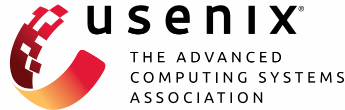


File Systems Fated for Senescence? Nonsense, Says Science!

Alex Conway and Ainesh Bakshi, *Rutgers University;* Yizheng Jiao and Yang Zhan, *The University of North Carolina at Chapel Hill;* Michael A. Bender, William Jannen, and Rob Johnson, *Stony Brook University;* Bradley C. Kuszmaul, *Oracle Corporation and Massachusetts Institute of Technology;* Donald E. Porter, *The University of North Carolina at Chapel Hill;* Jun Yuan, *Farmingdale State College of SUNY;* Martin Farach-Colton, *Rutgers University*

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

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

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

ISBN 978-1-931971-36-2

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

File Systems Fated for Senescence? Nonsense, Says Science!

Alex Conway*, Ainesh Bakshi*, Yizheng Jiao‡, Yang Zhan‡,
Michael A. Bender†, William Jannen†, Rob Johnson†, Bradley C. Kuszmaul§,
Donald E. Porter‡, Jun Yuan¶, and Martin Farach-Colton*

*Rutgers University, †Stony Brook University,
‡The University of North Carolina at Chapel Hill,
§Oracle Corporation and Massachusetts Institute of Technology,
¶Farmingdale State College of SUNY

Abstract

File systems must allocate space for files without knowing what will be added or removed in the future. Over the life of a file system, this may cause suboptimal file placement decisions which eventually lead to slower performance, or ***aging***. Traditional file systems employ heuristics, such as collocating related files and data blocks, to avoid aging, and many file system implementors treat aging as a solved problem.

However, this paper describes realistic as well as synthetic 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; performing a thousand pulls can reduce performance by up to a factor of 30. We further present microbenchmarks demonstrating that common placement strategies are extremely sensitive to file-creation order; varying the creation order of a few thousand small files in a real-world directory structure can slow down reads by $15 - 175\times$, depending on the file system.

We argue that these slowdowns are caused by poor layout. We demonstrate a correlation between read performance of a directory scan and the locality within a file system's access patterns, using a ***dynamic layout score***.

In short, many file systems are exquisitely prone to read aging for a variety of write workloads. We show, however, that aging is not inevitable. BetrFS, a file system based on write-optimized dictionaries, exhibits almost no aging in our experiments. BetrFS typically outperforms the other file systems in our benchmarks; aged BetrFS even outperforms the unaged versions of these file systems, excepting Btrfs. We present a framework for understanding and predicting aging, and identify the key features of BetrFS that avoid aging.

1 Introduction

File systems tend to become fragmented, or ***age***, as files are created, deleted, moved, appended to, and truncated [18, 23].

Fragmentation occurs when logically contiguous file blocks—either blocks from a large file or small files from the same directory—become scattered on disk. Reading these files requires additional seeks, and on hard drives, a few seeks can have an outsized effect on performance. For example, if a file system places a 100 MiB file in 200 disjoint pieces (i.e., 200 seeks) on a disk with 100 MiB/s bandwidth and 5 ms seek time, 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 [19].

The state of the art in mitigating aging applies best-effort heuristics at allocation time to avoid fragmentation. For example, file systems attempt to place related files close together on disk, while also leaving empty space for future files [7,17,18,25]. Some file systems (including ext4, XFS, Btrfs, and F2FS among those tested in this paper) 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 opinions about the significance of aging. The seminal work of Smith and Seltzer [23] showed that file systems age under realistic workloads, and this aging affects performance. 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 [26] says:

> Modern Linux file systems 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.

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, as seek times have been relatively stable, but bandwidth grows (approximately) as the square root of the capacity. Consider the same level of fragmentation as the above example, but on a new, faster disk with 600MiB/s bandwidth but still a 5ms seek time. Then the 200 seeks would introduce four-fold 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.

As for SSDs, there is a widespread belief that fragmentation is not an issue. For example, PCWorld measured the performance gains from defragmenting an NTFS file system on SSDs [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 issue 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 (i.e., aging) is a first-order performance concern, and that 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 [10–12, 27], 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, establishing that aging is a solvable problem.

Results. We use realistic application workloads to age five widely-used file systems—Btrfs [21], ext4 [7,17,25], F2FS [15], XFS [24] and ZFS [6]—as well as the BetrFS research file system. 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 her workstation. A second workload ages the file system by running a mail-server benchmark, emulating aging over continued use of the server.

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 degradation 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 "unaged" version of the file system, and perform the same measurement. We treat the differences in read throughput between the aged and unaged copies as the result of aging.

We find that:

- All the production file systems age 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 SSDs. Similarly, our mail-server slows down 4–30× on HDDs due to aging.
- Aging can happen quickly. For example, ext4 shows over a 2× slowdown after 100 git pulls; Btrfs and ZFS slow down similarly after 300 pulls.
- BetrFS exhibits essentially no aging. Other than Btrfs, BetrFS's aged performance is better than the other file systems' unaged performance on almost all benchmarks. For instance, on our mail-server workload, unaged ext4 is 6× slower than aged BetrFS.
- 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 1GiB of data. Two of the four took over 25 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 only 10% of files are created out of order relative to the directory structure (and therefore relative to a depth-first search of the directory tree), Btrfs, ext4, F2FS, XFS and ZFS cannot achieve a throughput of 5 MiB/s. If the files are copied completely out of order, then of these only XFS significantly exceeds 1 MiB/s. This need not be the case; BetrFS maintains a throughput of roughly 50 MiB/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 that appends 4 KiB chunks to 10 files in a round-robin fashion on a hard drive, Btrfs and F2FS realize 10 times lower read throughput than if each file is written completely, one at a time. ext4 and XFS are more stable but eventually age by a factor of 2. ZFS has relatively low throughout but did not age. BetrFS throughput is stable, at two thirds of full disk bandwidth throughout the test.

2 Related Work

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

Feature	Btrfs	ext4	F2FS	XFS	ZFS	BetrFS
Grouped allocation within directories	✓	✓		✓	✓	✓
Extents	✓	✓		✓	✓	
Delayed allocation	✓	✓	✓	✓	✓	✓
Packing small files and metadata	✓ (by OID)					✓
Default Node Size	16 K	4 K	4 K	4 K	8 K	2–4 M
Maximum Node Size	64 K	64 K	4 K	64 K	128 K	2–4 M
Rewriting for locality						✓
Batching writes to reduce amplification			✓			✓

Table 1: Principal anti-aging features of the file systems measured in this paper. The top portion of the table are commonly-deployed features, and the bottom portion indicates features our model (§3) indicates are essential; an ideal node size should match the natural transfer size, which is roughly 4 MiB for modern HDDs and SSDs. OID in Btrfs is an object identifier, roughly corresponding to an inode number, which is assigned at creation time.

over fifty real file systems from five servers over durations ranging from one to three years. An overarching goal of Smith and Seltzer's work was to evaluate file systems with representative levels of aging.

