Efficient Directory Mutations in a Full-Path-Indexed File System

YANG ZHAN, YIZHENG JIAO, and DONALD E. PORTER,
The University of North Carolina at Chapel Hill
ALEX CONWAY, ERIC KNORR, and MARTIN FARACH-COLTON, Rutgers University
MICHAEL A. BENDER and JUN YUAN, Stony Brook University
WILLIAM JANNEN, Williams College
ROB JOHNSON, VMware Research

22

Full-path indexing can improve I/O efficiency for workloads that operate on data organized using traditional, hierarchical directories, because data is placed on persistent storage in scan order. Prior results indicate, however, that renames in a local file system with full-path indexing are prohibitively expensive.

This article shows how to use full-path indexing in a file system to realize fast directory scans, writes, and renames. The article introduces a range-rename mechanism for efficient key-space changes in a write-optimized dictionary. This mechanism is encapsulated in the key-value Application Programming Interface (API) and simplifies the overall file system design.

We implemented this mechanism in B^{ε}-trees File System (BetrFS), an in-kernel, local file system for Linux. This new version, BetrFS 0.4, performs recursive greps 1.5x faster and random writes 1.2x faster than BetrFS 0.3, but renames are competitive with indirection-based file systems for a range of sizes. BetrFS 0.4 outperforms BetrFS 0.3, as well as traditional file systems, such as ext4, Extents File System (XFS), and Z File System (ZFS), across a variety of workloads.

CCS Concepts: • **Information systems** → **Key-value stores**; **Indexed file organization**; • **Software and its engineering** → **File systems management**;

Additional Key Words and Phrases: B^{ε}-trees, file system, write optimization

---

This research was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, IIS 1251137, IIS-1247750, CCF 1617618, CCF 1439084, CCF-1314547, and by NIH grant CA198952-01. The work was also supported by VMware, by EMC, and by NetApp Faculty Fellowships.
Authors' addresses: Y. Zhan, Y. Jiao, and D. E. Porter, Department of Computer Science, The University of North Carolina at Chapel Hill, Chapel Hill, NC, 27599; emails: {yzhan, yizheng, porter}@cs.unc.edu; A. Conway, E. Knorr, and M. Farach-Colton, Department of Computer Science, Rutgers University, Piscataway, NJ, 08854; emails: alexander.conway@rutgers.edu, erk58@scarletmail.rutgers.edu, farach@cs.rutgers.edu; M. A. Bender and J. Yuan, New Computer Science, Stony Brook University, Stony Brook, NY, 11794; emails: {bender, junyuan}@cs.stonybrook.edu; W. Jannen, Computer Science Department, Williams College, Williamstown, MA, 01267; email: jannen@cs.williams.edu; R. Johnson, VMware Research, 3425 Hillview Ave, Palo Alto, CA, 94304; email: robj@vmware.com.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
1553-3077/2018/11-ART22 $15.00
https://doi.org/10.1145/3241061

ACM Reference format:
Yang Zhan, Yizheng Jiao, Donald E. Porter, Alex Conway, Eric Knorr, Martin Farach-Colton, Michael A. Bender, Jun Yuan, William Jannen, and Rob Johnson. 2018. Efficient Directory Mutations in a Full-Path-Indexed File System. *ACM Trans. Storage* 14, 3, Article 22 (November 2018), 27 pages.
https://doi.org/10.1145/3241061

1 INTRODUCTION

Today's general-purpose file systems do not fully utilize the bandwidth of the underlying hardware. An ideal file system guarantees fast reads and writes at every scale. For example, ext4 can write large files at near disk bandwidth but typically creates small files at less than 3% of disk bandwidth. Similarly, ext4 can read large files at near disk bandwidth, but scanning directories with many small files has performance that ages over time. For instance, a git version-control workload can degrade ext4 scan performance by up to 15× [13, 55].

At the heart of this issue is how data is organized, or ***indexed***, on disk. The most common design pattern for modern file systems is to use a form of indirection, such as inodes, between the name of a file in a directory and its physical placement on disk. Indirection simplifies implementation of some metadata operations, such as renames or file creates, but the contents of the file system can end up scattered over the disk in the worst case. Cylinder groups and other best-effort heuristics [34] are designed to mitigate this scattering.

