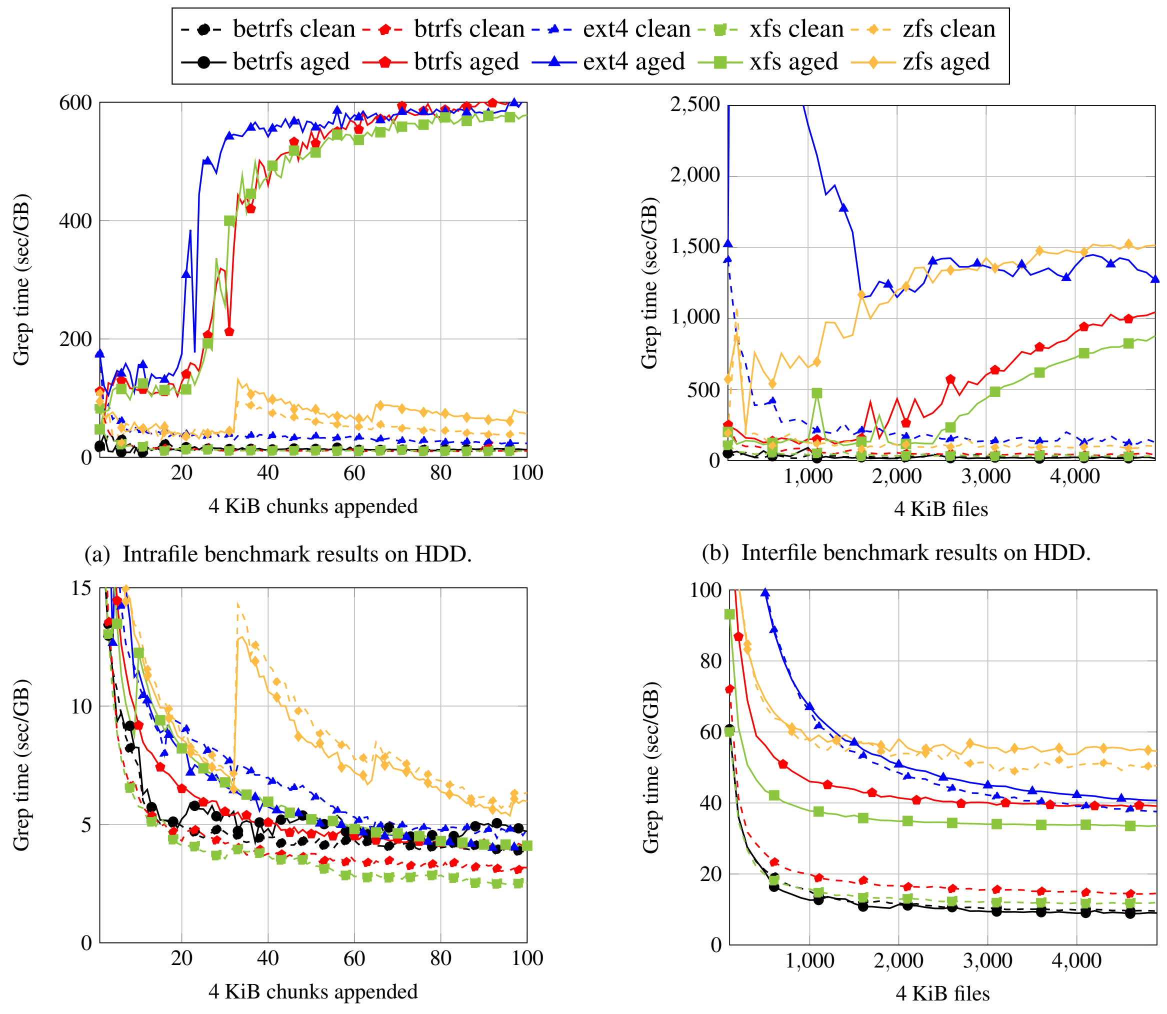
In summary, this microbenchmark is enough to significantly age a file system on an HDD, however the results are less clear on an SSD, although some file systems will still age. While not pictured, we also mention, that for a HDD, significant aging can be induced with only 10 files, and 16KB chunks.

Interfile Fragmentation. Many workloads read multiple files with some logical relationship, and frequently those files are placed in the same directory. Interfile fragmentation occurs when files which are related—in this case being close together in the directory tree—are not colocated in the LBA space.

Although many file systems attempt to group related objects (i.e., files in the same directory or directory subtree) in close physical proximity, these heuristics do not always succeed, and thematically it can be extremely difficult to maintain. For example, FFS-style file systems place files within the same directory in the same cylinder group. However, once space in a cylinder group is exhausted, the file system is forced to store new files elsewhere. Similarly, if a file is created in one directory and moved to another, the file is not physically moved into a different cylinder group.