Other tools have been subsequently developed for synthetically aging a file system. In order to measure NFS performance, TBBT [28] was designed to synthetically age a disk to create a initial state for NFS trace replay.

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 specify a target layout score for the resulting image.

Both TBBT and Impressions create file systems with a specific level of fragmentation, whereas our study identifies realistic workloads that induce 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. We introduce a variation of this measure, the ***dynamic layout score*** in Section 3.3.

The ***degree of fragmentation*** (***DoF***) is used in the study of fragmentation in mobile devices [13]. DoF is the ratio of the actual number of extents, or ranges of contiguous physical blocks, to the ideal number of extents. Both the layout score and DoF measure how one file is fragmented.

Several studies have reported file system statistics such as number of files, distributions of file sizes and types, and organization of file system namespaces [3, 9, 22]. These statistics can inform parameter choices in aging frameworks like TBBT and Impressions [2, 28].

2.3 Existing Strategies to Mitigate Aging

When files are created or extended, blocks must be allocated to store the new data. Especially when data is rarely or never relocated, as in an update-in-place file system like ext4, initial block allocation decisions determine performance over the life of the file system. Here we outline a few of the strategies use in modern file systems to address aging, primarily at allocation-time (also in the top of Table 1).

Cylinder or Block Groups. FFS [18] introduced the idea of ***cylinder groups***, which later evolved into block groups or allocation groups (XFS). Each group maintains information about its inodes and a bitmap of blocks. A new directory is placed in the cylinder group that contains more than the average number of free inodes, while inodes and data blocks of files in one directory are placed in the same cylinder group when possible.

ZFS [6] is designed to pool storage across multiple devies [6]. ZFS selects from one of a few hundred ***metaslabs*** on a device, based on a weighted calculation of several factors including minimizing seek distances. The metaslab with the highest weight is chosen.

In the case of F2FS [15], a log-structured file system, the disk is divided into segments—the granularity at which the log is garbage collected, or cleaned. The primary locality-related optimization in F2FS is that writes are grouped to improve locality, and dirty segments are filled before finding another segment to write to. In other words, writes with temporal locality are more likely to be placed with physical locality.

Groups are a best-effort approach to directory locality: space is reserved for co-locating files in the same directory, but when space is exhausted, files in the same directory can be scattered across the disk. Similarly, if a file is renamed, it is not physically moved to a new group.

Extents. All of the file systems we measure, except F2FS and BetrFS, allocate space using ***extents***, or runs of physically contiguous blocks. In ext4 [7, 17, 25], for example, an extent can be up to 128 MiB. Extents reduce bookkeeping overheads (storing a range versus an exhaustive list of blocks). Heuristics to select larger extents can improve locality of large files. For instance, ZFS selects from available extents in a metaslab using a first-fit policy.

Delayed Allocation. Most modern file systems, including ext4, XFS, Btrfs, and ZFS, implement delayed allocation, where logical blocks are not allocated until buffers are written to disk. By delaying allocation when a file is growing, the file system can allocate a larger extent for data appended to the same file. However, allocations can only be delayed so long without violating durability and/or consistency requirements; a typical file system

ensures data is dirty no longer than a few seconds. Thus, delaying an allocation only improves locality inasmuch as adjacent data is also written on the same timescale; delayed allocation alone cannot prevent fragmentation when data is added or removed over larger timescales.

Application developers may also request a persistent preallocation of contiguous blocks using fallocate. To take full advantage of this interface, developers must know each file's size in advance. Furthermore, fallocate can only help intrafile fragmentation; there is currently not an analogous interface to ensure directory locality.

Packing small files and metadata. For directories with many small files, an important optimization can be to pack the file contents, and potentially metadata, into a small number of blocks or extents. Btrfs [21] stores metadata of files and directories in copy-on-write B-trees. Small files are broken into one or more fragments, which are packed inside the B-trees. For small files, the fragments are indexed by object identifier (comparable to inode number); the locality of a directory with multiple small files depends upon the proximity of the object identifiers.

BetrFS stores metadata and data as key-value pairs in two B^{ε}-trees. Nodes in a B^{ε}-tree are large (2–4 MiB), amortizing seek costs. Key/value pairs are packed within a node by sort-order, and nodes are periodically rewritten, copy-on-write, as changes are applied in batches.

BetrFS also divides the namespace of the file system into ***zones*** of a desired size (512 KiB by default), in order to maintain locality within a directory as well as implement efficient renames. Each zone root is either a single, large file, or a subdirectory of small files. The key for a file or directory is its relative path to its zone root. The key/value pairs in a zone are contiguous, thereby maintaining locality.

3 A Framework for Aging

3.1 Natural Transfer Size

Our model of aging is based on the observation that the bandwidth of many types of hardware is maximized when I/Os are large; that is, sequential I/Os are faster than random I/Os. We abstract away from the particulars of the storage hardware by defining the ***natural transfer size*** (NTS) to be the amount of sequential data that must be transferred per I/O in order to obtain some fixed fraction of maximum throughput, say 50% or 90%. Reads that involve more than the NTS of a device will run near bandwidth.

From Figure 1, which plots SSD and HDD bandwidth as a function of read size, we conclude that a reasonable NTS for both the SSDs and HDDs we measured is 4MiB.

The cause of the gap between sequential- and random-I/O speeds differs for different hardware. For HDDs, seek times offer a simple explanation. For SSDs, this gap is hard to explain conclusively without vendor support, but common theories include: sequential accesses are easier to stripe across internal banks, better leveraging parallelism [14]; some FTL translation data structures have nonuniform search times [16]; and fragmented SSDs are not able to prefetch data [8] or metadata [13]. Whatever the reason, SSDs show a gap between sequential and random reads, though not as great as on disks.

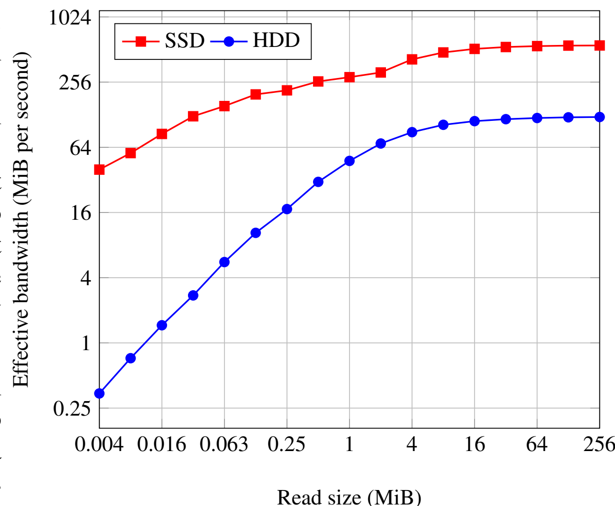


Figure 1: Effective bandwidth vs. read size (higher is better). Even on SSDs, large I/Os can yield an order of magnitude more bandwidth than small I/Os.