Full-path indexing is an alternative to indirection, known to have good performance on nearly all operations. File systems that use full-path indexing store data and metadata in depth-first-search order, that is, lexicographic order by the full-path names of files and directories. With this design, scans of any directory subtree (e.g., `ls -R` or `grep -r`) should run at near disk bandwidth. The challenge is maintaining full-path order as the file system changes. Prior work [15, 23, 24] has shown that the combination of write-optimization [5–10, 38, 44, 45] with full-path indexing can realize efficient implementations of many file system updates, such as creating or removing files, which makes relatively small changes to the depth-first-search order, but a few operations still have prohibitively high overheads.

The Achilles' heel of full-path indexing is the performance of renaming large files and directories. For instance, renaming a large directory changes the path to every file in the subtree rooted at that directory, which changes the depth-first search order dramatically. Competitive rename performance in a full-path-indexed file system requires making these changes in an I/O-efficient manner.

BetrFS 0.1 had slow renames because these changes in depth-first search order were implemented by explicitly moving the data, resulting in large I/O costs. This leads to considerable I/O overheads and disrupts the depth-first-search order. Indeed, it might seem that these I/O costs are inherent because so much of the depth-first search order needs to change.

The primary contribution of this article is showing that one *can*, in fact, use full-path indexing in a file system without introducing unreasonable rename costs. A file system can use full-path indexing to improve directory locality—and still have efficient renames.

Previous Full-Path-Indexed File Systems. The first version of BetrFS [23, 24] (v0.1), explored full-path indexing. BetrFS uses a write-optimized dictionary to ensure fast updates of large and small data and metadata, as well as fast scans of files and directory-tree data and metadata. Specifically, BetrFS uses two B^{ε}-trees [7, 9] as persistent key-value stores, where the keys are full path names to files and the values are file contents and metadata, respectively. B^{ε}-trees organize data on disk such that logically contiguous key ranges can be accessed via large, sequential I/Os. B^{ε}-trees aggregate small updates into large, sequential I/Os, ensuring efficient writes.

This design established the promise of full-path indexing when combined with B$^{\varepsilon}$-trees. Recursive greps run 3.8x faster than in the best standard file system. File creation runs 2.6x faster. Small, random writes to a file run 68x faster.

However, renaming a directory has predictably miserable performance [23, 24]. For example, renaming the Linux source tree, which must delete and reinsert all the data to preserve locality, takes 21.2s in BetrFS 0.1, as compared to 0.1s in Btrfs (copy-on-write B-tree file system).

Relative-Path Indexing. BetrFS 0.2 backed away from full-path indexing and introduced zoning [55, 56]. Zoning is a schema-level change that implements ***relative-path indexing***. In relative-path indexing, each file or directory is indexed relative to a local neighborhood in the directory tree. See Section 2.2 for details.

Zoning strikes a "sweet spot" on the spectrum between indirection and full-path indexing: large file and directory renames are comparable to indirection-based file systems, and sequential scans are at least 2x faster than inode-based file systems (but 1.5x slower than BetrFS 0.1).

There are, however, a number of significant, diffuse costs to relative-path indexing, which tax the performance of seemingly unrelated operations. For instance, two-thirds of the way through the TokuBench benchmark, BetrFS 0.2 shows a sudden, precipitous drop in cumulative throughput for small file creations, which is due to the cost of maintaining zones (Figure 4).

Perhaps the most intangible cost of zoning is that it introduces complexity into the system and thus hides optimization opportunities. In a full-path-indexed file system, one can implement nearly all file system operations as simple point or range operations. Adding indirection breaks this simple mapping. Indirection generally causes file system operations to map onto more key/value operations and often introduces reads before writes. Because reads are slower than writes in a write-optimized file system, making writes depend upon reads forgoes some of the potential performance benefits of write-optimization.

Consider `rm -r`, for example. With full-path indexing, one can implement this operation with a single range-delete message, which incurs almost no latency and requires a single synchronous write to become persistent [55, 56]. Using a single range-delete message also unlocks optimizations internal to the key/value store, such as freeing a dead leaf without first reading it from disk. Adding indirection on some directories (as in BetrFS 0.2) requires a recursive delete to scatter reads and writes throughout the keyspace and disk (Table 3).