We present a microbenchmark to measure the impact of namespace creation order on interfile locality. The microbenchmark starts by creating one thousand directories in a newly formatted file system. During this phase, each new directory is created under a randomly selected parent. The microbenchmark next populates this directory hierarchy with 4 KiB files, randomly choosing a parent directory for each file. We note that this experiment was run on the same machines as the other experiments, but with the root partition on the HDD, rather than a USB drive. Based on the (incomplete) results at the time of submission, booting from USB yields similar data.

After every 100 newly created files, we measure file system's read performance using a cold-cache `grep`, and we calculate its dynamic layout score. We then repeat



(a) Intrafile benchmark results on HDD.

(b) Interfile benchmark results on HDD.

(c) Intrafile benchmark results on SSD.

(d) Interfile benchmark results on SSD.

Figure 2: Intra- and interfile benchmark results.

these measures on an "unaged" version of the file system, by copying the contents to a fresh partition using *cp -a*. The results are summarized in Figures 2b and 2d.

On hard disks, all unaged file systems sustain high throughputs, or, visually, low grep costs. With the exception of BetrFS, however, the grep costs per GB increase dramatically as the file systems are aged.

The rate of aging varies by file system, which we believe is due to differences in the allocation strategies employed by each file system. ext4 ages precipitously at first, and then levels off; this is commensurate with the heuristic of spreading small directories across disparate cylinder groups to accommodate future growth. As more data is added to these directories, the costs of shifting cylinder groups is amortized. In fact, the layout score of ext4 aged *improves* slightly until the performance levels off, albeit at a very poor level. More generally, there is a dramatic difference in the layout scores of all file systems, except BetrFS, between the aged and unaged versions, and in particular, all have aged scores well below 0.1. The most likely explanation for the difference is whether the file system employs delayed allocation for data, and the degree to which this choice is factored into placement decisions of directories. We note that delayed allocation here does not represent a solution to the problem, as the "clean" lines have unrealistically good visibility into future write patterns; rather, the "aged" lines illustrate the degree to which locality can be lost when placement decisions have to be made with incomplete information. As on our other microbenchmarks, BetrFS shows essentially no signs of aging, sustaining consistently high performance and dynamic layout scores.

As in the intra-file fragmentation experiment, performance on SSD improves as the experiment progresses, as directory search costs are amortized on larger files. In this experiment, reading small files is roughly 10×

slower than reading fewer, large files, as the underlying I/O requests are smaller and less efficient at the device level. Performance improves because, as the number of files increases, the overhead of traversing the directory hierarchy can be amortized over more data. We note that no aged file system other that BetrFS achieves a lower cost than 25sec/GB. Some of the unaged file systems perform poorly here as well.

Effect of File Creation Order. The final read-based microbenchmark we ran involves copying the Linux source code to the target filesystem, one file at a time in random order, generating directories as necessary (using `cp --parents`). We measure the `grep` time of the resulting filesystem, and compare it to one that creates identical contents in search order (using `cp -a`). The results are presented in Figure 3.

On the hard disk, this microbenchmark highlights the sensitivity of file system performance to file creation order, as every file system except BetrFS shows a 3-15x slowdown when the files are copied in a random order.

The effect largely disappears on SSD, except for btrfs and XFS. The fastest file system for this benchmark is btrfs in the standard order. The rest of the file systems exhibit much worse performance than unaged btrfs or XFS, regardless of how the files are created. This experiment indicates that, under the best circumstances, XFS and btrfs can efficiently organize directories and smaller files on an SSD, but most file systems have room for improvement.

To further explain the performance difference between hard disk and SSD, we plot the histograms of discontinuity sizes for btrfs and ext4 in Figure 3. As the graphs show, btrfs in standard order has a very small number of discontinuities, and hence gets good performance on both hard disk and SSD. ext4 in the standard order has more discontinuities, but they are all small, so it performs well on hard disk, although not as well as btrfs. In the random order, ext4 has substantially larger discontinuities than btrfs, but only about 50% more. These results indicate that only the number of discontinuities matter on SSDs, whereas HDD performance is more sensitive to the magnitude of discontinuities.

7 Application Level Read-Aging: Git

To measure aging in the "real-world," we create a workload designed to simulate a developer using git to work on a collaborative project.

Git is a distributed version control system that enables collaborating developers to synchronize their source code changes. Git users "pull" changes from other developers, which then get merged with their own changes. In a typical workload, a Git user may perform pulls multiple times per day over several years in a long-running project. Git can synchronize all types of file system changes, so performing a Git pull may result in the creation of new source files, deletion of old files, file renames, and file modifications. Git also maintains its own internal data structures, which it updates during pulls. Thus Git performs many of the operations similar to those shown in Section 6 to cause file system aging.