In order to avoid aging, file systems should avoid breaking large files into pieces significantly smaller than the NTS of the hardware. They should also group small files that are logically related (close in recursive traversal order) into clusters of size at least the NTS and store the clusters near each other on disk. We consider the major classes of file systems and explore the challenges each file system type encounters in achieving these two goals.

3.2 Allocation Strategies and Aging

The major file systems currently in use can be roughly categorized as B-tree-based, such as XFS, ZFS, and Btrfs, update-in-place, such as ext4, and log-structured, such as F2FS [15]. The research file system that we consider, BetrFS, is based on B^{ε}-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 below.

B-trees. The aging profile of a B-tree depends on the leaf size. If the leaves are much smaller than the NTS, 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 NTS increases *write*

amplification, or the ratio between the amount of data changed and the amount of data 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 under a wide variety of workloads.

In Section 6, 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 packed memory array [5]. However, such an arrangement might well incur an unacceptable performance overhead to keep the leaves arranged in logical order, and we know of no examples of B-trees implemented with such leaf-arrangement algorithms.

Write-Once or Update-in-Place Filesystems. When data is written once and never moved, such as in update-in-place file systems like ext4, sequential order is very difficult to maintain: imagine a workload that writes two files to disk, and then creates files that should logically occur between them. Without moving one of the original files, data cannot be maintained sequentially. Such pathological cases abound, and the process is quite brittle. As noted above, delayed allocation is an attempt to mitigate the effects of such cases by batching writes and updates before committing them to the overall structure.

B^{ε}-trees. B^{ε}-trees batch changes to the file system in a sequence of cascading logs, one per node of the tree. Each time a node overflows, it is flushed to the next node. The seeming disadvantage is that data is written many times, thus increasing the write amplification. However, each time a node is modified, it receives many changes, as opposed to B-tree, which might receive only one change. Thus, a B^{ε}-tree has asymptotically lower write amplification than a B-tree. Consequently, it can have much larger nodes, and typically does in implementation. BetrFS uses a B^{ε}-tree with 4MiB nodes.

Since 4MiB is around the NTS for our storage devices, we expect BetrFS not to age—which we verify below.

Log-structured merge trees (LSMs) [20] and other write-optimized dictionaries can resist aging, depending on the implementation. As with B^{ε}-trees, it is essential that node sizes match the NTS, the schema reflect logical access order, and enough writes are batched to avoid heavy write amplification.

3.3 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 grep test. 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 file system. This 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 by using ext4's du output. We will 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 file system that are allocated in a contiguous sequence in the logical block space. We extend this 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.

One potential shortcoming of this measure is that it does not distinguish between small and large discontiguities. Small discontiguities on a hard drive should induce fewer expensive mechanical seeks than large discontiguities in general, however factors such as track length, difference in angular placement and other geometric considerations can complicate this relationship. A more sophisticated measure of layout might be more predictive. We leave this for further research. On SSD, we have found that the length of discontiguities has a smaller effect. Thus we will show that dynamic layout score strongly correlates with grep test performance on SSD and moderately correlates on hard drive.

4 Experimental Setup

Each experiment compares several file systems: BetrFS, Btrfs, ext4, F2FS, XFS, and ZFS. We use the versions of XFS, Btrfs, ext4 and F2FS 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 used BetrFS 0.3 in the experiments[1]. We use default recommended file system settings unless otherwise noted. Lazy inode table and journal initialization are turned off on ext4, pushing more work onto file system creation time and reducing experimental noise.

All experimental results are 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. Unless

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

otherwise noted, all experiments are cold-cache.

The system runs 64-bit Ubuntu 13.10 server with Linux kernel version 3.11.10 on a bootable USB stick. All HDD tests are performed on two 20GiB partitions located at the outermost region of the drive. For the SSD tests, we additionally partition the remainder of the drive and fill it with random data, although we have preliminary data that indicates this does not affect performance.

5 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 both intra-file fragmentation and inter-file fragmentation and show the performance impact aging can have on read performance in the worst cases.

Intrafile Fragmentation. When a file grows, there may not be room to store the new blocks with the old blocks on disk, and a single file's data may become scattered.

Our benchmark creates 10 files by first creating each file of an initial size, and then appending between 0 and 100 4KiB chunks of random data in a round-robin fashion until each file is 400KiB. In the first round the initial size is 400KiB, so each entire file is written sequentially, one at a time. In subsequent rounds, the initial size becomes smaller, so that the number of round-robin chunks increases until in the last round the data is written entirely with a round-robin of 4KiB chunks. After all the files are written, the disk cache is flushed by remounting, and we wait for 90 seconds before measuring read performance. Some file systems appear to perform background work immediately after mounting that introduced experimental noise; 90 seconds ensures the file system has quiesced.

The aging process this microbenchmark emulates is multiple files growing in length. The file system must allocate space for these files somewhere, but eventually the file must either be moved or fragment.

Given that the data set size is small and the test is designed to run in a short time, an fsync is performed after each file is written in order to defeat deferred allocation. Similar results are obtained if the test waits for 5 seconds between each append operation. If fewer fsyncs are performed or less waiting time is used, then the performance differences are smaller, as the file systems are able to delay allocation, rendering a more contiguous layout.