Our Contributions. This article presents a B$^{\varepsilon}$-tree variant, called a ***lifted B$^{\varepsilon}$-tree***, that can efficiently rename a range of lexicographically ordered keys, unlocking the benefits of full-path indexing. We demonstrate the benefits of a lifted B$^{\varepsilon}$-tree in combination with full-path indexing in a new version of BetrFS, version 0.4, which achieves:

- fast updates of data and metadata,
- fast scans of the data and metadata in directory subtrees and fast scans of files,
- fast renames, and
- fast subtree operations, such as recursive deletes.

We introduce a new key/value primitive called ***range rename***. Range renames are the keyspace analogue of directory renames. Given two strings, p_1 and p_2, range rename replaces prefix p_1 with prefix p_2 in all keys that have p_1 as a prefix. Range rename is an atomic modification to a contiguous range of keys, and the values are unchanged. Our main technical innovation is an efficient implementation of range rename in a B$^{\varepsilon}$-tree. Specifically, we reduce the range rename cost from the size of the subtree to the height of the subtree.

Using range rename, BetrFS 0.4 returns to a simple schema for mapping file system operations onto key/value operations; this in turn consolidates all placement decisions and locality

optimizations in one place. The result is simpler code with less ancillary metadata to maintain, leading to better performance on a range of seemingly unrelated operations.

The technical insight behind efficient B^{ε}-tree range rename is a method for performing large renames by direct manipulation of the B^{ε}-tree. Zoning shows us that small key ranges can be deleted and reinserted cheaply. For large key ranges, range rename is implemented by slicing the tree at the source and destination. Once the source subtree is isolated, a pointer swing moves the renamed section of keyspace to its destination. The asymptotic cost of such tree surgery is proportional to the height, rather than the size, of the tree.

Once the B^{ε}-tree has its new structure, another challenge is efficiently changing the pivots and keys to their new values. In a standard B^{ε}-tree, each node stores the full path keys; thus, a straightforward implementation of range rename must rewrite the entire subtree.

We present a method that reduces the work of updating keys by removing the redundancy in prefixes shared by many keys. This approach is called ***key lifting*** (Section 5). A lifted B^{ε}-tree encodes its pivots and keys such that the values of these strings are defined by the path taken to reach the node containing the string. Using this approach, the number of paths that need to be modified in a range rename also changes from being proportional to the size of the subtree to the depth of the subtree.

Our evaluation shows improvement across a range of workloads. For instance, BetrFS 0.4 performs recursive greps 1.5x faster and random writes 1.2x faster than BetrFS 0.3, but renames are competitive with standard, indirection-based file systems. As an example of simplicity unlocked by full path indexing, BetrFS 0.4 implements recursive deletion with a single range delete, significantly out-performing other file systems.

2 BACKGROUND

This section presents background on BetrFS, relevant to the proposed changes to support efficient keyspace mutations. Additional aspects of the design are covered in previous articles [7, 23, 24, 55, 56], though many details are otherwise undocumented outside of reading the released source code.

2.1 B^{ε}-Tree Overview

The B^{ε}-tree is a ***write-optimized*** B-tree variant that implements the standard key/value store interface: insert, delete, point query, and predecessor and successor queries (i.e., range queries). By write-optimized, we mean the insertions and deletions in B^{ε}-trees are orders of magnitude faster than in a B-tree, while point queries are just as fast as in a B-tree. Furthermore, range queries and sequential insertions and deletions in B^{ε}-trees can run at near disk bandwidth.

Because insertions are much faster than queries, the common read-modify-write pattern can become bottlenecked on the read. Therefore, B^{ε}-trees provide an ***upsert*** that logically encodes, but lazily applies a read-modify-write of a key-value pair. An upsert operation is semantically equivalent to performing a read-modify-write of the value associated with a key and converts a read-modify-write to a blind insert. Thus, upserts are as fast as inserts.