The benchmarks performs 10,000 pulls from the Linux git repository, starting from the initial commit. After every 100 pulls, it performs a recursive grep test and computes the file system's dynamic layout score. It then copies the contents of the file system to a freshly formatted partition to measure the file system's performance in an unaged state. Figure 4 shows the results of the grep tests on both hard disk and SSD.

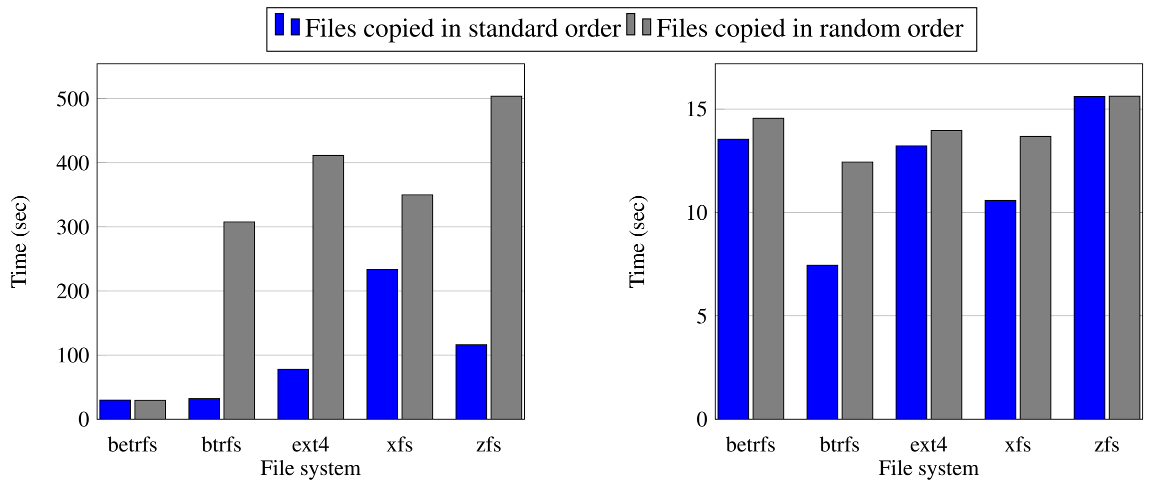
On a hard disk, we see a clear aging trend in all file systems except BetrFS (the zig-zags are the result of git garbage collection, discussed below). By the end of the experiment, all the file systems except BetrFS show performance drops under aging on the order of at least 3x, as much as 10x, and all are at least 15x worse than BetrFS.

On an SSD, btrfs and XFS show clear signs of aging, although they converge to a fully aged configuration after only about 1,000 pulls. While the slowdown is not as drastic as on hard drive, even with the active defragmentation of git garbage collection we see slowdowns of 2x-4x, and with it off at least 5x over BetrFS, which does not slow down. Perhaps most significantly due to the cost difference of the hardware, aged BetrFS on hard drive outperforms all the other aged file systems on solid state, and is close even when they are unaged.

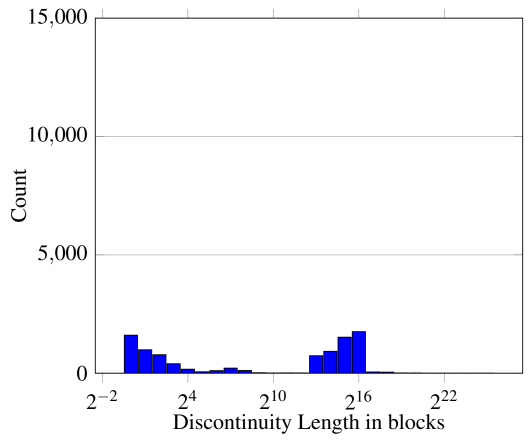
ext4 and zfs perform pooly in both the aged and unaged cases. We believe this is because the average file size decreases over the course of the test, and these file systems perform poorly even in an unaged state when the average file size is small. To test this hypothesis, we constructed synthetic workloads similar to the interfile fragmentation microbenchmark described, but varying the file size (in the microbenchmark it was uniformly 4KB). The results are shown in Figure 6; we believe that the slightly worse performance in the synthetic workloads is due to the increased complexity in the directory structure.

The zig-zag pattern in the graphs is created by an automatic garbage collection process in Git. Once a certain number of "loose objects" are created (in git terminology), many of them are collected and compressed into a "pack." At the file system level, this corresponds to merging numerous small files into a single large file. According to the Git manual, this process is designed to "reduce disk space and increase performance," so this is an example of an application-level attempt to mitigate file system aging.