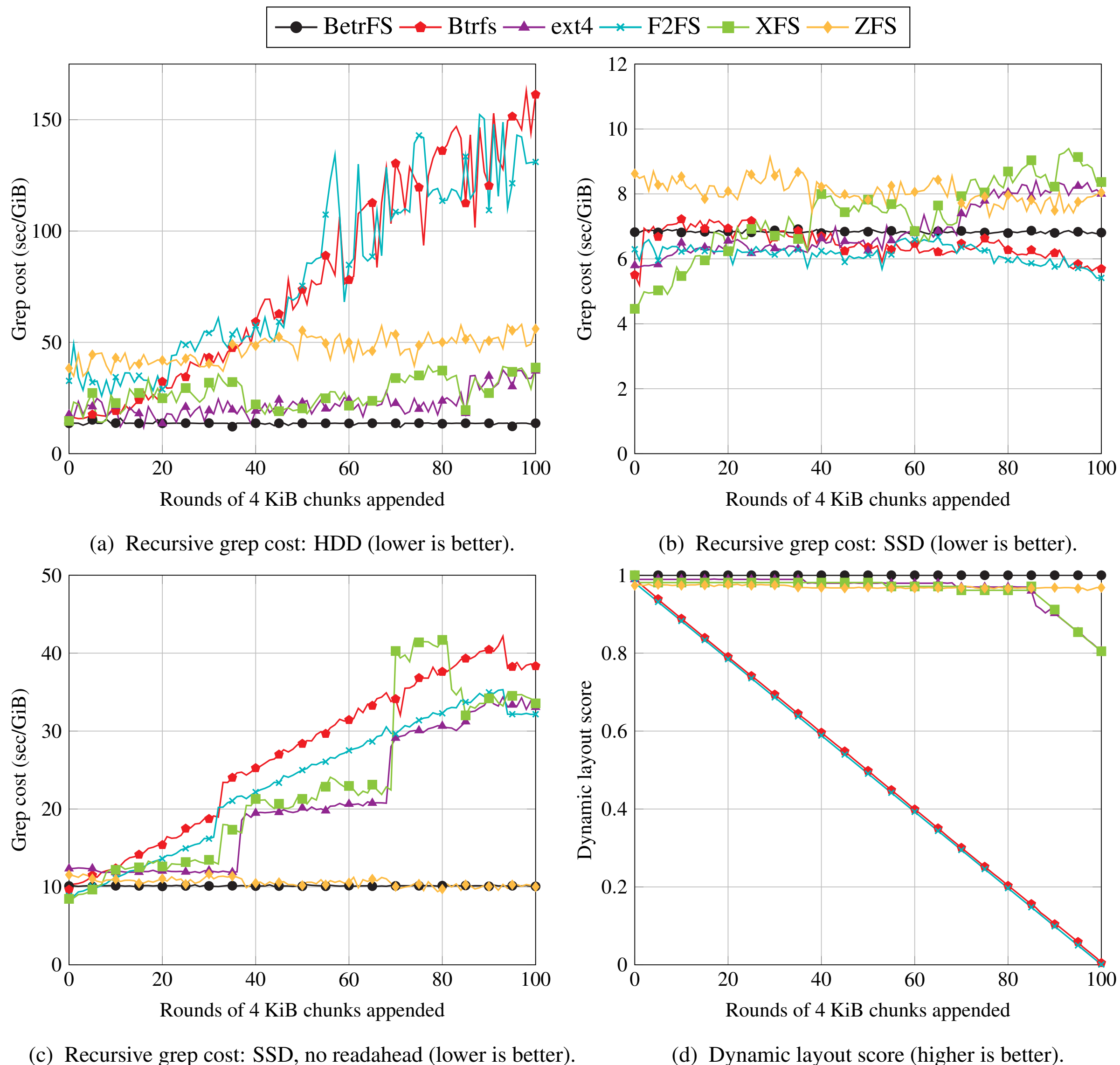
The performance of these file systems on an HDD and SSD are summarized in Figures 2. On HDD, the layout scores generally correlate (-0.93) with the performance of the file systems. On SSD, the file systems all perform similarly (note the scale of the y-axis). In some cases, such as XFS, ext4, and ZFS, there is a correlation, albeit at a small scale. For Btrfs, ext4, XFS, and F2FS, the performance is hidden by read-ahead in the OS or in the case of Btrfs also in the file system itself. If we disable read-ahead, shown in Figure 2c, the performance is more clearly correlated ($-.67$) with layout score. We do note that this relationship on an SSD is still not precise; SSDs are sufficiently fast that factors such as CPU time can also have a significant effect on performance.

Because of the small amount of data and number of files involved in this microbenchmark, we can visualize the layout of the various file systems, shown in Figure 3. Each block of a file is represented by a small vertical bar, and each bar is colored uniquely to one of the ten files. Contiguous regions form a colored rectangle. The visualization suggests, for example, that ext4 both tries to keep files and eventually larger file fragments sequential, whereas Btrfs and F2FS interleave the round robin chunks on the end of the sequential data. This interleaving can help explain why Btrfs and F2FS perform the way they do: the interleaved sections must be read through in full each time a file is requested, which by the end of the test takes roughly 10 times as long. ext4 and XFS manage to keep the files in larger extents, although the extents get smaller as the test progresses, and, by the end of the benchmark, these file systems also have chunks of interleaved data; this is why ext4 and XFS's dynamic layout scores decline. ZFS keeps the files in multiple chunks through the test; in doing so it sacrifices some performance in all states, but does not degrade.

Unfortunately, this sort of visualization doesn't work for BetrFS, because this small amount of data fits entirely in a leaf. Thus, BetrFS will read all this data into memory in one sequential read. This results is some read amplification, but, on an HDD, only one seek.

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 collocated in the LBA space.

We present a microbenchmark to measure the impact of namespace creation order on interfile locality. It takes a given "real-life" file structure, in this case the Tensorflow repository obtained from `github.com`, and replaces each of the files by 4KiB of random data. This gives us a "natural" directory structure, but isolates the effect of file ordering without the influence of intrafile layout. The benchmark creates a sorted list of the files as well as two random permutations of that list. On each round of the test, the benchmark copies all of the files, creating directories as needed with `cp --parents`. However, on the *n*th round, it swaps the order in which the first *n*% of files appearing in the random permutations are copied. Thus, the first round will be an in-order copy, and subsequent



(a) Recursive grep cost: HDD (lower is better).

(b) Recursive grep cost: SSD (lower is better).

(c) Recursive grep cost: SSD, no readahead (lower is better).

(d) Dynamic layout score (higher is better).

Figure 2: Intrafile benchmark: 4KiB chunks are appended round-robin to sequential data to create 10 400KiB files. Dynamic layout scores generally correlate with read performance as measured by the recursive grep test; on an SSD, this effect is hidden by the readahead buffer.

rounds will be copied in a progressively more random order until the last round is a fully random-order copy.

The results of this test are shown in Figure 4. On hard drive, all the file systems except BetrFS and XFS show a precipitous performance decline even if only a small percentage of the files are copied out of order. F2FS's performance is poor enough to be out of scale for this figure, but it ends up taking over 4000 seconds per GiB at round 100; this is not entirely unexpected as it is not designed to be used on hard drive. XFS is somewhat more stable, although it is 13-35 times slower than drive bandwidth throughout the test, even on an in-order copy. BetrFS consistently performs around 1/3 of bandwidth, which by the end of the test is 10 times faster than XFS, and 25 times faster than the other file systems. The dynamic layout scores are moderately correlated with this performance (-0.57).