In order to support efficient metadata operations, BetrFS keeps data and metadata in separate B^{ε}-trees. Like B-trees, B^{ε}-trees store key/value pairs in nodes, where they are sorted by key order. Also like B-trees, interior nodes store pointers to children, and ***pivot keys*** delimit the range of keys in each child.

The main distinction between B^{ε}-trees and B-trees is that interior B^{ε}-tree nodes are augmented to include ***message buffers***. A B^{ε}-tree models all changes (inserts, deletes, upserts) as ***messages***. Insertions, deletions, and upserts are implemented by inserting messages into the buffers, starting with the root node. If an interior node's buffer becomes full, the buffer's messages are *flushed* to the

node's children until messages are eventually applied to key/value pairs in leaves. A key technique behind write-optimization is that messages can accumulate in a buffer, and are flushed down the tree in larger batches, which amortize the costs of rewriting a node. Most notably, this batching can improve the costs of small, random writes by orders of magnitude.

The queries of B^{ε}-trees are tree-height operations because all the messages that aim for the same leaf must sit on the path from the root to the leaf they belong to. Since messages lazily propagate down the tree, queries may require traversing the entire root-to-leaf search path, checking for relevant messages in each buffer along the way. The newest target value is returned (after applying pending upsert messages, which encode key/value pair modifications).

In practice, B^{ε}-trees are often configured with large nodes (typically ≥4MiB) and fanouts (typically ≤16) to improve performance. Large nodes mean updates are applied in large batches, but large nodes also mean that many contiguous key/value pairs are read per I/O. Thus, range queries can run a near disk bandwidth, with at most one random I/O per large node.

The B^{ε}-tree implementation in BetrFS supports both ***point*** and ***range*** messages; range messages were introduced in v0.2 [55, 56]. A point message is addressed to a single key, whereas a range message is applied to a contiguous range of keys. Thus far, range messages have only been used for deleting a range of contiguous keys with a single message. In our experience, range deletes give useful information about the keyspace that is hard to infer from a series of point deletions, such as dropping obviated insert and upsert messages.

The B^{ε}-tree used in BetrFS supports transactions and crash consistency as follows. The B^{ε}-tree internally uses a logical timestamp for each message and multiversion concurrency control (MVCC) to implement transactions. Pending messages can be thought of as a history of recent modifications, and a stack of the messages from different transactions are maintained. At any point in the history, one can construct a consistent view of the data based on the isolation level and the time of the view. Crash consistency is ensured using logical logging, i.e., by logging the inserts, deletes, and the like, performed on the tree. Internal operations, such as node splits, flushes, and so on, are not logged. Nodes are written to disk using copy-on-write. Data are made durable either through flushing the log (every 5 seconds) or checkpointing. At a periodic checkpoint (every 60 seconds), all dirty nodes are written to disk and the log can be trimmed. Any unreachable nodes are then garbage collected and reused. Crash recovery starts from the last checkpoint, replays the logical redo log, and garbage collects any unreachable nodes; as long as an operation is in the logical log, it will be recoverable.

2.2 BetrFS Overview

BetrFS uses B^{ε}-tree as on-disk indexing structure and translates Virtual File System (VFS)-level operations into B^{ε}-tree operations. Across versions, BetrFS has explored schema designs that map VFS-level operations onto B^{ε}-tree operations as efficiently as possible.

All versions of BetrFS use two B^{ε}-trees: one for file data and one for file system metadata. The B^{ε}-tree implementation supports transactions, which we use internally for operations that require more than one message. BetrFS does not expose transactions to applications, which introduce some more complex issues around system call semantics [26, 36, 41, 47].

In BetrFS 0.1, the metadata B^{ε}-tree maps a full path onto the typical contents of a `stat` structure, including owner, modification time, and permission bits. The data B^{ε}-tree maps keys of the form (p, i), where p is the full path to a file and i is a block number within that file, to 4KB file blocks. Paths are sorted in a variant of depth-first traversal order.