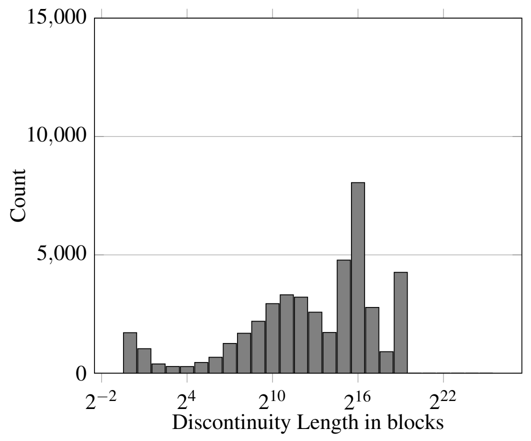
If we turn off the git garbage collection, the effect of aging is even more pronounced. On HDD btrfs, ext4 and zfs show at least a 15x performance over their unaged



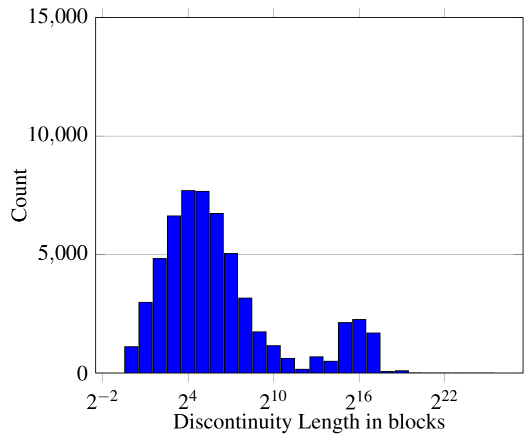
(a) HDD. (b) SSD.



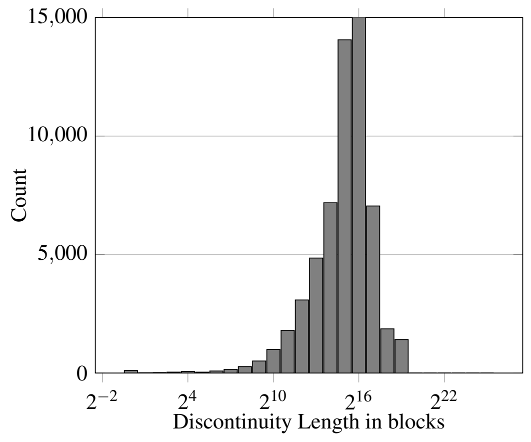
(c) Discontinuity lengths: In-order copy on btrfs



