

A Compleat File System for Commodity SSDs

Anonymous Author(s)

Submission Id: 170

Abstract

Despite the existence of file systems tailored for commodity SSDs and over a decade of research into flash file systems, this paper shows that no single Linux file system performs consistently well on an SSD across different workloads. Specifically, we define a compleat file system as one where no workloads realize less than 30% of the best file system’s performance, and most, if not all, workloads realize at least 85% of the best file system’s performance, across a diverse set of microbenchmarks and applications. No commodity file system is compleat on commodity SSDs.

This paper demonstrates that one can construct a single compleat file system for commodity SSDs. In the HDD space, BetrFS has met a similar goal using a write-optimized dictionary (WOD) to index persistent data. BetrFS roughly matches the fastest Linux file systems on an HDD in its worst cases, and, in its best cases, improves upon the best file systems’ performance by up to two orders of magnitude—but these results do not translate to SSD performance. This paper contributes a range of techniques to improve SSD file system performance. First, the paper presents a consolidated, cooperative architecture for running an in-kernel key-value store more efficiently than BetrFS’s stacked file system architecture. Second, the paper presents optimizations for range operations, reducing recursive deletion costs by an order of magnitude. The paper then presents a cooperative memory management design to efficiently manage large IO buffers necessary for efficient SSD utilization. Fourth, the paper presents a technique to share cached file system data, copy-on-write between the VFS page cache and the WOD, preserving strong crash consistency semantics without obstructing application writes.

With these optimizations, our file system is not only compleat, it is either the fastest or within 15% of the fastest general-purpose Linux file system on a range of microbenchmarks. At best, these optimizations improve random write throughput by $6\times$ compared to the fastest SSD file system. At worst, our file system is competitive with the other baseline file systems. These improvements translate to application-level gains; for instance, Dovecot mailserver and an rsync of the Linux source show up to $1.13\times$ and $1.8\times$ speedup over other commodity file systems, respectively.

1 Introduction

Flash-based Solid State Drives (SSDs) are ubiquitous. Unfortunately, there is no single file system that performs consistently well across a range of different operations or workloads on commodity SSDs. Table 1 (details in §7) illustrates this point: among a set of common general-purpose Linux file

systems, no single design offers top-of-the-line performance across the board. In each benchmark, performance tends to be bi-modal: each file system is either within 25% of the best, or very far from it.

Even F2FS [13], a log-structured file system designed explicitly for SSDs, fails to extract the device’s full throughput for random-writes—a workload where log-structured designs should shine. These results are corroborated elsewhere; for instance, one study shows that ext4 is the fastest file system for blogbench, whereas F2FS is the fastest on the same SSD for compile bench, and XFS is the fastest for dbench [12].

In all of these benchmarks, the slowest file system ranged from roughly a *half to a fifth* of the performance of the fastest. The end result is that, at installation time, a system administrator decides which workloads will enjoy the underlying hardware’s full performance, and which ones will be bottlenecked on the file system. Although work is ongoing to design and leverage high-end flash storage devices, such as NVMe and non-volatile main memory, this paper focuses on inexpensive SSDs because current file systems already leave performance on the table with slower SSDs, and addressing these computational bottlenecks is a prerequisite to saturating faster flash devices. Further, commodity SSDs still represent a tremendous share of deployments worldwide.

This paper investigates whether it is possible to construct a **compleat**¹ file system for commodity SSDs, on which *any* operation roughly matches the fastest implementation of that operation on that device. Concretely, given a set of representative workloads, we set a goal of building a file system where no operation is worse than 30% of the best implementation, and most, if not all, realize at least 85% of the best implementation’s performance. In Table 1, data points where a file system realizes less than 30% of the best file system’s performance are shaded in red, and those that realize more than 85% of the best implementation’s performance are shaded in green. Every baseline file system has at least one red cell.

Recently, file systems built using write-optimized dictionaries (WODs), such as LSM-trees or B^e-trees, have shown how to meet this objective on an HDD. Specifically, BetrFS [35] shows that, on an HDD, a write-optimized file system can offer sequential reads and writes at disk bandwidth while performing random writes orders-of-magnitude faster than an update-in-place file system. Furthermore, it demonstrates significant performance improvements to metadata intensive operations, such as directory traversals and file creation.

Unfortunately, many of the BetrFS performance properties do not immediately translate to SSDs. In particular, Table 1

¹**compleat**, *adjective*, having all necessary or desired elements or skills.

File System	Sequential I/O		Random Writes		Tokubench	Utility Latency		
	Read(80g)	Write(80g)	4 KB	4 Byte		grep	rm	find
Ext4	534 (0)	316 (6)	16 (3)	0.026 (0)	13.6 (1.5)	10.15 (0.20)	1.81 (0.08)	0.86 (0.02)
Btrfs	568 (0)	328 (2)	13 (0)	0.024 (0)	6.0 (0.2)	4.61 (0.21)	2.53 (0.05)	0.78 (0.16)
Xfs	531 (6)	315 (4)	19 (2)	0.027 (0)	4.5 (0.1)	6.09 (0.84)	2.74 (0.07)	0.84 (0.06)
f2fs	528 (1)	320 (6)	16 (2)	0.033 (0)	4.7 (0.2)	4.72 (0.53)	2.36 (0.07)	0.83 (0.06)
zfs	551 (0)	304 (7)	8 (0)	0.008 (0)	12.5 (0.1)	1.25 (0.01)	3.31 (0.37)	0.43 (0.01)
BetrFS	181 (40)	55 (6)	92 (7)	0.269 (0)	4.0 (0.1)	2.46 (0.15)	51.41 (6.96)	0.27 (0.01)
FestiFS	497 (0)	310 (3)	116 (2)	0.363 (0)	11.8 (0.2)	1.36 (0.02)	1.57 (0.04)	0.22 (0.01)

Table 1. Throughput in MB/s (left) and latency in seconds (right) of various file systems on a commodity SSD. Higher throughput and lower latency are better. Standard deviation is in parentheses. Btrfs and BetrFS v0.6 have compression disabled. Any data points within 15% of the highest throughput or lowest latency are highlighted in green; those that are less than 30% of the best throughput (or more than $3.33\times$ the best latency) are highlighted in red.

shows that, on a commodity SSD, BetrFS only achieves a third to a sixth of the sequential bandwidth of other file systems, and the gains for searches are mixed. It seems counterintuitive that a file system would exhibit such different performance profiles when the only system change is a faster block device, but there are principled reasons why this is so.

This paper investigates these bottlenecks and shows key techniques that enable a file system to offer top-of-the-line performance across a wide range of benchmarks on commodity SSDs. We present and evaluate these results in a prototype file system, called **BetrFS v0.6**, forked from BetrFS v0.4, the most recent public release at github.com/oscarlab/betrfs.

Consolidated layering. This paper presents a simplified storage substrate for an in-kernel write-optimized key-value store (§3). BetrFS’s B^e -tree is stacked on ext4, essentially using ext4 as a block allocator. In addition to well-sung problems like double journaling and double caching, these abstraction boundaries introduce a number of subtle issues. For example, both the baseline BetrFS on-disk layout and Linux support for stacking file systems make zero-copy I/O behavior effectively impossible. Similarly, the ideal layer[s] for implementing read-ahead heuristics is unclear: traditionally this is done at lower levels, but the BetrFS key-value layer has visibility into both on-disk layout and higher-level semantics.

To address these issues, we define a simple, persistent file layer that has immutable metadata and that provides the bare minimum feature set required to support an in-kernel, write-optimized key-value store. In several cases, clarifying the division of labor among layers requires modifying the key-value store to implement functionality in different ways, such as moving read-ahead into the key-value store layer, as hinted at above. In the case of metadata caching, the VFS structures are specialized and effective; minor changes to BetrFS’s VFS interaction yields more effective batching of metadata updates, and better economizes B^e -tree queries. In total, these changes improve sequential write throughput over BetrFS by $4\times$, and reduce grep time by 41%.

Although this layering choice is specific to BetrFS, we note there is a larger, ongoing debate in the community about how to balance the need for rapid prototyping with representative

performance data [17]. Most commonly, this discussion focuses on whether user-level frameworks, such as FUSE, are representative [31]. Our results indicate that, for a reasonable subset of POSIX, one can efficiently “shim” a user-level implementation into Linux, gaining the benefits of both approaches.

Keyspace ranges as first-class primitives. This paper presents optimizations for *range operations* that act on sets of key-value pairs that are contiguous in the keyspace (§4). By expressing file-system operations as range operations, we unlock data-structural optimizations that are critical for good performance. Specifically, BetrFS previously introduced a range delete [35]—used for recursive file deletion—and a range rename [37] operation. These operations unlock optimizations, such as unlinking entire nodes or subtrees that are covered by a range, and are necessary to keep a one-to-one correspondence between file system and key-value store operations.