This full-path schema means that entire sub-trees of the directory hierarchy are stored in logically contiguous ranges of the key space. For instance, recursive directory traversals (e.g., `find` or recursive `grep`) are very fast in BetrFS 0.1. Within a file, blocks are sorted by their block offset

so that sequential file reads and writes translate into sequential B^ε-tree queries and inserts, which run at nearly disk bandwidth.[1]

Unfortunately, with this schema, file and directory renames and deletes do not map easily onto key/value store operations. In BetrFS 0.1, file and directory renames were implemented by copying the file or directory from its old location to the new location. File deletions were implemented by performing a B^ε-tree delete on each block of the file. As a result, these operations were orders of magnitude slower than conventional file systems.

BetrFS 0.2 improved rename performance by replacing the full-path indexing schema of BetrFS 0.1 with a ***relative-path*** indexing schema [55, 56]. The goal of relative path indexing is to get the rename performance of inode-based file systems and the recursive-directory-traversal performance of a full-path-indexed file system.

BetrFS 0.2 accomplishes this by partitioning the directory hierarchy into a collection of connected regions called ***zones***. Each zone has a single root file or directory and, if the root of a zone is a directory, it may contain sub-directories of that directory. Each zone is given a zone ID (analogous to an inode number).

The key insight behind relative-path indexing is that the perceived tradeoff between rename and search is a "Goldilocks problem"—full path indexing is too expensive when you have to move large files or directories, and indirection only hurts searches when small items become heavily scattered. Thus, relative-path indexing only uses indirection for files or directories that are sufficiently large; smaller files are indexed and packed together based on the path relative to a common ancestor.

Renaming a zone root requires only updating the name-to-zone identifier in the metadata index and is as efficient as a rename on any other file system, such as ext4 or btrfs. Renaming data inside of a zone, or across zones requires moving the data. Thus, tuning zone sizes sets an upper bound on how much data is moved in a rename.

Relative-path indexing made renames on BetrFS 0.2 almost as fast as inode-based file systems and recursive-directory traversals almost as fast as BetrFS 0.1.

Zoning gets many of the benefits of full-path indexing. However, our experience has been that relative-path indexing introduces a number of overheads and precludes other opportunities for mapping file-system-level operations onto B^ε-tree operations. For instance, zones introduce overhead because zones must be split and merged to keep all zones within a target size range, which is a critical performance invariant. These overheads can become a first-order performance issue, as in the Tokubench benchmark results for BetrFS 0.2.

Furthermore, relative-path indexing also has bad worst-case performance. It is possible to construct arrangements of nested directories that will each reside in their own zone. Reading a file in the deepest directory will require reading one zone per directory (each with its own I/O). Such a pathological worst case is not possible with full-path indexing in a B^ε-tree, and an important design goal for BetrFS is keeping a reasonable bound on the worst cases.

Finally, zones break the clean mapping of directory subtrees onto contiguous ranges of the key space, preventing us from using range-messages to implement bulk operations on entire subtrees of the directory hierarchy. For example, with full-path indexing, we can use range-delete messages not only to delete files, but entire subtrees of the directory hierarchy. We could also use range messages to perform a variety of other operations on subtrees of the directory hierarchy, such as recursive `chmod`, `chown`, and timestamp updates.

At the root of the problem of zones is that the logical structure of the file system is obscured from the heuristics in the tree, leading to duplicated work, heuristics working at cross-purposes,

[1]Sequential file IO in BetrFS 0.1 ran at roughly a third of disk bandwidth due to logging and other overheads that were addressed in v0.2.

and missed optimization opportunities. The goal of this article is to show that by making rename a first-class key/value store operation, we can use full-path indexing to produce a simpler, more efficient, and more flexible system end-to-end.

3 OVERVIEW

The goal of this section is to explain the performance considerations behind our data structure design and to provide a high-level overview of that design. For most operations, indexing data by full path is straightforward—simply map a path onto metadata or data. The challenge is efficiently renaming large files or directories. The following sections focus on the design of a B$^{\varepsilon}$-tree variant that can implement efficient rename operations.

Our high-level strategy is to simply copy small files and directories in order to preserve locality—i.e., copying a few-byte file is no more expensive than updating a pointer. Once a file or directory becomes sufficiently large, copying the data becomes expensive and of diminishing value, i.e., the cost of indirection is amortized over more data. Thus, most of what follows is focused on efficiently renaming *large* files and directories—"large" meaning at least as large as a B$^{\varepsilon}$-tree node.