(d) Discontinuity lengths: Random-order copy on btrfs

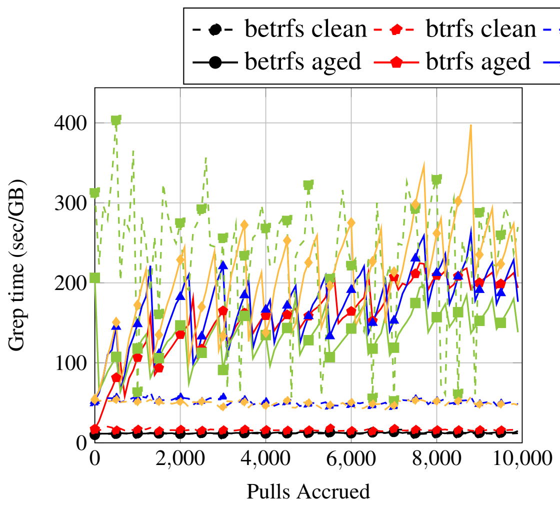


(e) Discontinuity lengths: In-order copy on ext4

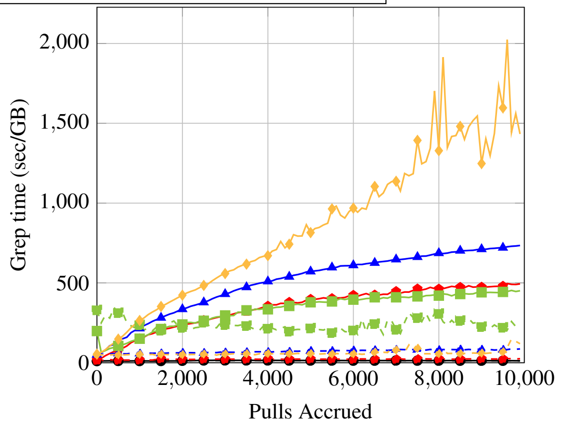


(f) Discontinuity lengths: Random-order copy on ext4

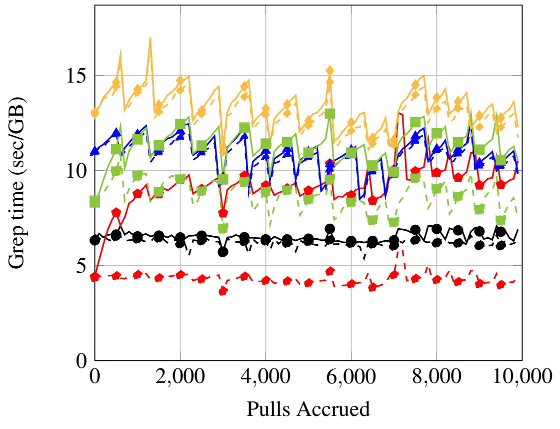
Figure 3: Time to perform a grep through the linux source code copied onto the target file system in the standard and a random order, and histograms of the discontinuity sizes for ext4 and btrfs.



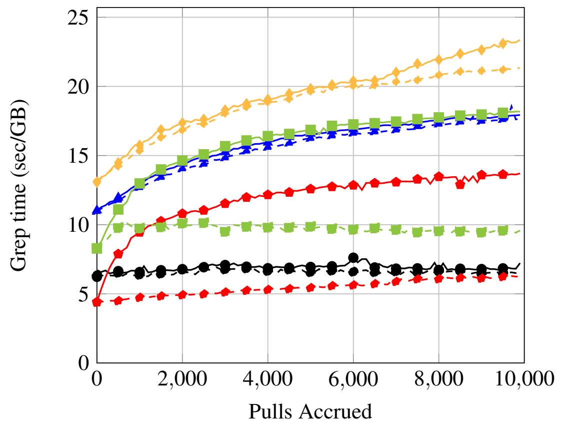
(a) Results: HDD, 20GB partition, git garbage collection on



(b) Results: HDD, 20GB partition, git garbage collection off



(c) Results: SSD, 20GB partition, git garbage collection on



(d) Results: SSD, 20GB partition, git garbage collection off

Figure 4: Git read-aging experimental results on 20GB partitions.

versions and have 50-90x worse performance than BetrFS at the end of the test. XFS performs relative poorly both in the aged and unaged versions.

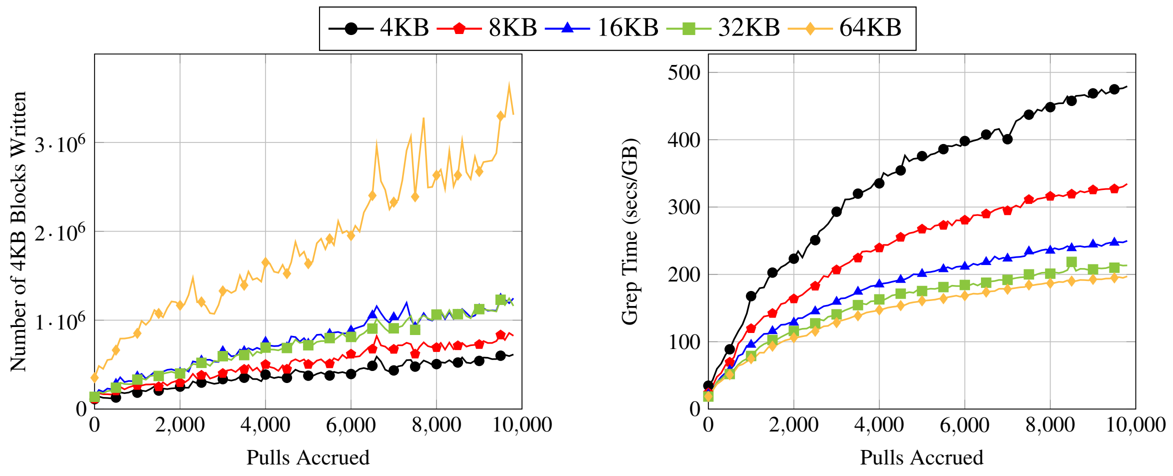
On SSD, the same patterns emerge as with garbage collection on, but exacerbated: ext4 and zfs perform poorly but in the aged and unaged states, XFS and btrfs both aged significantly, around 2x each, and BetrFS has strong level performance in both states.

We note that this analysis, both of the microbenchmarks and of the git workload, runs counter to the commonly held belief that locality is solely a hard drive issue. While the random read performance of solid state drives does somewhat mitigate the aging effects, it does still clearly have a major performance impact.

7.1 Btrfs Node-Size Trade-Off

Btrfs allows users to specify the node size of its metadata B-tree when formatting the file system. Because small files are stored in the metadata B-tree, a larger node size results in a less fragmented file system while incurs more expensive metadata updates.