Because techniques based on modifying contiguous keyspace ranges are relatively new, they have been studied less than other WOD optimizations; but they are gaining popularity. For instance, RocksDB also recently introduced a range delete operation [16]. We find that, in moving from a hard drive to an SSD, range operations are too CPU intensive to keep up with a faster device. We present additional optimizations that recover this performance, as well as simple changes to VFS caching behavior that eliminate redundant queries. These optimizations speed up recursive deletion by $9\times$, making BetrFS v0.6 comparable to other file systems on this workload.

Cooperative memory management. This paper describes cooperative memory management strategies for supporting the large buffers required by write-optimized key-value stores (§5). In particular, to aggregate small updates into large I/Os, a write-optimized key-value store must be able to efficiently allocate and potentially resize buffers that are on the order of hundreds of kilobytes to megabytes in size—because of the nature of how updates are aggregated, it is often difficult to accurately predict buffer sizes at allocation time. Yet Linux’s

internal kernel memory allocators are primarily optimized for pages or small objects, not buffers on the order of megabytes. Similarly, dynamically adjusting kernel mappings of large buffers can be expensive, involving TLB shutdowns across cores. By adopting new memory management strategies, BetrFS v0.6 improves performance across the board, including increases of 25% and 31% over BetrFS's already strong 4KiB and 4B random write throughputs, respectively.

More broadly, the Linux kernel's allocation interfaces are optimized for relatively small objects; managing large buffers is inefficient. Although scatter-gather style IO can mitigate this issue for large data extents, similar issues are likely to arise in log-structured file systems or other file systems that need to serialize metadata or other smaller objects into larger on-disk extents. As devices continue to grow faster, the computational budget per I/O will continue to shrink. Thus, we expect these memory management techniques to be of broad utility.

VFS and key-value store integration. Finally, this paper describes a strategy for sharing versioned data, copy-on-write, between the VFS page cache and a write-optimized key-value store (§6). BetrFS's strong, point-in-time crash-consistency guarantees depend on potentially writing multiple versions of a data block to disk, but the Linux VFS is optimized for update-in-place style file system behaviors, such as in ext4's ordered data mode. Specifically, when writing a dirty page, the VFS blocks subsequent updates to that page until write-back is complete or the page is copied. If the VFS blocks page updates, it prevents write-optimized key-value stores from aggregating application-level writes into larger device-level I/Os, harming performance. Prior work skirts the issue by copying all data, but such an approach wastes memory, squanders CPU cycles in the common case, and effectively converts I/O-bound workloads into CPU-bound workloads. This paper presents a solution that, in the common case, shares pages copy-on-write between the key-value store and the VFS page cache and effectively passes pages by reference through the levels of the key-value store for efficient aggregation. Subsequent writes are not obstructed, and sequential-write throughput has now improved $6\times$ over BetrFS, to within our 15% goal of the fastest sequential-write implementation on an SSD file system.

We note that it is common for file systems with more complex on-disk formats, such as ZFS, to maintain their own caches that are distinct from the VFS page cache [8]. The design in this paper instead facilitates broad reuse of existing VFS caching mechanisms.

Results. Our microbenchmark results are summarized in Table 1 (additional workloads presented in § 7). At its best, BetrFS v0.6 shows a $10\times$ improvement upon 4B random write performance of F2FS—a log-structured file system designed for SSDs; this is 35% higher than BetrFS. At its worst, all cells for BetrFS v0.6 are within our 15% goal.

Our results appear surprising at first, because write-optimized file systems primarily improve performance by (1) coalescing small or random writes into large sequential I/Os, and (2) preserving locality so that sequential file reads translate into large sequential I/Os. Since SSDs have good random I/O, coalescing I/Os may seem unnecessary. Yet, even on an SSD, coalescing writes significantly improves both random write performance and the efficiency of subsequent reads.

We will release the BetrFS v0.6 source code upon publication.

2 Background

This section presents background on BetrFS, the baseline file system forked to make BetrFS v0.6. This section summarizes key details needed to understand the rest of this paper, some of which have not been documented previously. Additional BetrFS details are available elsewhere [3, 9, 36, 37].

2.1 B^E-tree Overview

BetrFS is an in-kernel Linux file system that uses a B^E-tree [3] to index on-disk data. A B^E-tree exports a dictionary (i.e., key-value) interface, similar to an LSM-tree [25]. Like a B-tree, a B^E-tree stores key-value pairs in leaves. Unlike a B-tree, a B^E-tree encodes mutations to key-value pairs as *messages*, and each interior tree node stores a log of recent messages. B^E-tree nodes are also considerably larger than nodes in a typical B-tree, typically 2–4 MiB range.

B^E-trees *batch* updates for good performance. Messages accumulate in the root until there are enough changes to amortize the cost of rewriting the root and at least one child. This process of *flushing* messages from parent to child recurs down the tree; LSM-tree compaction serves a similar role. This strategy of batching updates is called *write optimization*, and although individual updates are rewritten at each level of the tree, the rewriting costs are divided by the batch size, which can yield an average cost much smaller than a single I/O.

A related benefit of this technique is that searches, such as during `grep`, are much faster because rewriting nodes *recovers* locality [5, 6]. Combining improved locality with larger nodes means that a B^E-tree search (or range query) requires fewer total I/Os than a typical file system index structure.

Unlike a pure logging data structure, a B^E-tree has the invariant that all messages that target a given key lie on a single root-to-leaf path. Thus, finding the latest version of a given key-value pair requires reading a logarithmic number of nodes in the worst case, possibly reconstructing the value by applying messages. Of course, caching reduces I/O costs in practice, as does creating and caching *materialized views* of key-value pairs with all of the relevant messages applied.

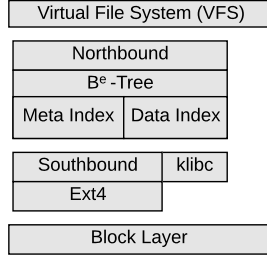


Figure 1. BetrFS Architecture. The “northbound” layer converts VFS operations into key-value store operations, indexed by full path. The B^ε-tree code maintains two indexes, which are stored on ext4.

2.2 BetrFS Architecture

Figure 1 illustrates the baseline BetrFS architecture. System calls pass through the VFS layer and caches, as in any other file system. The BetrFS “northbound” layer translates VFS operations into key-value operations and passes them to the B^ε-tree implementation, which was ported into the Linux kernel from TokuDB. As a user-level key-value store and database, TokuDB was programmed against a file abstraction; the “southbound” layer translates its POSIX-style file API into ext4 requests. In practice, the B^ε-tree implementation only writes to 11 files, and only to two files with any frequency: one for the log and one that stores the B^ε-tree nodes. Finally, a small shim layer, called klibc, translates additional supporting functions, such as memory allocation, to kernel-level APIs.

BetrFS maintains two indexes: one for metadata and one for file data. The metadata index uses complete paths as keys, and stores `stat` metadata structures. The data index stores 4KiB file blocks using (full path, block index) tuples as keys.

Most VFS-level operations are straightforward translations to key-value operations. For example, a file is created by adding an entry to the metadata index (where the key is the full path name). One can append to a file by inserting 4KiB blocks to the data index, where the key is the full path and block number, or truncate a file by deleting key-value pairs corresponding blocks from the data index, and updating the size in the metadata index. Renaming a file is implemented with a range rename operator, which can atomically replace the prefix on a range of keys (and associated values), potentially replacing older keys with the target names [36, 37].

When a file block is written from the VFS to the file system, the block’s contents are copied into the B^ε-tree layer. Each time a block is flushed, it is also copied within memory.

In order to improve small, random reads, leaf nodes are partitioned into multiple **basement nodes** (typically 32, each roughly 128KiB), which are independently compressed and can be read into memory without the rest of the leaf. Within each basement node, data is serialized as a packed list of key-value pairs.

When a node is written to disk, it is first serialized into a large, contiguous buffer, then compressed. Early versions of BetrFS relied on compression to reduce the on-disk storage costs of highly redundant keys, as well as to reduce the total I/O. More recent BetrFS versions use *lifting* [36, 37] to achieve similar space savings, both on disk and in memory, and to reduce the computational costs of in-memory key comparisons. Lifting is essentially a trie-style encoding, where the longest common prefix of all keys in a subtree is removed and stored along with the pivots and child pointer.

Crash Consistency. BetrFS uses a redo log and B^ε-tree checkpoints to ensure crash consistency. The B^ε-tree itself is copy-on-write. After a crash, the redo log is replayed against the newest stable B^ε-tree checkpoint. Between checkpoints, new copies of nodes may be written to disk, but these will be discarded (and reconstructed from the log) if the system crashes before the next checkpoint. The checkpointing process also deletes stale checkpoints and nodes that are not reachable from any active checkpoints. To detect potential corruptions at rest, BetrFS checksums each node on disk, and uses checksums to detect corrupted log entries.

The crash recovery semantics are roughly equivalent to full-data journaling in ext4. More precisely, all operations (including data writes) are effectively written to the log in the order they are received from the VFS. After a crash, the state is consistent with a prefix of the log, up to the last log flush. Note that sync variants induce a log flush, so after any sync, the log up to that point in time is durable.