On SSD, half the file systems perform stably throughout the test with varying degrees of performance. The other half have a very sharp slowdown between the in-order state and the 10% out-of-order state. These two modes are reflected in their dynamic layout scores as well. While ext4 and ZFS are stable, their performance is worse than the best cases of several other file systems. BetrFS is the only file system with stable fast performance; it is faster in every round than any other file system even in their best case: the in-order copy. In this cases the performance strongly correlates with the dy-

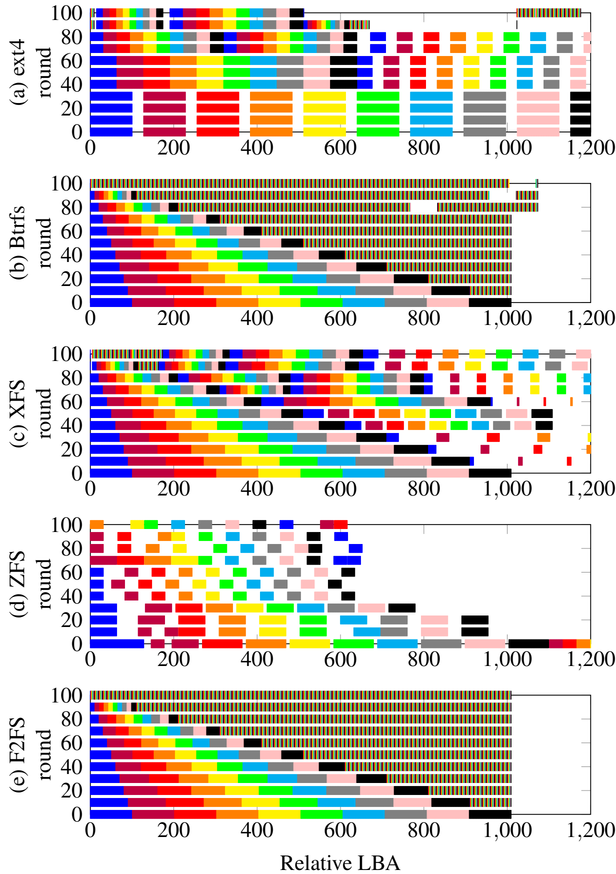


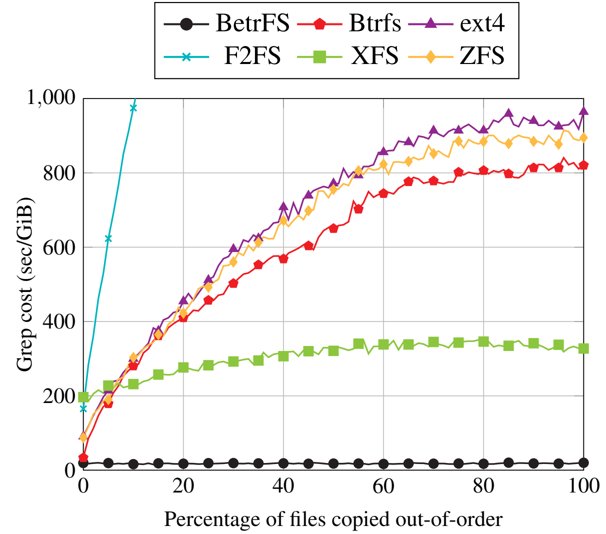
Figure 3: Intrafile benchmark layout visualization. Each color represents blocks of a file. The x-axis is the logical block address (LBA) of the file block relative to the first LBA of any file block, and y-axis is the round of the experiment. Rectangle sizes indicate contiguous placement, where larger is better. The brown regions with vertical lines indicate interleaved blocks of all 10 files. Some blocks are not shown for ext4, XFS and ZFS.

namic layout score (−0.83).

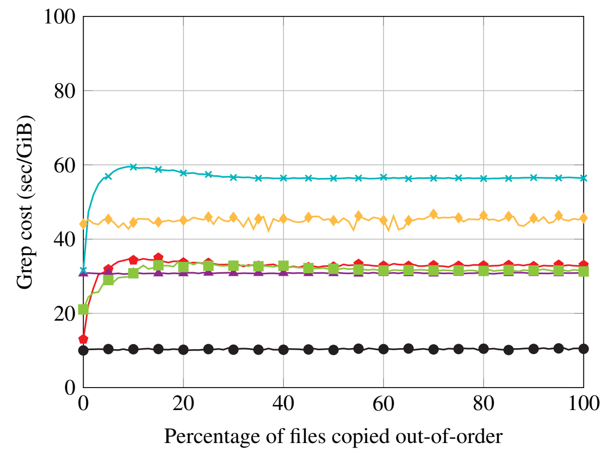
6 Application Level Read-Aging: Git

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

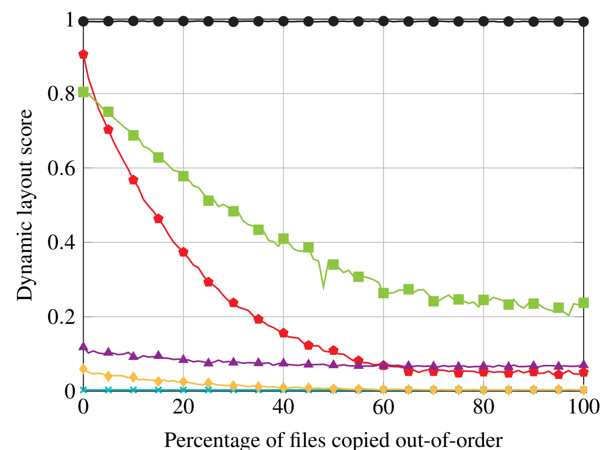
Git is a distributed version control system that enables collaborating developers to synchronize their source code changes. Git users ***pull*** changes from other developers, which then get merged with their own changes. In a typical workload, a Git user may perform pulls multiple times per day over 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.



(a) Recursive grep cost: HDD (Lower is better).



(b) Recursive grep cost: SSD (Lower is better).



(c) Dynamic layout score (higher is better).

Figure 4: Interfile benchmark: The TensorFlow github repository with all files replaced by 4KiB random data and copied in varying degrees of order. Dynamic layout scores again are predictive of recursive grep test performance.

Thus, Git performs many operations which are similar to those shown in Section 5 that cause file system aging.

We present a git benchmark that performs 10,000 pulls from the Linux git repository, starting from the initial commit. After every 100 pulls, the benchmark performs a recursive grep test and computes the file system's dynamic layout score. This score is compared to the same contents copied to a freshly formatted partition.

On a hard disk (Figure 5a), there is a clear aging trend in all file systems except BetrFS. By the end of the experiment, all the file systems except BetrFS show performance drops under aging on the order of at least 3x and as much as 15x relative to their unaged versions. All are at least 15x worse than BetrFS. In all of the experiments in this section, F2FS ages considerably more than all other file systems, commensurate with significantly lower layout scores than the other file systems—indicating less effective locality in data placement. The overall correlation between grep performance and dynamic layout score is moderate, at -0.41.

On an SSD (Figure 5c), Btrfs and XFS show clear signs of aging, although they converge to a fully aged configuration after only about 1,000 pulls. While the effect is not as drastic as on HDD, in all the traditional file systems we see slowdowns of 2x-4x over BetrFS, which does not slow down. In fact, aged BetrFS on the HDD outperforms all the other aged file systems on an SSD, and is close even when they are unaged. Again, this performance decline is strongly correlated (-0.79) with the dynamic layout scores.

The aged and unaged performance of ext4 and ZFS are comparable, and slower than several other file systems. We believe this is because the average file size decreases over the course of the test, and these file systems are not as well-tuned for small files. To test this hypothesis, we constructed synthetic workloads similar to the interfile fragmentation microbenchmark (Section 5), but varied the file size (in the microbenchmark it was uniformly 4KB). Figure 6 shows both the measured, average file size of the git workload (one point is one pull), and the microbenchmark. Overall, there is a clear relationship between the average file size and grep cost.