We run the same test with a 4K nodesize, the default setting, as well as 8K, 16K, 32K and 64K, which is the maximum setting allowed. In Figure 5b, we show the same sort performance graphs as in Figure 4 , one for each node size. The 4K node size has the worst read performance by the end of the test, and the performance consistently improves as we increase the node size all the way to 64K. In Figure 5a, we plot the number of 4K blocks written to disk between each test (within the 100 pulls). As expected, the 64K node size writes the maximum number of blocks and the 4K node writes the least. for each nodesize during the test. We thus demonstrate—as predicted by our model—aging is reduced by larger block size, but at the cost of write-amplification.



(a) Results: HDD, 4GB partition, git garbage collection off (b) Results: HDD, 4GB partition, git garbage collection off

Figure 5: How node size affects btrfs write amplification and aging.

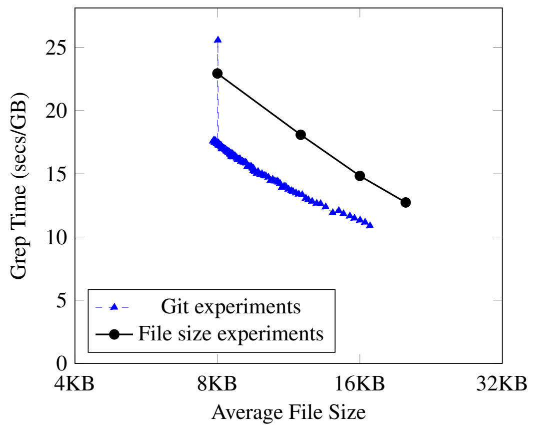


Figure 6: How average file size affects ext4 grep performance.

8 Application Level Aging: Mail Server

In addition to the git workload, we evaluate aging with the Dovecot email server. Dovecot is configured with the Maildir backend, which stores each message in a file, and each inbox in a directory. On a typical mailserver, users would receive new email, delete old emails and search through their mailbox. We simulate this behavior for 2 users, each having 80 mailboxes.

A cycle or "day" for the mailserver comprises of 8,000 operations, where each operation is equally likely to be a insert or a delete, corresponding to receiving a new email or deleting an old one. Each email is a string of random characters, the length of which is uniformly distributed over the range [1, 32K]. Each mailbox is initialized with 1,000 messages. We simulate the mailserver for 100 cycles and after each cycle we perform a recursive grep for a random string. As in our git benchmarks, we then copy the partition to a freshly formatted "clean" filesystem, and run a recursive grep there.

We run this experiment on a rotational disk with a 4GB partition size. Unlike the other experiments, we ran the root file system on a separate partition of the same disk. We were not able to collect a full data set using a USB root device by the time of submission, but spot checked results are consistent with our data.

Figure 7 shows the read costs (lower is better) in seconds per GB of the grep test on hard disks. Although the unaged versions of all file systems show consistent performance over the life of the benchmark, the aged versions ofext4, btrfs, XFS and zfs all show significant degradation over time. In particular, aged ext4 performance degrades by 4.4×, and is 28× slower than aged BetrFS. XFS slows down by a factor of 7 and btrfs by a factor of 30. zfs slows down drastically, and after running the test for 3 weeks, we were only able to get as far as 4,300 iterations. However, the aged version of BetrFS does not show any slowdown.

At the time of submission, we were not able to complete the zfs clean test, as it regularly crashed. The BetrFS clean results were also not complete at the time of submission, but were consistent with the aged version.

9 Deferred Work in BetrFS

This section describes experiments to measure the potential costs that BetrFS pays in order to avoid aging.

BetrFS avoids aging by storing data in a B$^\varepsilon$-tree with 4 MiB nodes, each stored contiguously on disk. Nodes can store multiple file blocks as well as blocks from separate files. Within a node, file blocks are stored in logical order, and their contents are stored in the same order as a recursive directory traversal.

Updates to blocks are stored as "messages" in the root

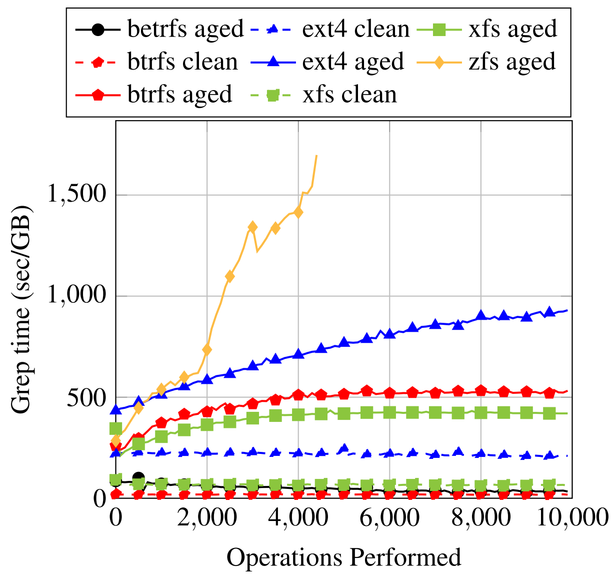


Figure 7: Mailserver read-aging experimental results.