Since we index file and directory data and metadata by full path, a file or directory rename translates into a prefix replacement on a *contiguous* range of keys. For example, if we rename directory `/tmp/draft` to `/home/paper/final`, then we want to find all keys in the B$^{\varepsilon}$-tree that begin with `/tmp/draft` and replace that prefix with `/home/paper/final`. This involves both updating the key itself, and updating its location in the B$^{\varepsilon}$-tree so that future searches can find it.

Since the affected keys form a contiguous range in the B$^{\varepsilon}$-tree, we can move the keys to their new (logical) home without moving them physically. Rather, we can make a small number of pointer updates and other changes to the tree. We call this step ***tree surgery***. We then need to update all the keys to contain their new prefix, a process we call ***batched key update***.

As described in Section 2, internal message buffers store pending changes for one or more (contiguous) keys, which can be thought of as a history of pending updates, that are replayed upon reads, or eventually applied to a leaf (a.k.a., flushing). At any point in the stream of messages, there is a consistent history of changes. Messages in the history before the *range-rename* are moved to the new destination; messages in the history after the *range-rename* are kept at the original location. Similarly, queries before the *range-rename* can find the key-value pairs at the old location, while queries after the *range-rename* would observe the key-value paris at the new location.

In summary, the algorithm has two high-level steps:

Tree Surgery. We identify a subtree of the B$^{\varepsilon}$-tree that includes all keys in the range to be renamed (Figure 1). Any ***fringe*** nodes (i.e., on the left and right extremes of the subtree), which contain both related and unrelated keys, are split into two nodes: one containing only affected keys and another containing only unaffected keys. The number of fringe nodes will be at most logarithmic in the size of the sub-tree. At the end of the process, we will have a subtree that contains only keys in the range being moved. We then change the pivot keys and pointers to move the subtree to its new parent.

Batched Key Updates. Once a subtree has been logically relocated, full-path keys in the subtree will still reflect the original key range. We describe a B$^{\varepsilon}$-tree modification to make these batched key updates efficient. Specifically, we modify the B$^{\varepsilon}$-tree to factor out common prefixes from keys in a node, similar to prefix-encoded compression. We call this transformation ***key lifting***. This transformation does not lose any information—the common prefix of keys in a node can be inferred from the pivot keys along the path from the root to the node by concatenating the longest common prefix of enclosing pivots along the path. As a result of key lifting, once we perform tree surgery to isolate the range of keys affected by a rename, the prefix to be replaced in each key will already