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, as show in Figures 5b, 5d and 5f, the effect of aging is even more pronounced, and the zig-zags essentially disappear.

On both the HDD and SSD, the same patterns emerge as with garbage collection on, but exacerbated: F2FS aging is by far the most extreme. ZFS ages considerably on the HDD, but not on the SSD. ZFS on SSD and ext4 perform worse than the other file systems (except F2FS aged), but do not age particularly. XFS and Btrfs both aged significantly, around 2x each, and BetrFS has strong, level performance in both states. This performance correlates with dynamic layout score both on SSD (-0.78) and moderately so on HDD (-0.54).

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, aging clearly has a major performance impact.

Git Workload with Warm Cache. The tests we have presented so far have all been performed with a cold cache, so that they more or less directly test the performance of the file systems' on-disk layout under various aging conditions. In practice, however, some data will be in cache, and so it is natural to ask how much the layout choices that the file system makes will affect the overall performance with a warm cache.

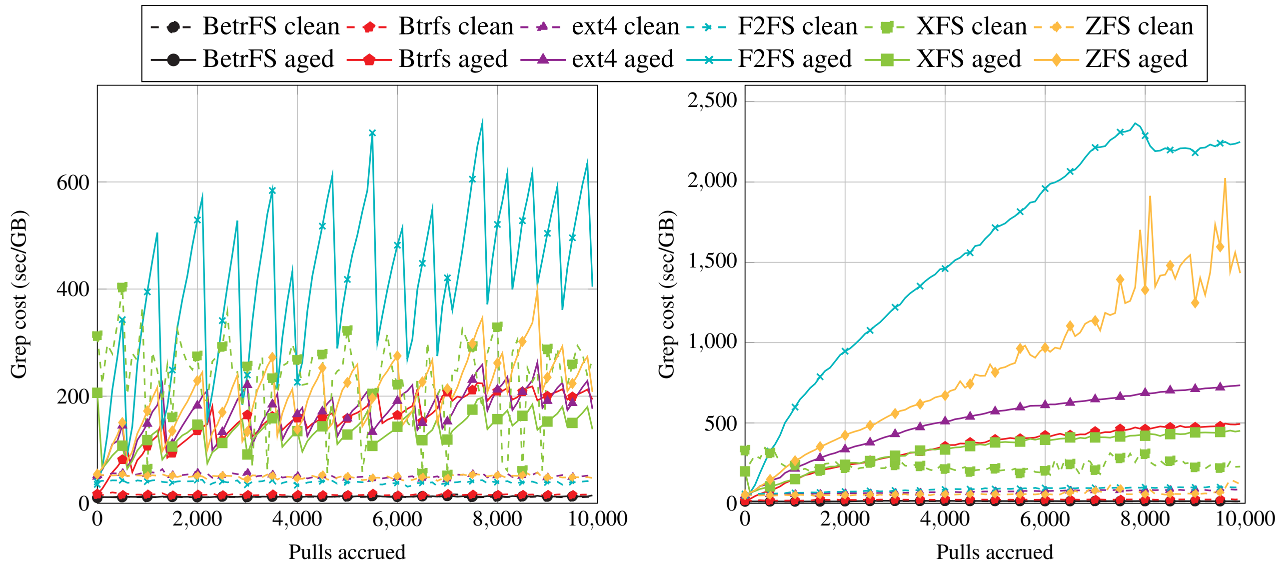
We evaluate the sensitivity of the git workloads to varying amounts of system RAM. We use the same procedure as above, except that we do not flush any caches or remount the hard drive between iterations. This test is performed on a hard drive with git garbage collection off. The size of the data on disk is initially about 280MiB and grows throughout the test to approximately 1GiB.

The results are summarized in Figure 7. We present data for ext4 and F2FS; the results for Btrfs, XFS and ZFS are similar. BetrFS is a research prototype and unstable under memory pressure; although we plan to fix these issues in the future, we omit this comparison.

In general, when the caches are warm and there is sufficient memory to keep all the data in cache, then the read is very fast. However, as soon as there is no longer sufficient memory, the performance of the aged file system with a warm cache is generally worse than unaged with a cold cache. In general, unless all data fits into DRAM, a good layout matters more than a having a warm cache.

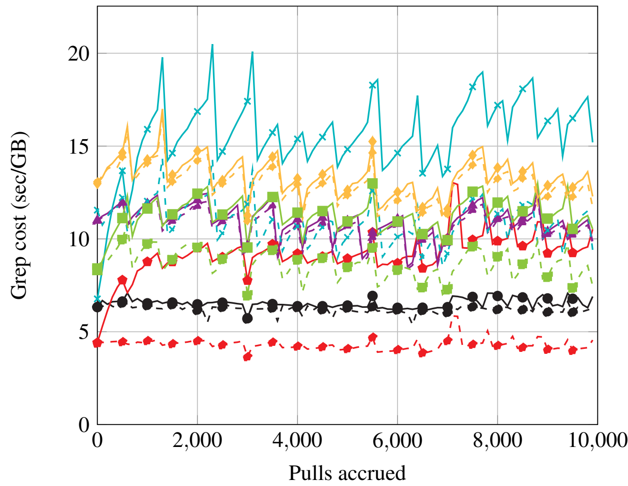
Btrfs Node-Size Trade-Off. Btrfs allows users to specify the node size of its metadata B-tree at creation time. Because small files are stored in the metadata B-tree, a larger node size results in a less fragmented file system, at a cost of more expensive metadata updates.

We present the git test with a 4KiB node size, the default setting, as well as 8KiB, 16KiB, 32KiB, and 64KiB (the maximum). Figure 8a shows similar performance graphs to Figure 5, one line for each node size. The 4KiB node size has the worst read performance by the end of

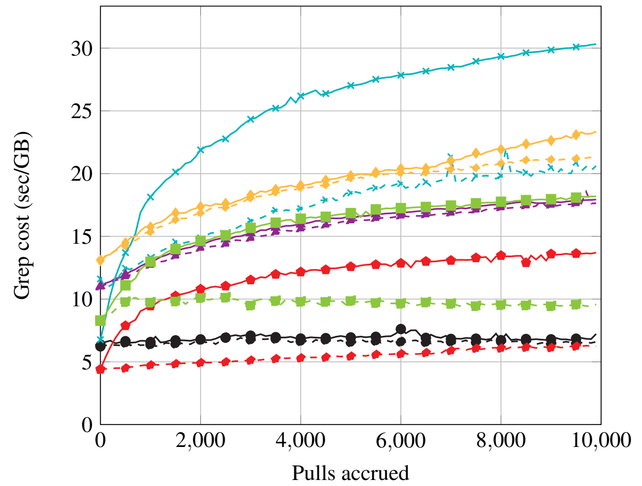


(a) HDD, git garbage collection on (Lower is better).