Range Messages. Important operations, such as renaming or deleting files, require updating potentially many key-value pairs. In the first version of BetrFS, these operations suffered poor scaling due to the sheer volume of messages. Subsequent BetrFS versions introduced a **range message** abstraction that can operate on all key-value pairs within a contiguous key range [35]. For instance, a single range delete message might delete all of the blocks of a single file, rather than requiring a separate deletion message for each individual block. The range message abstraction also unlocks further optimizations.

3 Consolidating Storage Layers

This section describes the ways that BetrFS v0.6 reallocates functionality across the layered Linux storage stack. We first describe several challenges that arise when stacking system components beneath the VFS and then present an efficient design for an in-kernel key-value store. We then discuss optimizations for read-ahead and existence checks in a multi-layer file system. The changes in this section address performance bottlenecks in sequential I/O, scans, small file creation and metadata-intensive workloads (recursive deletion).

Name	SuperBlock	Log	Meta Index	Data Index
Size	8 MB	2 GiB	25 GiB	223 GiB

Table 2. Simplified SFL on-disk layout. The sizes are given for a rough indication of proportion on a 250GiB disk. The SuperBlock region is abstracted as eight logical files.

3.1 The Simple File Layer

To expedite development, BetrFS ported TokuDB [28]—a user-space B^e-tree implementation—to the Linux kernel by implementing the `klibc` “shim” layer. `Klibc` translates many user-level interfaces to Linux kernel APIs, and importantly, it emulates POSIX file interfaces on top of ext4 (See Fig. 1). On an HDD, `klibc` introduces some marginal, but acceptable, overheads that are overshadowed by the dramatic performance gains that the B^e-tree brings to many workloads. On an SSD, however, the overheads of BetrFS’s stacked approach become acute. We describe several of these overheads below.

Cross-layer data copies. By stacking the B^e-tree on ext4, BetrFS’s logical file data is copied to multiple memory locations when it is read; B^e-tree node data is read from disk into ext4’s page cache, node data is copied into the B^e-tree, and finally, logical file data is assembled from B^e-tree messages and copied into the BetrFS page cache. Similar copying occurs on the write path.

Double-journaling. For a concrete example of the double-journaling problem, consider small, synchronous operations, such as those found in a rename-intensive microbenchmark. A `sync()` in BetrFS leads to a small, synchronous write to the BetrFS journal, which leads to a small write followed by a `sync()` in ext4, which then leads to a journal transaction to commit a metadata update in ext4. Double journaling is clearly not a best practice—one journal is sufficient.

Double-buffering delays. The VFS layer employs both time-based and space-based heuristics to manage dirty page write-back. In BetrFS, when the dirty-page threshold reaches a “high water mark” and pages are written, they are written to ext4; different pages in ext4 are then dirtied and scheduled for write-back, leading to no net change in the dirty page count. In total, double buffering between BetrFS and ext4 leads to high variance and “stutters” during sequential I/O workloads.

We observe that most of these stacking-related complexities are unnecessary. The BetrFS B^e-tree implementation does not require a full-fledged file system; rather, it uses 11 fixed files. Two—one per index—are large, `fallocate()`-ed files that store tree nodes; one stores the log; and the other 9 store small amounts of infrequently-changing metadata, such as whether the key-value store was shutdown cleanly or not. Effectively, these files approximate a static disk layout: a region of disk for nodes, a region for the log’s circular buffer,

and a superblock for global metadata. Features such as dynamic block allocation, or even dynamic file creation, are not necessary for BetrFS’s write-optimized key-value store.

Based on this observation, we introduce a storage back-end, called the **Simple File Layer (SFL)**, that gives the abstraction of precisely the 11 files and APIs that the B^e-tree implementation requires, and addresses all four issues mentioned above. Table 2 illustrates the SFL on-disk layout (note that several metadata files are consolidated into a single superblock-style region). For convenience, SFL still provides some measure of POSIX-like APIs, e.g., files. Although SFL is somewhat tailored to BetrFS v0.6’s B^e-tree implementation, a similar file-system shim may be used to efficiently port other user-level code into a kernel.

To eliminate page copying overheads, the SFL I/O interface accepts references to physical page frames, and it can be used synchronously or asynchronously. SFL does not support a file handle with a cursor; rather, all read and write requests supply a file offset and a pre-allocated buffer. Each file occupies a single contiguous extent, and the SFL block size is 4KiB.

SFL eliminates double-journaling by using immutable metadata. Crash consistency for changes to the file system-level data and metadata is already handled by the B^e-tree log; SFL is only responsible for ensuring that synchronous writes are written synchronously.

We modify the B^e-tree code to use a statically allocated disk region as a circular log buffer, and we converted the logging code to use self-validating records with sequence numbers and checksums. As a hint for recovery, we store a recent starting point to search for the range of valid log entries, but this is only updated periodically; the log can be reconstructed by brute force if need be.

Finally, the SFL interface eliminates double buffering, by exporting a direct I/O-style interface where the caller handles all buffer management, including freeing buffers after an I/O completes.

3.2 Read-ahead

In support of efficient sequential reads, we investigated different read-ahead strategies. The VFS has visibility into I/O patterns on specific file handles, and it already has a mechanism to detect sequential accesses. However, VFS-level read-ahead heuristics are not a good fit when nodes on the device are large, as the heuristics only operate on the order of KiB. For a B^e-tree that stores data in 4 MiB nodes, ideally, while processing the current 4 MiB, one should be prefetching the entire next leaf and any additional ancestor nodes, applying messages, and materializing a view of the next leaf. Identifying read-ahead entirely at lower levels is also challenging and error-prone, as unrelated application requests may be interleaved.

BetrFS v0.6 implements a cooperative read-ahead design. We use a standard sequential read heuristic in the northbound layer, which identifies a run of sequential read accesses to a

file. This hint is passed to the B^e -tree layer, which causes the B^e -tree to asynchronously read either the next two basement nodes in the current leaf, or the next leaf node if the query hits a leaf's last basement node.

3.3 Existence Checks and File Creation

We observe a performance pathology in workloads that create a large number of small files. From the perspective of the key-value store, the VFS layer issues series of alternating point queries to the metadata index (to check whether the file exists), followed by inserting a message to create that file's inode. By default, on every query BetrFS implements a heuristic which applies messages *in memory only* to the leaf node of each root-to-leaf path traversed by the query. This effectively creates a materialized view of all pending messages at the leaf, intended to optimize subsequent queries. However, carrying a single message down the B^e -tree on every existence check defeats the purpose of batching these updates.

We address this problem in BetrFS v0.6 by deferring and batching messages that create inodes, with an optimization we call **conditional logging**. We first observe that logging inode creation in the BetrFS v0.6 redo log is sufficient for crash recovery, provided that the inode is inserted into the B^e -tree before that region of the log is reclaimed. Further, as long as the new inode is cached in the VFS in the dirty state, subsequent queries for this inode will be serviced correctly from the VFS cache. Assuming the system does not crash, the inode will eventually be written back to the B^e -tree, likely after accumulating subsequent changes, and after subsequent existence queries have moved onto different root-to-leaf paths.

The main additional coordination mechanism this optimization requires is an additional reference count on sections of the B^e -tree redo log, tracking how many dirty inodes still require this section of log for durability. In the worst case, a dirty inode can delay reuse of a section of the circular log buffer on disk. In practice, however, the period for checkpointing the log is once a minute, which is more than sufficient in practice for dirty inodes to be written to the B^e -tree without obstructing reuse of a portion of the log buffer.

4 Range Message Optimizations

BetrFS implements file removal with a range delete operation. Range deletes are sufficiently fast on HDDs but become a bottleneck when performing recursive deletions on SSDs. For smaller numbers of range messages, recursive deletion performance was comparable to other file systems; the recursive deletion workload examined in this paper is several times larger than prior reported results, and execution time jumps non-linearly from seconds to minutes as the number of deletions scales. This section explains the root cause of these problems, and presents optimizations to address the issues.

BetrFS introduced the PacMan optimization [35] as a way to reduce range delete message overheads: a range delete message may “eat” overlapping messages as the range delete message is flushed down the tree. On SSDs, we find that, in many cases, PacMan actually slows some workloads considerably.

We discovered that the primary reason the PacMan optimization slows down recursive deletion in BetrFS is because, at its core, the VFS is designed for “point” operations, and it bakes assumptions about inode-style indirection into its API. Although BetrFS replaces individual VFS operations with more efficient range operations whenever possible, the VFS infrastructure naively composes these operations.

BetrFS's `rm -rf` behavior does not allow for range delete aggregation. Although BetrFS's B^e -tree locally aggregates range delete messages with overlapping ranges, `rm -rf` enumerates and deletes explicit objects, e.g. it deletes all keys starting with `/dir/a`, then `/dir/c`, etc.. This pattern produces range deletes for logically adjacent, but non-overlapping key ranges. Because BetrFS cannot merge any of these range delete messages, nodes in the B^e -tree fill up with only range messages. When the PacMan optimization runs, it compares every range message in a node to every other message in the node, only to discover that no range deletes allow for any of the other messages to be discarded. So it is a quadratic algorithm that has no effect on the tree, that runs during every node flush. Furthermore, BetrFS must perform flushing to make space in the root node even though, in many cases, all the range delete messages could be logically summarized as a single delete to obviate the I/O costs of flushing.