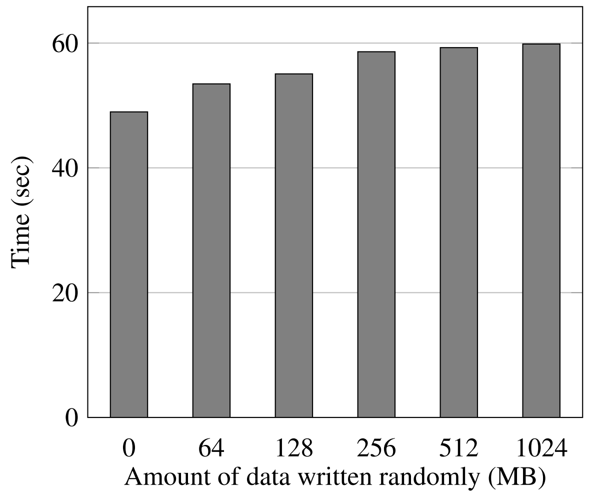


Figure 8: Grep time after performing random 4 KiB writes to a 4 GiB file.

node, and only flushed to lower nodes once enough messages have accumulated to amortize the disk seek required to move them down the tree. Thus the same message may be written to disk several times, and reads of modified file blocks must apply messages to compute the current contents of the block. This latter issue does not require any additional I/O, but can increase CPU usage.

To better understand the overhead caused by message buffering, we sequentially wrote 4GB and 16GB files and then performed a large number of random writes to each file, periodically measuring the time required to perform a grep. In one experiment, we performed random 8 million 4-byte, unaligned writes. In the second experiment, we performed 1 million random, aligned, 4KB writes. This created a very large B$^\varepsilon$-tree and filled the buffers of internal nodes with many small messages. Thus each grep has to apply all these pending messages to compute the current content of the file. All experiments were performed on hard disk. The results are shown in Figure 8.

As the experiment shows, performance drops as the number of messages in the tree increases, but levels off after the tree becomes completely full. The B$^\varepsilon$-tree used in BetrFS has a fanout of about 10, so the space available for pending messages is about 1/10th the size of the file. Small messages also have overhead, so, for example, the 4-byte writes actually consume about 50-60 bytes in the tree. In the 4GB file/4B write experiment, the internal nodes of the tree can hold about 16MB of writes, and the overhead levels off at that point. Similarly in the 4GB file/4KB and 16GB file/4KB experiments, the internal nodes can hold about 400MB and 1.6GB of writes, respectively, and again the overhead levels off there. In all cases, overhead from applying messages is less than 65%, and typically less than 20%.

10 Conclusion

The experiments above suggest that conventional wisdom on fragmentation, aging, allocation and file systems is inadequate in several ways.

First, while it may seem intuitive to write data as few times as possible, writing data only once creates a tension between the logical ordering of the filesystem's current state and the potential to make modifications without disrupting the future order. Moving (and therefore) writing data multiple times allows the filesystem to maintain locality. The overhead of these multiple writes can be managed by moving data in batches, as is done in write-optimized data structures.

For example, in BetrFS, data might be written as many as a logarithmic number of times, whereas in ext4, it will be written once, yet BetrFS in general is able to perform as well as or better than an unaged ext4 file system and significantly outperforms aged ext4 file systems.

Second, today's file system heuristics are not able to maintain enough locality to enable reads to be performed at the disks natural transfer size. And since the natural transfer size on a rotating disk is a function of the seek time and bandwidth, it will tend to increase with time. Thus we expect this problem to possibly become worse with newer hardware, not better.

Finally, we expected because of our analysis that non-write-optimized file systems would age, and indeed all our experiments showed surprisingly quick and dramatic aging. This rapid aging is important because it means that users (of at least certain workloads) likely have little or no experience of using unaged file systems and therefore they do not feel the pain of degrading performance.

References

[1] Fragging wonderful: The truth about defragging your ssd. http://www.pcworld.com/article/2047513/fragging-wonderful-the-truth-about-defragging-your-ssd.html. Accessed 25 September 2016.

[2] Agrawal, N., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Generating realistic impressions for file-system benchmarking. *ACM Transactions on Storage (TOS)5*, 4 (Dec. 2009), art. 16.

[3] Agrawal, N., Bolosky, W. J., Douceur, J. R., and Lorch, J. R. A five-year study of file-system metadata. *Trans. Storage 3*, 3 (Oct. 2007).