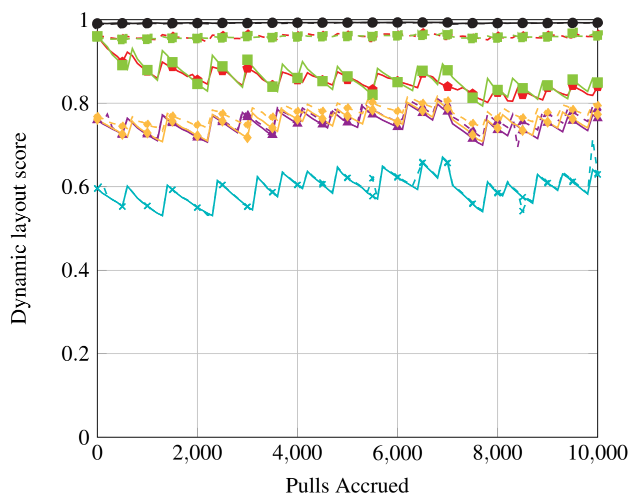
(b) HDD, git garbage collection off (Lower is better).



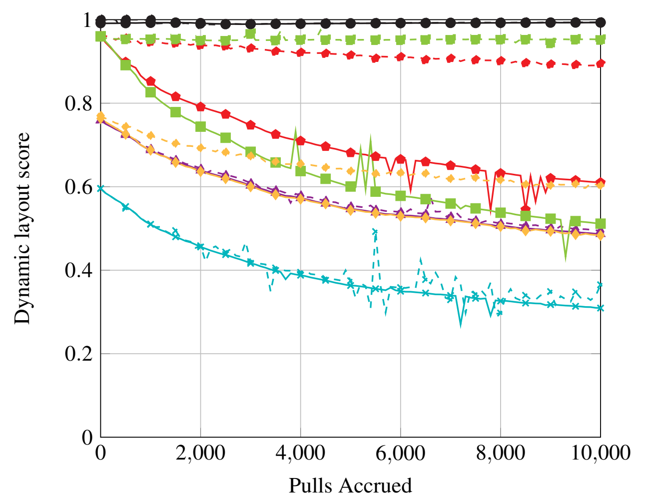
(c) SSD, git garbage collection on (Lower is better).



(d) SSD, git garbage collection off (Lower is better).



(e) Dynamic layout score: git garbage collection on (Higher is better).



(f) Dynamic layout score: git garbage collection off (Higher is better).

Figure 5: Git read-aging experimental results: On-disk layout as measured by dynamic layout score generally is predictive of read performance.

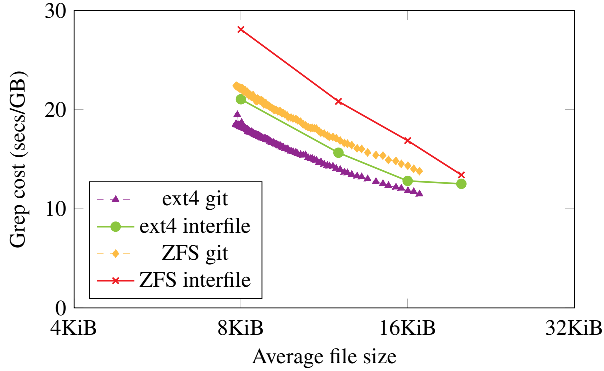


Figure 6: Average file size versus unaged grep costs (lower is better) on SSD. Each point in the git line is the average file size for the git experiment, compared to a microbenchmark with all files set to a given size.

the test, and the performance consistently improves as we increase the node size all the way to 64KiB. Figure 8b plots the number of 4KiB blocks written to disk between each test (within the 100 pulls). As expected, the 64KiB node size writes the maximum number of blocks and the 4KiB node writes the least. We thus demonstrate—as predicted by our model—that aging is reduced by a larger block size, but at the cost of write-amplification.

7 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. We simulate 2 users, each having 80 mailboxes receiving new email, deleting old emails, and searching through their 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, and, because inserts and deletes are balanced, mailbox size tends to stay around 1,000. 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 file system, and run a recursive grep.

Figure 9 shows the read costs in seconds per GiB of the grep test on hard disk. Although the unaged versions of all file systems show consistent performance over the life of the benchmark, the aged versions of ext4, 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, taking about 20 minutes per GiB by