Range deletion in `rmdir`. Our first optimization is to issue a range delete message in `rmdir`, not just when unlinking files. This seems counter-intuitive and perhaps dangerous, at first, since POSIX requires that a directory be empty before it can be removed. However the purpose of this range delete is not to delete live data; it is to “PacMan” stale messages, including previously deleted key-value pairs from files that were once in that directory and other, disjoint range deletes.

This change alone creates an order-of-magnitude performance improvement over BetrFS in a recursive deletion benchmark, and it does so without modifying the quadratic time algorithm or reducing BetrFS v0.6's I/O activity (the recursive deletion benchmark is too small to realize any reductions in I/O activity from the discarded messages).

To understand this optimization's surprising effectiveness, consider the message sequence produced by a recursive deletion, e.g. `rm -rf`. For each directory deleted, there are a series of range delete messages that correspond to files in that directory followed by a range delete message that corresponds to the directory. The PacMan optimization will traverse these messages in the opposite order that they were issued; in other

words, PacMan will consider a directory's range delete message before it considers any narrower range delete messages for the directory's children. Thus the directory's range delete message will "gobble" the range delete messages for the files in that directory, and so on. Adding directory range deletes last lets PacMan apply more beneficial messages first.

Bypassing B^e -tree queries for empty directories. Rmdir semantics require a directory to be empty, which baseline BetrFS confirms by issuing a B^e -tree query. This query turns out to be a performance bottleneck because, in a WOD, adding a message to the tree—which is how inserts/deletes are implemented—is much faster than performing a query.

In our optimization, we bypass these B^e -tree queries by maintaining correct caches of inode link counts (nlink) in the VFS layer. An `rm -rf` already recursively visits each child directory to identify a list of files to delete. This initial traversal warms the relevant VFS inode caches, making these checks fast. We note that the VFS already maintains this information and expects it to be correct; our primary change was ensuring that these cached values are coherent with the link counts on disk. In summary, maintaining these counters substantially improves performance by avoiding B^e -tree queries.

Redundant messages. The VFS protocol for removing an inode is complex, in order to handle edge cases, such as open handles to unlinked files. We found the lower-level implementation of BetrFS issued two file deletion messages on two different VFS hooks; this was correct, but wasteful. Although adding messages to a write-optimized dictionary is relatively inexpensive, it is not free; removing the extra message further lowered the computational overheads of a recursive deletion.

Opportunistic inode instantiation. The VFS embeds several implementation assumptions into its file system API, including the assumption that a directory's metadata (inode) is stored separately from its data (listing). As a result, the VFS does not cache child metadata during a parent's `readdir`—neither directory entries nor inodes [29]; instead, separate `lookup` calls are needed. In BetrFS, a `readdir` scans contiguous items in the metadata index, and these items contain both the names and the inodes for a directory's children. Thus, the same range query that returns the file names under a directory can also populate the VFS caches with no extra I/O.

We augment BetrFS v0.6 `readdir` to populate child inodes and directory entries in VFS caches. Then, subsequent lookups can avoid the significant overheads of redundant queries to the key-value layer. The power of this optimization is most apparent in a cold cache recursive deletion; because `rm -rf` deletes each entry in a directory tree, bottom-up, all inodes are opportunistically cached during the top-down traversal. This optimization is measured in isolation in the evaluation (Table 3, +DC row), and removes more than one second of latency from recursive deletion (from 3.4s to 2.3s).

Revisiting the apply-on-query optimization. Baseline BetrFS has an optimization heuristic, that, upon a query, pushes certain messages down the B^e -tree and applies them to *in-memory* key-value pairs in cached leaves. We call this heuristic *apply-on-query*. If the cached leaf node is clean, then apply-on-query searches for and applies any pending message that targets a key-value pair in the basement node involved in the query, even if no messages on the root-to-leaf path actually affect the query's result. Note that, in this case, the leaf's in-memory state remains clean, and the B^e -tree's on-disk state is unchanged. The goal of this optimization is to exploit query locality: subsequent queries to the same basement node can quickly check whether an interior node has relevant pending messages, versus a CPU-intensive buffer traversal to identify the relevant messages.

If the cached leaf node is dirty, then apply-on-query *flushes* pending messages that target any key-value pair in the entire leaf. We use the term "flush" when a message is applied to a dirty in-memory node and, ultimately, to the on-disk B^e -tree. Apply-on-query was designed for HDDs, where it is worthwhile to spend tremendous amounts of computation to save even one additional I/O. By aggressively flushing pending messages, apply-on-query may postpone future writes to this leaf by keeping interior buffers as empty as possible.

For a workload like recursive deletion, which alternates range deletions and point queries (existence checks) and moves through the keyspace sequentially, this heuristic causes an interesting pathology when the leaf node is clean. Suppose that the first deletion adds a range delete at the "left" of the keyspace that maps onto a clean basement node. A subsequent query to the next, disjoint key (for the next file to delete) will trigger an apply-on-query for the prior range delete, even though that range delete is non-overlapping. This process continues across the entire keyspace covered by the leaf node.

In such a workload, one incurs all of the costs of apply-on-query, and gets none of the locality benefits, as the deleted keys will not be queried again. Moreover, one must still recheck all ancestor nodes on the root-to-leaf path for relevant messages to apply. We note that checking range messages are more expensive than point messages, both because there are two key comparisons (versus one), and the indexing structure is more complex to handle partially overlapping ranges. So a range-delete intensive workload exacerbates these costs. On an SSD, with a tighter computational budget per IO, this optimization uses sufficient CPU time that it significantly increases latency of a range-delete-intensive application, with potentially marginal benefits in terms of total IOs.

In BetrFS v0.6, we introduce a new apply-on-query policy: we only flush or apply pending messages if one or more pending messages affect the outcome of the query. Said differently, if at least one pending message in an interior node targets a key-value pair in the query, BetrFS v0.6 does apply-on-query as before; otherwise, the in-memory tree state is left unchanged. The key takeaway is that faster devices necessitate

revisiting the relative weights of I/O and CPU amplification. The CPU cost of identifying all pending messages for an entire basement or leaf node is quite large, but, at least for common file system workloads, the resulting I/O savings is not. BetrFS v0.6 must already find and apply pending messages that affect the query, and this information is often a sufficient signal to exploit query locality.

Even with all other optimizations in this paper applied to the code base except for this one, this optimization alone further reduces recursive deletion latency by roughly one second (from 2.56s to 1.57s). We show the performance impact of this optimization in Table 3 in the +QRY row.

5 Memory Allocator Optimizations

This section describes optimizations to the management of large node buffers in the kernel. These overheads manifest primarily in workloads that issue many small writes, such as the random write microbenchmark and TokuBench.

Memory management was a critical bottleneck when moving BetrFS from a hard drive to an SSD, largely because buffer sizes are difficult to predict when managing B^E-tree messages in memory. For instance, flushing from a parent to a child may cause a recursive flush, and the child may need to temporarily allocate a larger in-memory buffer than it will ultimately need on disk.

Although memory allocation is a well-studied problem, there is a subtle mismatch in interfaces and techniques. Most Linux allocators assume objects will be standard sizes, most I/Os will be small (order of pages), and that large, irregular buffers are the exceptional case. Specifically, the only way to reliably allocate buffers on the order of megabytes in the kernel is with `vmalloc`, which is relatively expensive, as `vmalloc` changes the kernel's memory mapping on *every* CPU. We note that `kmalloc` supports larger allocations in a best-effort manner, but, in practice, they quickly fail once physically contiguous pages in the buddy allocator are exhausted; for any non-trivial workload, BetrFS will quickly exhaust physical contiguity, but BetrFS only requires virtual contiguity.

In contrast, the B^E-tree implementation used in BetrFS was developed in user-space, using standard `malloc`, `realloc`, and `free` interfaces. In particular, most of this code was written with the assumption that `realloc` is an efficient way to dynamically grow a buffer that is hard to predict a priori. A key *implementation* assumption here is that most allocators have an efficient memoization of both allocated sizes and can often identify and reclaim fragmented space, say from rounding up to a power of two, on a `realloc` call. In Linux, however, `vmalloc` can only identify the size of an object by an expensive search of the kernel's memory mappings, and there is not an efficient way to incrementally grow a `vmalloc`'ed buffer. To be clear, these assumptions are true for small objects, using `kmalloc`; the challenge for BetrFS is efficient management of large, variable-sized objects in the kernel.

The overheads of memory management became critical for workloads that dealt with a large volume of small messages, such as TokuBench, random writes, and recursive deletion—accounting for at least 10% of execution time for each one.