[4] Ahn, W. H., Kim, K., Choi, Y., and Park, D. DFS: A de-fragmented file system. In *Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)* (2002), pp. 71–80.

[5] Btrfs. Wiki section on defragmentation. https://wiki.archlinux.org/index.php/Btrfs#Defragmentation. Accessed 10 May 2016.

[6] Bender, M. A., Demaine, E., and Farach-Colton, M. Cache-oblivious B-trees. *SIAM J. Comput. 35*, 2 (2005), 341–358.

[7] Bonwick, J., and Moore, B. ZFS: The last word in file systems. In *SNIA Developers Conference* (Santa Clara, CA, USA, Sept. 2008). Slides at http://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf, talk at https://blogs.oracle.com/video/entry/zfs_the_last_word_in. Accessed 10 May 2016.

[8] Card, R., Ts'o, T., and Tweedie, S. Design and implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux* (Amsterdam, NL, Dec. 8–9 1994), pp. 1–6. http://e2fsprogs.sourceforge.net/ext2intro.html.

[9] Chen, F., Koufaty, D. A., and Zhang, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2009), SIGMETRICS '09, ACM, pp. 181–192.

[10] Downey, A. B. The structural cause of file size distributions. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2001), SIGMETRICS '01, ACM, pp. 328–329.

[11] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M., Farach-Colton, M., Johnson, R., Kuszmaul, B. C., and Porter, D. E. BetrFS: Write-optimization in a kernel file system. *ACM Transactions on Storage (TOS) 11*, 4 (Nov. 2015), art. 18.

[12] Ji, C., Chang, L.-P., Shi, L., Wu, C., Li, Q., and Xue, C. J. An empirical study of file-system fragmentation in mobile storage systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (Denver, CO, 2016), USENIX Association.

[13] Jung, M., and Kandemir, M. Revisiting widely held ssd expectations and rethinking system-level implications. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (New York, NY, USA, 2013), ACM, pp. 203–216.

[14] Ma, D., Feng, J., and Li, G. A survey of address translation technologies for flash memories. *ACM Comput. Surv. 46*, 3 (Jan. 2014), 36:1–36:39.

[15] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The new ext4 filesystem: current status and future plans. In *Ottowa Linux Symposium (OLS)* (Ottowa, ON, Canada, 2007), vol. 2, pp. 21–34.

[16] McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS) 2*, 3 (Aug. 1984), 181–197.

[17] Min, C., Kim, K., Cho, H., Lee, S., and Eom, Y. I. SFS: random write considered harmful in solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (San Jose, CA, USA, Feb. 14–17 2012), art. 12.

[18] ORACLE. Defragmenting an XFS file system. Oracle Linux Administrator's Solution Guide for Release 6. `https://docs.oracle.com/cd/E37670_01/E37355/html/ol_defrag_xfs.html`. Accessed 10 May 2016.

[19] PARK, J., KANG, D. H., AND EOM, Y. I. File defragmentation scheme for a log-structured file system. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems* (New York, NY, USA, 2016), APSys '16, ACM, pp. 19:1–19:7.

[20] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (Aug. 2013), art. 9.

[21] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A comparison of file system workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2000), ATEC '00, USENIX Association, pp. 4–4.

[22] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (Feb. 1992), 26–52.

[23] SMITH, K. A., AND SELTZER, M. File system aging — increasing the relevance of file system benchmarks. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (Seattle, WA, June 15–18 1997), pp. 203–213.

[24] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference* (San Diego, CA, USA, Jan.22–26 1996), art. 1.

[25] TS'O, T. E2fsprogs: Ext2/3/4 filesystem utilities, May 17 2015. `http://e2fsprogs.sourceforge.net/`. Version 1.42.13. Accessed 10 May 2016.

[26] TWEEDIE, S. EXT3, journaling filesystem. In *Ottowa Linux Symposium* (Ottowa, ON, Canada, July 20 2000).

[27] WIRZENIUS, L., OJA, J., STAFFORD, S., AND WEEKS, A. *Linux System Administrator's Guide*. The Linux Documentation Project, 2004. `http://www.tldp.org/LDP/sag/sag.pdf`. Version 0.9.

[28] YUAN, J., ZHAN, Y., JANNEN, W., PANDEY, P., AKSHINTALA, A., CHANDNANI, K., DEO, P., KASHEFF, Z., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Optimizing every operation in a write-optimized file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (Santa Clara, CA, USA, Feb. 22–25 2016), pp. 1–14. `https://www.usenix.org/conference/fast16/technical-sessions/presentation/yuan`.

[29] ZHU, N., CHEN, J., AND CHIUEH, T.-C. TBBT: Scalable and accurate trace replay for file server evaluation. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (Santa Clara, CA, USA, Feb. 16–19 2005), pp. 323–336.