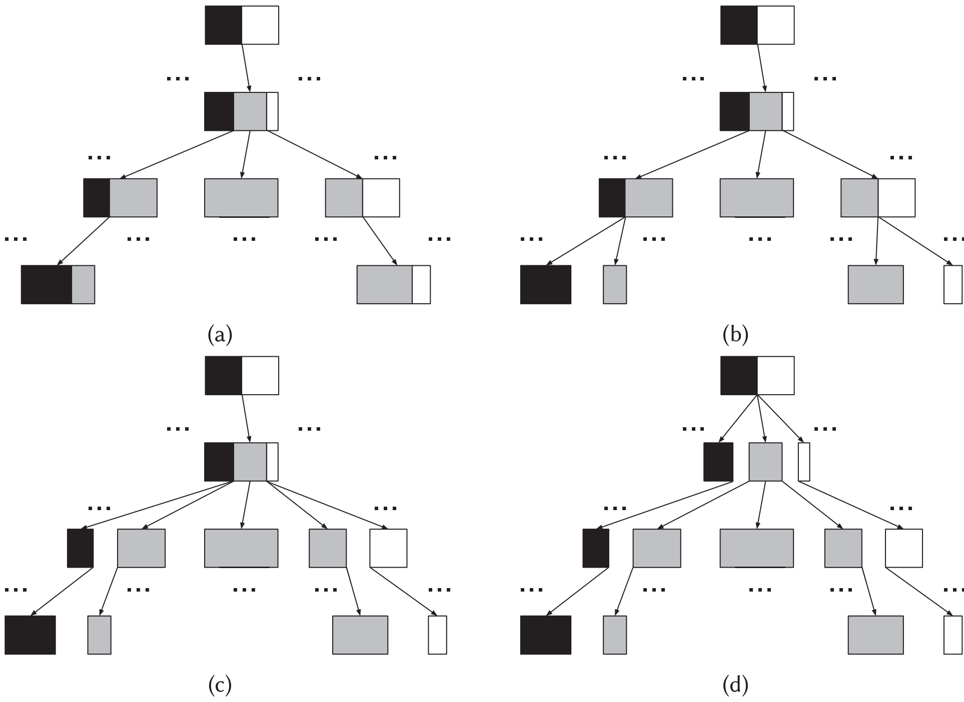


Fig. 1. Slicing /gray between /black and /white.

be removed from every key in the sub-tree. Furthermore, since the omitted prefixes are inferred from the sub-tree's parent pivots, moving the sub-tree to its new parent implicitly replaces the old prefix with the new one. Thus, no further work needs to be done to perform the batch key update after tree surgery, and a large subtree can be left untouched on disk during a range rename. In the worst case, only a logarithmic number of nodes on the fringe of the subtree will have keys that need to be updated.

Buffered Messages and Concurrency. Our range rename operation must also handle any pending messages targeting the affected keys. These messages may be buffered in any node along a search path from the root to one of the affected keys. Our solution leverages the fact that messages have a logical timestamp and are applied in logical order. Thus, it is sufficient to ensure that pending messages for a to-be-renamed subtree must be flushed into the subtree before we begin the tree surgery for a range rename.

Note that most of the work in tree surgery involves node splits and merges, which are part of normal B^{ε}-tree operation. Thus, the tree remains a "valid" B^{ε}-tree during this phase of the range rename. Only the pointer swaps need to be serialized with other operations. Thus, this approach does not present a concurrency bottleneck.

The following two sections explain tree surgery and lifting in more detail.

4 TREE SURGERY

This section describes our approach to renaming a directory or large file via changes within the B^{ε}-tree, such that most of the data is not physically moved (or even accessed) on disk. Files that are smaller than 4MiB reside in at most two leaves. We, therefore, move them by copying. We reserve tree surgery only for larger files and, for simplicity of the prototype, directories of any size.

For the sake of discussion, we assume that a rename is moving a source file over an existing destination file; the process would work similarly (but avoid some work) in the case where the destination file does not exist. Our implementation respects Portable Operating System Interface (POSIX) restrictions for directories (i.e., you cannot rename over a non-empty destination directory), but our technique could easily support different directory rename semantics. In the case of renaming over a file, where a rename implicitly deletes the destination file, we use transactions in the B^ε-tree to insert both a range delete of the destination and a range rename of the source; these messages are applied atomically.

This section also operates primarily at the B^ε-tree level, not the directory namespace. Unless otherwise noted, pointers are pointers within the B^ε-tree.

A B^ε-tree node ***covers*** a key k if and only if the node is on k's search path. Note that a node covers k, even if k is not a key in the tree. Each B^ε-tree node thus covers a range of keys defined by a max and min key. For most nodes, the max and min are the pivots bracketing the pointer from the parent to the child. But for some nodes, such as the first or last child of a parent, the range will be defined by a pivot at the parent and a pivot at a more distant ancestor.

In renaming a file, the goal is to capture a range of contiguous keys and logically move these key/value pairs to a different point in the tree. For anything large enough to warrant using this rename approach, some B^ε-tree nodes will exclusively store messages or key/value pairs for the source or destination. Some nodes may also include unrelated messages or key/value pairs before and after this key range in sort order; within the B^ε-tree, these nodes will be along the left and right side of the nodes that store only the messages or key/value pairs of interest.

An important abstraction for tree surgery is the ***Lowest Common Ancestor***, or ***(LCA)***, of two keys: the LCA is the B^ε-tree node lowest in the tree on the search path for both keys (and, hence, including all keys in between). During a range rename, the source and destination key ranges will each have an LCA, and their LCAs may be the same node.

The first step in tree surgery is to find the source LCA and destination LCA. In the process of identifying the LCAs, we also flush any pending messages for the source or destination key range so that they are buffered at or below the corresponding LCAs.