We address these issues in several ways. First, we observe that most B^E-tree code that uses large, dynamically sized buffers already tracks its own used and free space. We modify the internal `free` and `realloc` interfaces to pass this information back to our “southbound” allocator (or wrapper for `kmalloc` and `vmalloc`). This elides the expensive searches for `vmalloc` sizes in the kernel code, or the need to add our own memoization, when freeing a `vmalloc`'ed region.

Second, and because memory management is critical to overall performance of the code, a number of side caches and allocators had developed over time as point fixes for specific bottlenecks. However, this also led to an increase in complexity and important information being lost by the time an allocation reaches the lowest level of the file system; thus, part of this work involved removing intermediate caches and memory pools, and streamlining one efficient allocator. Although baseline BetrFS had a small cache of 32 128K `vmalloc`'ed regions, targeting one common size, we found this was insufficient in practice, and expanded this cache of larger buffers to include additional, common powers of two.

Third, we introduce a cooperative memory management interface, where the B^E-tree and supporting kernel code can negotiate buffer sizes that can be allocated efficiently. Previously, B^E-tree code selected seemingly arbitrary buffer sizes, and grew them as they overflowed—typically by doubling the size. We augment our internal allocator to also return the available space at allocation time (similar to user-space `malloc_size`), and adapt code sites that rely heavily on `realloc`, often followed by relatively expensive re-initialization, to always leverage the full buffer amount at allocation time.

Unlike `malloc_size`, our interface can deliberately return much more space than requested. Based on profiling the behavior of the code, we found a number of sites where the final size of a buffer is bi-modal: it is either relatively small, or will grow to be megabytes in size. We found that by simply avoiding incremental powers-of-two, we could get buffers to their expected sizes quickly, and elide additional copies.

6 Sharing Pages between the VFS and B^E-tree

This section describes how BetrFS v0.6 shares pages of file data in the VFS page cache with the B^E-tree internal structures—realizing zero-copy I/O between the VFS and the disk, in the absence of updates to the same page during write back. These optimizations are introduced to improve the performance of large, sequential reads and writes.

By default, Linux file systems place written data in the in-kernel page cache, in the “dirty” state. A dirty page may remain in the page cache for a short period, in anticipation that an application may issue additional writes to the same

page. Dirty pages are eventually written to the underlying file system; during write back, the pages are locked and may not be updated by applications until the write completes, preventing on-disk consistency problems.

In the case of BetrFS, however, these writes require copies (in-memory) when adding a message to the tree. First, data is copied from the VFS page cache into the B^e -tree (in memory). A B^e -tree is designed to buffer changes for a longer period, to amortize the costs of writes into a larger node. Keeping pages in the page cache in a locked state for longer periods may disrupt subsequent application writes. Second, data may be flushed from the root node to children, or even down to the leaf. For both simplicity, and with a focus on smaller values and slower disks, values (even 4KiB pages) are copied in memory on each flush. We note that there are already optimizations in BetrFS to avoid writing every flush *to disk* in cases where a number of writes are following the same root-to-leaf path; however, the complete data is always `memcpy`-ed at each level. Because hard disks are so slow relative to memory, this computational overhead had minimal impact on BetrFS performance on a HDD; on an SSD, this overhead becomes significant. Finally, there is a copy at the final write to the underlying file system. We note that SFL (§3) is a necessary building block to eliminate this final copy.

In BetrFS zero copy is less straightforward, as the B^e -tree implementation may track multiple versions of a given key-value pair in memory, in order to optimize the I/O behavior. For both crash consistency and performance, BetrFS has several additional requirements:

- The same file block may exist in multiple versions, over a potentially long time. If a 4KiB page is rewritten a new copy is inserted into the tree, but multiple, older versions may exist in interior B^e -tree nodes, both in memory and on disk. Once a version of a file block is in the B^e -tree, it is not modified again by the B^e -tree implementation. Modifying a page frame that is added to the B^e -tree is equivalent to editing the history of updates.
- One does not know at the start of write-back where on disk a page will be placed. This is both because of potentially flushing messages between the start of write-back and a forcing condition (such as an `fsync`), and the fact that on disk, B^e -tree nodes are copy-on-write.

The high-level strategy in BetrFS v0.6 is to move pages through the tree by reference to the physical frame number while in memory, rather than by value. This ensures that there is, at most, one copy per write to a page. Although a seemingly simple change, this introduces some design challenges.

Tracking Messages. We add a new message type to the B^e -tree implementation for insert messages, called `insertByRef(key, reference, derefMem())`, which accepts a key, an opaque reference to a value, and a function to convert the reference back to a value. In the case of BetrFS

v0.6, we use page frame numbers as our reference, and a function such as `kmap()` to copy memory contents, if needed.

VFS-level Copy-on-Write. When a page is inserted to the B^e -tree using the `insertByRef` message, we need to ensure that page contents do not change again until any references to the page are released by the B^e -tree, as the B^e -tree expects messages to be immutable until they are obviated.

In order to avoid obstructing subsequent writes, we modify the “northbound” code that interacts with the VFS layer to change write-back to copy-on-write using the `PG_private` flag and file-system level hooks. The B^e -tree implementation will not write this page; if a VFS operation (e.g., a `write()` system call or a write via `mmap`) occurs, the VFS will call into a lower-level function which first checks the private fields, giving BetrFS v0.6 the opportunity to allocate a new page to accept the write. In the case where the B^e -tree releases all references to the page before a subsequent write, the copy can be elided and the existing VFS page can accept the write.

On-disk Node Format. Zero-copy I/O is not possible in the BetrFS node layout. Within a basement node, key-value pairs are packed in a series of `key1, value1, key2, value2`, etc. We modify the basement node (sub-leaf node, §2) layout to first pack keys and any small values at the front of the basement node, and then place all 4KiB values in aligned sectors at the end of the basement node. With this layout, when a node is read into a buffer, all file blocks in a typical file will already be placed in 4KiB-aligned buffers in memory.

In the case of writing a node, this also reduces the serialization cost when combined with scatter-gather style I/O. The keys and small values must still be serialized, but a list of page reference can be directly passed to SFL.

Interior B^e -tree nodes follow a similar design, with small messages packed at the front of the node, and page-sized values in aligned locations at the end of the node.

Reads and Caching. When a node is read into memory in BetrFS v0.6, we allocate a virtually contiguous region of kernel memory to map the node if needed, but do not necessarily require the pages in that region to be physically contiguous. Node contents, including data blocks, can be cached, copy-on-write, by both the B^e -tree internally, and in the page cache. Each layer may also free memory independently. For instance, suppose a single page is shared between the VFS and a larger B^e -tree node; here, the B^e -tree node can be freed, except for the one page, which becomes exclusively owned by the VFS.

7 Evaluation

In this section, we evaluate BetrFS v0.6 performance using a combination of microbenchmarks and application workloads. Our evaluation seeks to answer the following questions:

- To what degree does each optimization described in the paper improve performance?

File System	Sequential I/O				Random Writes			Utility Latency		
	Read(80g)		Write(80g)		4 KB	4 Byte	Tokubench	grep	rm	find
Ext4	534	(1)	322	(8)	22 (2)	0.036 (0)	33.7 (7.5)	5.68 (0.14)	2.30 (1.26)	0.47 (0.04)
Btrfs	568	(1)	333	(0)	15 (2)	0.028 (0)	11.6 (0.2)	4.39 (0.41)	2.28 (0.02)	0.57 (0.05)
Xfs	534	(2)	320	(13)	26 (0)	0.036 (0)	7.2 (0.1)	4.44 (0.04)	2.52 (0.03)	0.59 (0.01)
f2fs	529	(1)	320	(5)	23 (3)	0.046 (0)	0.0 (0.0)	3.55 (0.14)	2.22 (0.02)	0.86 (0.17)
zfs	548	(0)	312	(2)	8 (0)	0.008 (0)	11.4 (0.1)	1.19 (0.02)	3.11 (0.54)	0.46 (0.02)
Betrfs	128	(1)	33	(0)	87 (1)	0.265 (0)	9.9 (1.3)	2.38 (0.16)	40.66 (1.15)	0.36 (0.04)
FestiFS+SFL	462	(0)	222	(1)	96 (2)	0.262 (0)	5.4 (0.1)	1.44 (0.07)	44.71 (4.58)	0.19 (0.00)
+RG	462	(1)	226	(0)	97 (1)	0.274 (0)	5.3 (0.1)	1.44 (0.06)	5.02 (0.31)	0.21 (0.01)
+MLC	463	(0)	226	(1)	115 (2)	0.352 (0)	8.3 (0.1)	1.44 (0.02)	4.21 (0.13)	0.24 (0.04)
+PGSH	497	(0)	310	(3)	118 (1)	0.360 (0)	7.7 (0.1)	1.46 (0.06)	3.41 (0.10)	0.20 (0.00)
+DC	496	(1)	312	(1)	116 (2)	0.358 (0)	7.8 (0.1)	1.33 (0.02)	2.30 (0.10)	0.20 (0.01)
+CL	497	(1)	306	(1)	118 (1)	0.364 (0)	11.7 (0.2)	1.42 (0.08)	2.56 (0.06)	0.22 (0.00)
+QRY	497	(0)	310	(3)	116 (2)	0.363 (0)	11.8 (0.2)	1.36 (0.02)	1.57 (0.04)	0.22 (0.01)

Table 3. Throughput in MB/s (left) and latency in seconds (right) of various file systems on a commodity SSD. Higher throughput and lower latency is better. Standard deviation is in parentheses. Btrfs and BetrFS v0.6 have compression disabled. Each optimization in BetrFS v0.6 is added with one row with the name of "+" followed by the abbreviation of this optimization in the table. "SFL" is short for Simple File Layer; "RG" range; "MLC" malloc; "PGSH" page sharing; "DC" dentry cache; "CL" conditional logging; "QRY" query path optimization. For instance, "+RG" adds the range message optimizations to BetrFS v0.6 with SFL. Any data points within 15% of the highest throughput or lowest latency are highlighted in green; those that are less than 30% of the best throughput (or more than $3.33\times$ the best latency) are highlighted in red.