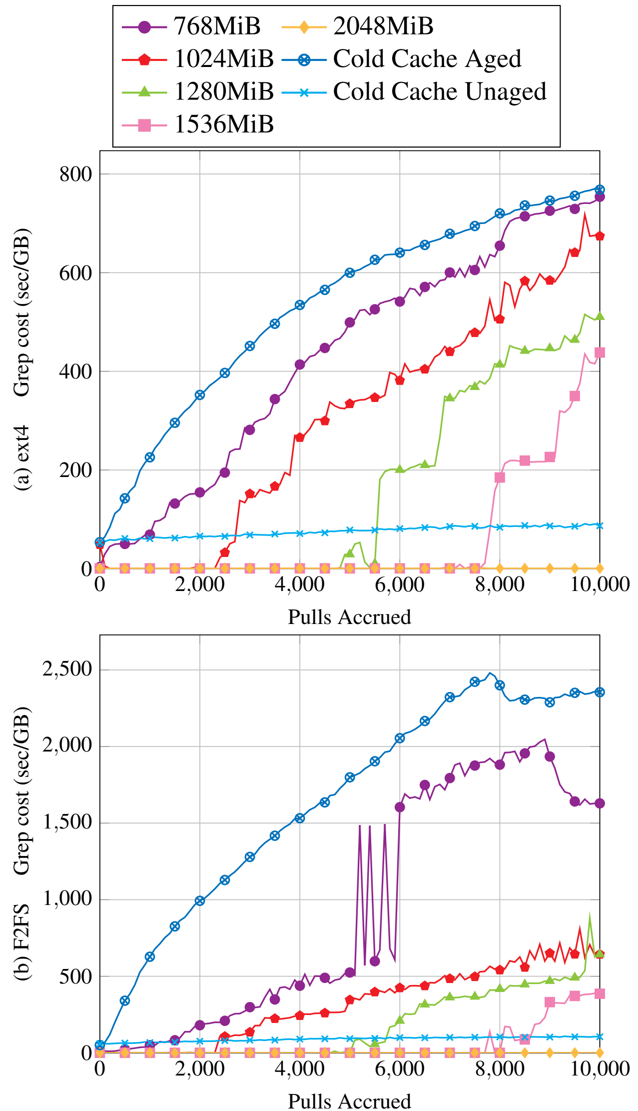


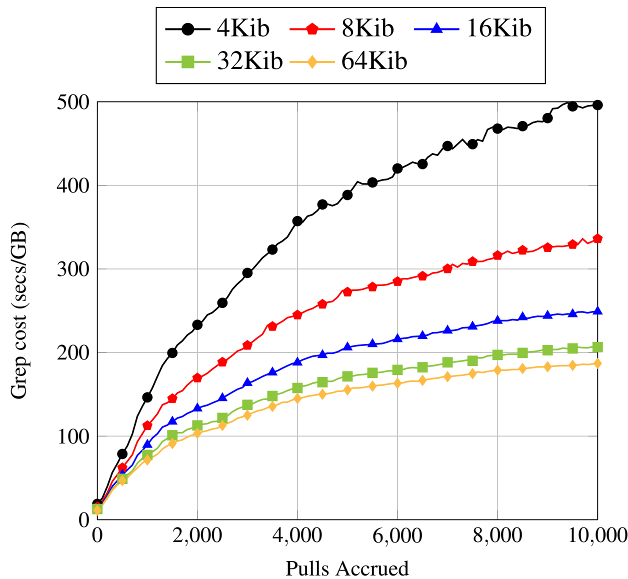
Figure 7: Grep costs as a function of git pulls with warm cache and varying system RAM on ext4 (top) and F2FS (bottom). Lower is better.

cycle 20. However, the aged version of BetrFS does not slow down. As with the other HDD experiments, dynamic layout score is moderately correlated (−0.63) with grep cost.

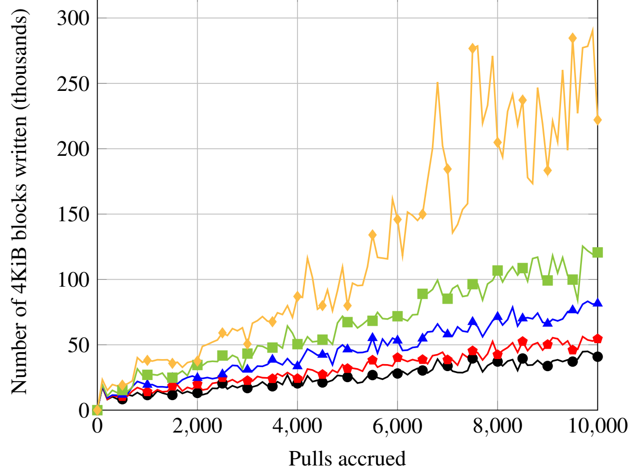
8 Conclusion

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 file system's current state and the potential to make modifications without disrupting the future order. Rewriting data multiple times allows the file system to maintain locality. The

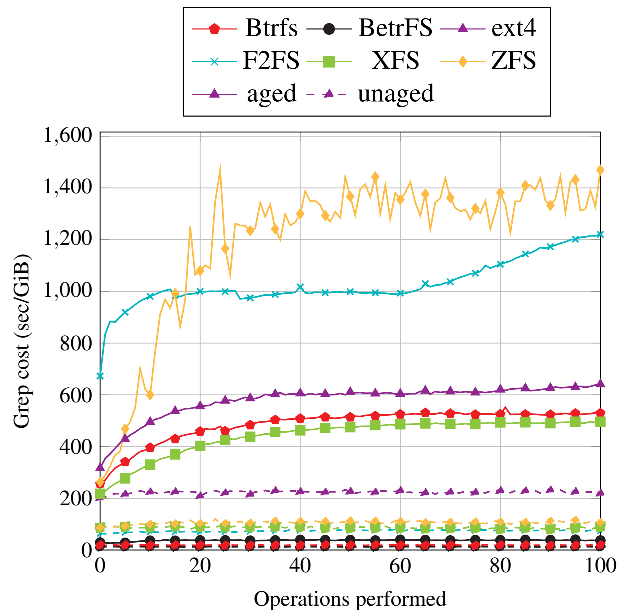


(a) Grep cost at different node sizes (lower is better).

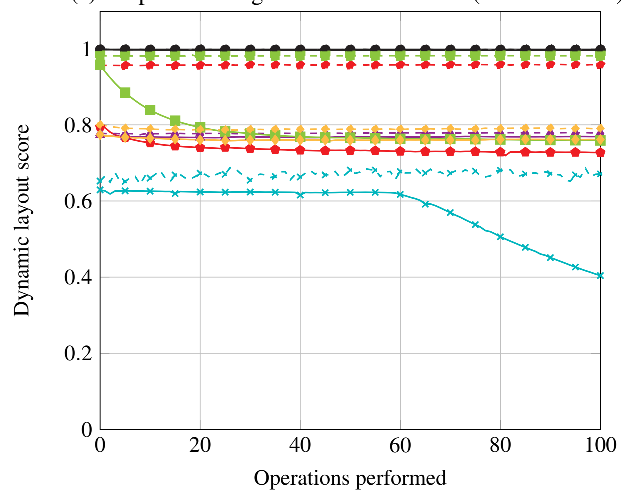


(b) Write amplification at different node sizes (lower is better).

Figure 8: Aging and write amplification on Btrfs, with varying node sizes, under the git aging benchmark.



(a) Grep cost during mailserver workload (lower is better).



(b) Mailserver layout (higher is better).

Figure 9: Mailserver performance and layout scores.

overhead of these multiple writes can be managed by rewriting data in batches, as is done in write-optimized dictionaries.

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.

We experimentally confirmed our expectation that non-write-optimized file systems would age, but we were surprised by how quickly and dramatically aging impacts performance. This rapid aging is important: a user's experience with unaged file systems is likely so fleeting that they do not notice performance degradation. Instead, the performance costs of aging are built into their expectations of file system performance.

Finally, because representative aging is a difficult goal, simulating multi-year workloads, many research papers benchmark on unaged file systems. Our results indicate that it is relatively easy to quickly drive a file system into an aged state—even if this state is not precisely the state of the file system after, say, three years of typical use—and this degraded state can be easily measured.

Acknowledgments

We thank the anonymous reviewers and our shepherd Philip Shilane for their insightful comments on earlier drafts of the work. Part of this work was done while Jiao, Porter, Yuan, and Zhan were at Stony Brook University. This research was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, CNS-1161541, IIS-1247750, CCF-1314547, and VMware.

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] BENDER, M. A., DEMAINE, E., AND FARACH-COLTON, M. Cache-oblivious B-trees. *SIAM J. Comput. 35*, 2 (2005), 341–358.

[6] 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.

[7] 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`.

[8] 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.

[9] 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.

[10] ESMET, J., BENDER, M. A., FARACH-COLTON, M., AND KUSZMAUL, B. C. The TokuFS streaming file system. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)* (2012).

[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: A right-optimized write-optimized file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (Santa Clara, CA, USA, Feb. 22–25 2015), pp. 301–315.

[12] 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.

[13] 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.

[14] 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.

[15] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (Santa Clara, CA, USA, Feb. 22–25 2015), pp. 273–286.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] O'NEIL, P., CHENG, E., GAWLIC, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385. http://dx.doi.org/10.1007/s002360050048doi: 10.1007/s002360050048.

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

[22] 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.

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

[26] 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.

[27] 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`.

[28] 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.