- Do these optimizations translate into application-level performance improvements?

All experiments were conducted on a PowerEdge T130 with a 4-core 3.00 GHz Intel Xeon E3-1220 v6 CPU, and 32 GB RAM. The system runs 64-bit Ubuntu 18.04.6 with Linux kernel version 4.19.99 when we test BetrFS and BetrFS v0.6 variants and 5.9.15 for other file systems, in order to give the competition the advantage of any recent advances. We boot with the root file system on a 500 GB TOSHIBA DT01ACA0 HDD. The SSD under test is a 250 GB Samsung EVO 860 SSD with a 512-byte page size and 12 GB write cache; we measure a peak raw sequential bandwidth at 567 MB/s for reads. For writes, the peak bandwidth is 502 MB/s when the data size is smaller than 12 GB and drops to 392 MB/s when the data size is larger than 12 GB, which we attribute to device-internal operations.

We compare BetrFS v0.6 with several general-purpose file systems, including BetrFS, btrfs, f2fs, ext4, xfs, and zfs. We mount these file systems with default parameters, unless otherwise specified. We use the versions of xfs, btrfs, ext4, and f2fs that are part of the Linux 5.9.15 kernel, and zfs version 0.8.6 from zfsenlinux.org.

In terms of crash consistency, ext4 is configured with ordered mode; xfs and btrfs have similar semantics. BetrFS, BetrFS v0.6, ZFS, and F2FS provide guarantees comparable to full data journaling. BetrFS, BetrFS v0.6, btrfs, and ZFS also include checksumming to detect on-disk corruptions.

7.1 File System Microbenchmarks

We first conduct a series of file system microbenchmarks to understand how each optimization contributes to BetrFS v0.6 performance. Since BetrFS v0.6 augments the BetrFS design

with a series of optimizations, we evaluate BetrFS v0.6 performance with an increasing number of these optimizations "turned on".

In the sequential I/O tests, we used fio to sequentially write a single 80GiB file, and then re-read it after clearing the VFS caches, reporting average throughput and standard deviation. In the random write tests, we wrote to 256K randomly selected, block-aligned offsets within a 10GiB file, followed by a single `fsync()`. The first column shows the performance of random writes at a 4KiB granularity; the second column shows writes at 4 byte granularity. The TokuBench [7] benchmark creates three million 200-byte files in a balanced directory tree with a fanout of 128. We run TokuBench with 4 threads since our machine has 4 cores. For `rm`, we recursively delete a directory with 2 copies of Linux 3.11.10 source code. For `grep`, we search for the string "cpu_to_be64" in the directory of Linux source code. For `find`, we search for the files with the name `wait.c` in the directory of Linux source code.

Table 3 shows the file system throughput for sequential reads, sequential writes, random writes, and TokuBench, as well as latency of `grep`, recursive deletion, and `find` on an SSD. We note that our goal, as stated in §1, is, ideally, to build a single file system that is within 15% of the top performing file system for each operation (shaded in green). Cells shaded in red are more than 70% worse than the best file system.

Simple File Layer (SFL). We find that the impact of switching the "southbound" file system from ext4 to SFL (§3) significantly improved sequential read (by $2.5\times$ and within 20% of disk bandwidth) and reduces `grep` latency by 41%, as the read-ahead heuristics in the B^e-tree were more effective than the prior read-ahead behavior. Replacing ext4 with SFL

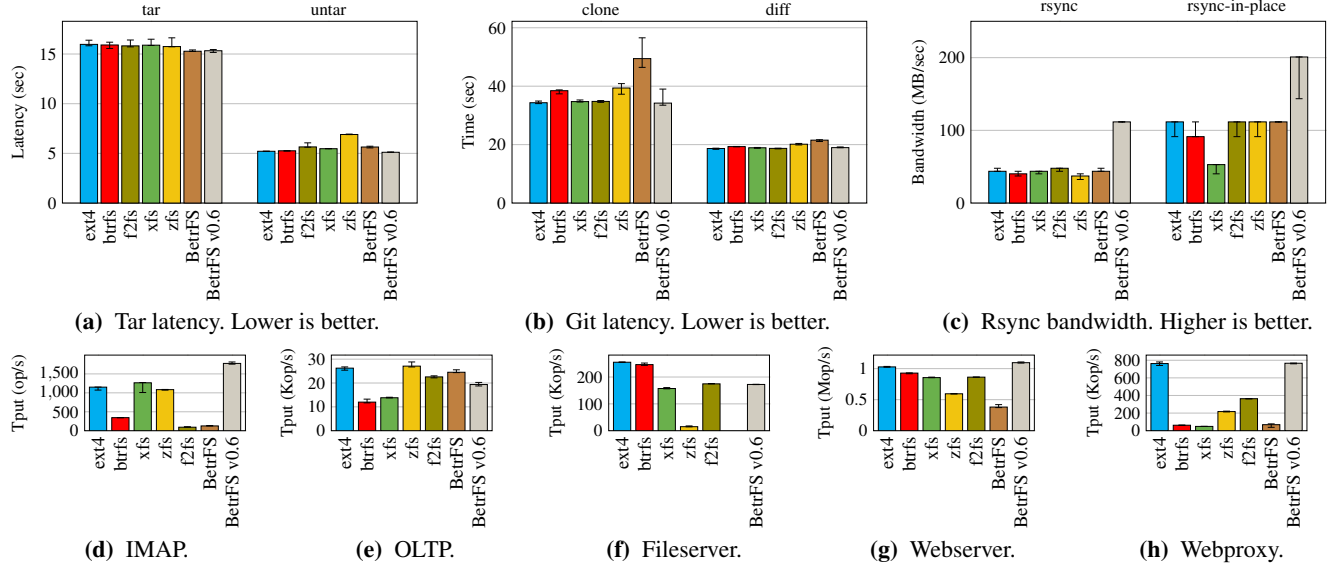


Figure 2. Application benchmarks. Figures 2c–2h are all throughput, and higher is better.

also removes double journaling and double caching, which yields significant improvements to several write-intensive workloads, including sequential write (+304% throughput), TokuBench (+35%), and recursive deletion (−13% latency).

Range (RG) and Query Path (QRY) Optimizations. As expected, the main beneficiary of the range optimizations is recursive deletion (§4), which is dominated by range deletion messages. These optimizations reduce the r_m time by an order of magnitude. Reworking the apply-on-query heuristic (also §4) brings BetrFS v0.6 to be the fastest file system on this microbenchmark, and marginally reduces `grep` time and increases sequential write throughput.

Memory Allocation (MLC). The memory allocation optimizations (§5) primarily benefit workloads that issue many small messages, such as random writes (19% and 28% increased throughput for 4KiB and 4B, respectively), TokuBench (57%), and recursive deletion (−16% latency). For these workloads, the overheads of managing large buffers of small objects become a first-order cost.

Page Sharing (PGSH). Page sharing (§6) primarily improves sequential IO performance, although it helps other update intensive workloads. Sequential read improves by over 30 MBps, and sequential write improves by nearly 100 MBps, making sequential writes competitive with other file systems. With an 80 GiB sequential write (needed to more than double DRAM size), the SSD appears bottlenecked on internal operations, well below the advertised peak bandwidth. This is independent of the file system under test. On shorter writes (e.g., 10 GiB, omitted for brevity), this optimization more than doubles the throughput compared to no page sharing.

Directory Cache (DC) and Conditional Logging (CL). We present the directory cache optimization (§4) for read-ahead separately; it removes more than an second of execution time from a recursive deletion. Similarly, the conditional logging optimization (§3.3) increases small file creation (TokuBench) throughput by 50%.

In total, each of these optimizations are necessary to meet our performance goal. Most notably, these optimizations further improve random write and search performance over both baseline BetrFS and other standard Linux file systems, indicating that relying solely on faster device-level random write behavior without corresponding data structural changes leaves over $6\times$ the throughput on the table.

7.2 Applications and Benchmarks

We measure end-to-end performance of BetrFS v0.6 on a range of applications and FileBench [27] server-side workloads, running on an SSD.

Rsync copies the Linux 3.11.10 source tree from a source to a destination directory within the same partition and file system. With the `−in-place` option, `rsync` writes data directly to the destination file rather than creating a temporary file and updating via atomic rename. **Tar** unpacks a tarball of Linux 3.11.10 source code and **untar** creates a tarball out of a copy of Linux 3.11.10 source code. **Git clone** clones the latest Linux source repository from one directory to another on the local system. **Git diff** workload reports the time to diff between the v4.14 and v4.07 Linux source tags. The Dovecot **mailserver** 2.2.13 is run with 8 threads. We initialize the mailserver with 10 folders, each containing 2,500 messages. Each of 8 clients performs 10,000 operations with 50% reads and 50% updates (marks, moves, or deletes).

The results are presented in Figures 2a–2d. In the best case, throughput of an in-place `rsync` is nearly double the

competition, including baseline BetrFS. Similarly, in IMAP, BetrFS v0.6 is much faster than any other file system, whereas BetrFS is the slowest. Among these workloads, BetrFS v0.6 is able to roughly match the fastest file system.

Filebench. Figures 2e–2h show the performance for four FileBench benchmarks: OLTP, WebServer, WebProxy, and FileServer. Note that baseline BetrFS crashes on FileServer.

BetrFS v0.6 performs as well as the best file systems for Webserver and Webproxy, whereas for OLTP and Fileserver the performance of BetrFS v0.6 is in the middle. In the case of OLTP, the primary bottleneck in BetrFS v0.6 is heavy use of `fsync` system calls, combined with some code paths for log writing that are synchronous in BetrFS v0.6 but asynchronous in other file systems. Profiling indicates that the lost relative performance on FileServer is a mixture of smaller overheads for sequential writes (commensurate with Table 3) and smaller, random reads. We were unable to remove these bottlenecks at submission time, but will continue this implementation effort.

8 Related Work

File Systems for SSDs. The flash era has intensified interest in log-structured file systems [24], largely due to flash’s faster random reads and the observation that flash translation layers perform best with sequential writes. SFS [18] is a log-structured file system based on NILFS2 (a log-structured file system tuned for HDDs). SFS co-locates data blocks with similar update likelihoods to minimize segment cleaning overheads. FlashLight [10] is a log-structured file system that introduces a hybrid indexing scheme, intra-inode index logging, and a GC scheme that adopts fine-grained data separation. These techniques reduce indexing overheads. F2FS [13] uses a flash-friendly on-disk layout, cost-effective indexing structures, and both multi-head and adaptive logging to optimize log-structured file system performance for modern SSDs.

Orthogonal to this work, some file systems exploit internal SSD parallelism. For instance, ParaFS [38] proposes 2-D data allocation as a way to maintain hot/cold data separation while exploiting channel-level parallelism.

More recently, there has been considerable interest in tiered storage architectures, often using byte-addressable non-volatile memory to first ingest data that is later migrated to capacity-optimized SSDs (or HDDs). NVMFS [20] is one such experimental file system that assumes two distinct types of storage media: NVRAM and NAND flash SSD. The fast NVRAM is used to store hot data and metadata, and SSD writes are sequentialized as in a log-structured file system. Strata [11] is a multi-tiered user-space file system that exploits NVMM as the high-performance tier, and SSDs/HDDs as the lower tiers. Nova [34] is a file system designed for hybrid memory systems, using log-structured file system techniques to exploit the fast random access that NVMs provide.

Anvil [33] exposes fine-grained control over storage address remapping within an SSD, facilitating efficient snapshots, deduplication, and single-write journaling. Transactional Flash [19] extends SSDs with a transactional API, supporting transactional operations in file and database systems.

BlueStore [1] replaces local file systems as the Ceph storage backend and addresses similar metadata performance issues caused by stacking file systems.

Optimizing WODs for Flash Storage. LSM-trees are a popular write-optimized dictionary (WOD), used in popular key-value stores, including RocksDB. RocksDB [16] recently implemented a range delete operation that is similar to the operation optimized in this paper. Several projects have looked at optimizations for running an LSM-tree on flash. bLSM [25], as a general-purpose log-structured merge tree for both HDD and SSD, introducing a new merge scheduler to minimize write latency and maintain write throughput; it also uses Bloom filters [4] to improve performance. VT-Trees [26] use indirection to avoid unnecessarily rewriting already-sorted data during compactions. LOCS [32] exposes internal flash channels to the LSM-tree key-value store so that compactions can exploit the abundant parallelism. Wiskey [15] separates keys from values in a persistent LSM-tree-based KV-store to minimize I/O amplification for SSD-conscious storage. PebblesDB [21] fragments interior levels of the LSM tree in order to reduce the overheads of compaction. SILK [2] is a key-value store, based on RocksDB, that incorporates the notion of an I/O scheduler to reduce interference and thus prevent latency spikes. KVell [14] presents a novel persistent key-value design for modern block-addressable NVMe SSDs; it adopts shared-nothing philosophy to avoid synchronization overhead, batches device accesses, and maintains an inexpensive partial sort in memory. All of these techniques are complementary to the techniques described in this paper.

VFS Directory Cache Optimization. Tsai et al. [30] propose caching the result of a `readdir`, which Linux does not do by default. Our `readdir` optimization is complementary: it does not reduce the number of `readdir` calls, but uses these results to reduce subsequent child lookup costs.

Write-Optimized File Systems. TableFS [22] and IndexFS [23] store file system metadata in an LSM-tree with a column-style schema to speed up insertion throughput. TokuFS [7] is a FUSE-based file system, also built using B⁺-trees. TokuFS showed that a write-optimized file system can support efficient write-intensive and scan-intensive workloads. KVFS [26] uses VT-trees. It is also FUSE-based, supports transactions, and has comparable performance to the in-kernel ext4 file system.

9 Conclusion

Although trade-offs are often considered fundamental in file system design, this paper demonstrates that one can build a complete file system for commodity SSDs. Further, this paper

demonstrates that the choice of persistent indexing data structure has a first-order impact on performance, even on faster flash devices—legacy data structures forego up to $6\times$ the random write throughput that one can realize with a B^e -tree on the same hardware. This is built on the basic technique of batching smaller I/Os into larger ones, but requires considerable infrastructure improvements and other optimizations to meet the computational budget required by flash storage.

References

- [1] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2019. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) (SOSP '19). Association for Computing Machinery, New York, NY, USA, 353–369. <https://doi.org/10.1145/3341301.3359656>
- [2] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. <https://www.usenix.org/conference/atc19/presentation/balmau>
- [3] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to B^e -Trees and Write-Optimization. *login: magazine* 40, 5 (October 2015), 22–28.
- [4] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.
- [5] 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. 2017. File Systems Fated for Senescence? Nonsense, Says Science!. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*. 45–58.
- [6] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald Porter, and Martin Farach-Colton. 2019. Filesystem Aging: It's more Usage than Fullness. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association, Renton, WA. <https://www.usenix.org/conference/hotstorage19/presentation/conway>
- [7] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2012. The TokuFS Streaming File System. In *4th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'12, Boston, MA, USA, June 13-14, 2012*. <https://www.usenix.org/conference/hotstorage12/workshop-program/presentation/esmet>
- [8] Brendan Gregg. 2008. ZFS L2ARC. <http://dtrace.org/blogs/brendan/2008/07/22/zfs-l2arc/>
- [9] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A Right-Optimized Write-Optimized File System. In *Proc. 13th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, CA, USA, 301–315.
- [10] Jaegeuk Kim, Hyotaek Shim, Seon-Yeong Park, Seungryoul Maeng, and Jin-Soo Kim. 2012. FlashLight: A Lightweight Flash File System for Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 11S, 1, Article 18 (June 2012), 23 pages. <https://doi.org/10.1145/2180887.2180895>
- [11] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 460–477.
- [12] Michael Larabel. 2018. Linux 4.16 File-System HDD and SSD Tests With EXT4/F2FS/Btrfs/XFS. <http://www.phoronix.com/vr.php?view=26157>. Last Accessed May. 21, 2020.
- [13] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 273–286.
- [14] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) (SOSP '19). Association for Computing Machinery, New York, NY, USA, 447–461. <https://doi.org/10.1145/3341301.3359628>
- [15] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Wiskey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.
- [16] Abhishek Madan and Andrew Kryczka. [n. d.]. DeleteRange: A New Native RocksDB Operation. <https://rocksdb.org/blog/2018/11/21/delete-range.html>. Accessed: 2020-06-05.
- [17] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. 2021. High Velocity Kernel File Systems with Bento. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 65–79. <https://www.usenix.org/conference/fast21/presentation/miller>
- [18] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (San Jose, CA) (FAST'12). USENIX Association, USA, 12.
- [19] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. 2008. Transactional Flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) (OSDI'08). USENIX Association, USA, 147–160.
- [20] Sheng Qiu and AL Narasimha Reddy. 2013. NVMFs: A hybrid file system for improving random write in nand-flash SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–5.
- [21] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) (SOSP '17). Association for Computing Machinery, New York, NY, USA, 497–514. <https://doi.org/10.1145/3132747.3132765>
- [22] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Presented as part of the 2013 USENIX Annual Technical Conference (ATC 13)*. 145–156.
- [23] K. Ren, Q. Zheng, S. Patil, and G. Gibson. 2014. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 237–248.
- [24] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems* 10, 1 (Feb. 1992), 26–52.
- [25] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. Scottsdale, AZ, USA, 217–228. <https://doi.org/10.1145/2213836.2213862>
- [26] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building Workload-Independent Storage with VT-Trees. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*.

- USENIX, San Jose, CA, 17–30. <https://www.usenix.org/conference/fast13/technical-sessions/presentation/shetty>
- [27] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *login Usenix Mag.* 41, 1 (2016). <https://www.usenix.org/publications/login/spring2016/tarasov>
 - [28] Tokudb. [n.d.]. <https://github.com/percona/PerconaFT>, Last Accessed Sep. 24 2018..
 - [29] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. 2015. How to Get More Value from your File System Directory Cache. In *SOSP*. 441–456.
 - [30] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E Porter. 2015. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 441–456.
 - [31] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 59–72. <https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor>
 - [32] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems (Amsterdam, The Netherlands) (EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 16, 14 pages. <https://doi.org/10.1145/2592798.2592804>
 - [33] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2015. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 111–118. <https://www.usenix.org/conference/fast15/technical-sessions/presentation/weiss>
 - [34] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 323–338.
 - [35] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2016. Optimizing Every Operation in a Write-optimized File System. In *Proc. 14th USENIX Conference on File and Storage Technologies (FAST)*. 1–14.
 - [36] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. 2018. The Full Path to Full-Path Indexing. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 123–138. <https://www.usenix.org/conference/fast18/presentation/zhan>
 - [37] Yang Zhan, Alexander Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E Porter, and Jun Yuan. 2020. How to Copy Files. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 75–89.
 - [38] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. 2016. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In *2016 USENIX Annual Technical Conference (ATC 16)*. USENIX Association, Denver, CO, 87–100. <https://www.usenix.org/conference/atc16/technical-sessions/presentation/zhang>

A Artifact Appendix

A.1 Abstract

This artifact contains a set of scripts that reproduce the experiments in the Eurosys 2022 paper “A Compleat File System for Commodity SSDs”, which introduces BetrFS version 0.6. BetrFS is an in-kernel Linux file system that uses a B⁺-tree to index on-disk data and BetrFS v0.6 performs consistently well on an SSD across different workloads.

A.2 Description & Requirements

A.2.1 How to access. The artifact consists of two public GitHub repositories. The first repository, <https://github.com/oscarlab/betrfs>, contains all public releases of the BetrFS source code. BetrFS version 0.6, which is the version of BetrFS where all of the paper’s optimizations applied, is denoted by the tag *v0.6*. Versions of BetrFS with the paper’s optimizations cumulatively applied can be found in this repository’s *eurosys22/sfl-kernel-4.19.99*, *eurosys22/range-kernel-4.19.99*, *eurosys22/malloc-opts-kernel-4.19.99*, *eurosys22/page-sharing*, *eurosys22/dc-kernel-4.19.99*, *eurosys22/cond-log-opt*, and *eurosys22/query-path-opt* branches.

The second repository, <https://github.com/oscarlab/betrfs-eurosys22-artifact>, contains scripts that automate the process of downloading, compiling, and installing dependencies; switching Linux kernels; running experiments; and aggregating experiment results. The *README.md* file in this repository contains detailed instructions, including Unix commands, that run the scripts provided to complete these tasks.

A.2.2 Hardware dependencies. Since the paper compares Linux file system performance on commodity SATA SSDs, the testing environment should have an independent SATA SSD that is sufficiently large, i.e., with least 250GiB of usable capacity, as well as a root file system partition with at least 250GiB of space.

A.2.3 Software dependencies. The artifact is known to work on the stock version of Ubuntu 18.04.6.

BetrFS v0.4, BetrFS v0.6, and the set of comparison file systems use different kernels. In particular,

- BetrFS v0.4: stock Linux v4.19.99
- BetrFS v0.6: patched Linux v4.19.99
- All other file systems: stock Linux v5.9.15 (the newest Linux kernel version at the time of development).

The artifact provides scripts to build and switch among the required kernels. The artifact also provides scripts to install required dependencies from Ubuntu repositories or public web sources, including git, bison, flex, gcc-7, g++7, valgrind, zlib, f2fs-tools, ZFS, the Dovecot mailserver, and Filebench.

A.2.4 Benchmarks. The Filebench benchmarking tool is used in this evaluation. The *betrfs* repository contains all

required Filebench workload personalities used in the experiments.

A.3 Set-up

To prepare the environment, the user should run the *install-deps.sh* script in the *betrfs-eurosys22-artifact* repository.

In addition, the user should confirm that the *install-deps.sh* script set meaningful values for the configuration variables in the *betrfs* repository’s *benchmarks/fs-info.sh* file. More importantly, the user should set *sb_dev* in *benchmarks/fs-info.sh* to the *devfs* path of the SSD on which experiments will be run.

A.4 Evaluation workflow

To compare performance of the different file systems and optimizations across workloads, the experiments should be run in an appropriate environment for each file system. There are scripts in the *betrfs-eurosys22-artifact* repository to complete these general steps:

- install and reboot into the appropriate kernel
- run the comprehensive evaluation script

The *pre-*.sh* scripts install the required kernel and any other dependencies, after which the machine must be rebooted.

The *eval-*.sh* scripts run the actual benchmarks. Results of the experiments are placed into CSV files in a directory named *results/*.

If desired, the *betrfs-eurosys22-artifact* repository also includes instructions for running individual tests from among the set of tests that are part of the comprehensive evaluation script.

A.4.1 Major Claims.

- (C1): On commodity SATA SSDs, BetrFS v0.6 performs consistently well across a range of workloads that test file system performance. This is proven by microbenchmarks in experiment set (E1).
- (C2): BetrFS v0.6’s consistently strong performance on common file system tasks corresponds to good performance on applications. This is proven by the experiments (E2) through (E6).

A.4.2 Experiments.

- *Experiment (E1): [FS microbenchmarks]*: file system microbenchmarks correspond to Table 3. Microbenchmarks include sequentially reading and writing large files, random writes, creating many small files in a balanced directory tree (Tokubench), recursive directory traversal (find, grep), and recursive directory deletion (rm).
- *Experiment (E2): [tar/untar]*: the time to create and expand tar archives of the Linux 3.11.10 source code, corresponding to Figure 2(a).

- *Experiment (E3): [git]*: Git latency, corresponding to Figure 2(b). Git clone reports the time to clone the latest Linux source repository from one directory to another on the local system, and Git diff reports the time to diff between the v4.14 and v4.07 Linux source tags.
- *Experiment (E4): [rsync]*: rsync throughput, corresponding to Figure 2(c). Rsync copies the Linux 3.11.10 source tree from a source to a destination directory within the same partition and file system. With the `-in-place` option, rsync writes data directly to the destination file rather than creating a temporary file and updating via atomic rename.
- *Experiment (E5): [IMAP]*: mailserver throughput, corresponding to Figure 2(d). The Dovecot mailserver 2.2.13 is run with 8 threads. The mailserver is initialized with 10 folders, each containing 2,500 messages. Each of 8 clients performs 10,000 operations with 50% reads and 50% updates (marks, moves, or deletes).
- *Experiment (E6): [Filebench]*: Filebench performance on a set of different simulated workloads, corresponding to Figures 2(e) through 2(h). The workloads include scaled versions of the OLTP, Fileserver, Webserver, and Webproxy workload personalities.

[Preparation] Before running any experiment, execute the *betrf*s-eurosys22-artifact repository's `pre-betrfs-v0.4.sh` script to prepare the environment for BetrFS v0.4, `pre-betrfs-v0.6.sh` script for BetrFS v0.6, and `pre-other.sh` for btrfs, ext4, xfs, zfs, and f2fs.

[Execution] To run all experiments for a given environment, execute the *betrf*s-eurosys22-artifact repository's `eval-betrfs-v0.4.sh` for BetrFS v0.4, `eval-betrfs-v0.6.sh` for BetrFS 0.6, and `eval-other.sh` for btrfs, ext4, xfs, zfs, and f2fs.

[Results] All results are placed in CSV files in the `results/` directory.

[Run-time] Although there is variation among the individual file systems, executing all experiments on a single file system takes approximately eight hours on the hardware described in the paper. This run-time estimate includes multiple experiment runs as executed by the scripts.

A.5 General Notes

The *betrf*s repository branches described above correspond to the cumulative application of the paper's optimizations. BetrFS v0.6 is the version of the file system that includes all optimizations; BetrFS v0.6 should be considered the final product of this study.