Slicing. The second step is to slice out the source and destination key ranges from any shared nodes. The goal of slicing is to separate unrelated key-value pairs that are not being moved but are packed into the same B^ε-tree node as the key ranges to be move. Slicing uses the same code used for standard B^ε-tree node splits, but slicing divides the node at the slicing key rather than picking a key in the middle of the node. As a result, slicing may result in nodes that temporarily violate constraints on target node size and fanout. However, these are performance, not correctness, constraints, so we can let queries continue concurrently, and we restore these invariants before completing the rename.

Figure 1 depicts slicing the sub-tree containing all gray keys from a tree with black, gray, and white keys. The top node is the parent of the LCA. Because messages have been flushed to the LCA, the parent of the LCA contains no messages related to gray. Slicing proceeds up the tree from the leaves and only operates on the left and right fringe of the gray sub-tree. Essentially, each fringe node is split into two smaller B^ε-tree nodes (see steps 1(b) and 1(c)). All splits, as well as later transplanting and healing, happen when the nodes are pinned in memory. During surgery, they are dirtied and written at the next checkpoint. Eventually, the left and right edge of an exclusively-gray subtree (step 1(d)) is pinned, whereas interior, all-grey nodes may remain on disk.

Our implementation requires that the source and destination LCA be at the same height for the next step. Thus, if the LCAs are not at the same level of the tree, we slice up to an ancestor of the higher LCA. The goal of this choice is to maintain the invariant that all B^ε-tree leaves be at the same depth.

Transplanting. Once the source and destination are both sliced, the source and destination LCAs are the roots of subtrees that contain only the key ranges to be moved and deleted, respectively. The transplanting step swaps the pointers to each LCA, or subtree, in the respective parents of the LCAs. The parent of the source now points to a subtree containing all data that once resided at the destination; we insert a range-delete message at the source, which triggers the B^{ε}-tree's built-in garbage collection, which will reclaim these nodes.

After this swap completes and the locks are released, new searches or updates for the destination will descend into the new subtree.

Healing. Our B^{ε}-tree implementation maintains the invariant that all internal nodes have between 4 and 16 children, which bounds the height of the tree. After the transplant completes, however, there may be a number of in-memory B^{ε}-tree nodes at the fringe around the source and destination that have fewer than four children.

We handle this situation by triggering a rebalancing within the tree. Specifically, if a node has only one child, the slicing process will merge it after completing the work of the rename. After the transplant completes, there may be a number of B^{ε}-tree nodes in memory at the fringe around the source and destination that have fewer children than desired. The healing process merges these smaller nodes back together, using the same approach as a typical B^{ε}-tree merge. At this point, it is also possible that this could trigger a rebalancing within the tree.

One very important optimization we found as part of healing was to agressively merge *stalks*, or new descendants of an LCA with only a single child. When the source and destination LCA are at different height, slicing will create some nodes with a single child. Traversing these stalks undermines the logarithmic search time one would expect from a tree and creates significant performance overheads. After healing, we immediately merge nodes at the fringe to a normal fanout in B^{ε}-tree.

Crash Consistency. BetrFS relies on three on-disk data structures to ensure crash consistency: the trees, a block table indicating the disk blocks used by the trees, and a write-ahead redo log. In general, BetrFS ensures crash consistency by appending pending messages to the redo log and then applying messages to the tree in a copy-on-write manner. At periodic intervals (by default every 60 seconds), BetrFS ensures that there is a consistent, sharp checkpoint of the tree on disk. After a crash, BetrFS reads the last checkpointed tree and replays the redo log after the checkpoint. Range rename works within this framework.

A range rename is *logically applied* to the tree as soon as (1) the range-rename message is added to the redo log and (2) tree surgery locks the root of the tree. The range rename is durable as soon as the redo log entry is written to disk. If the system crashes after a range rename is logged, the recovery will see a prefix of the message history that includes the range-rename message, and the tree surgery will be redone in response to the range-rename message. After both the tree surgery and the subsequent checkpoint of the tree complete, there is no need to repeat the tree surgery after a crash.

We note that the current implementation does tree surgery immediately after the message is placed in the redo log. It is possible to batch range rename messages and apply them lazily, but we leave this for future work. As a result of this choice, however, there are fewer cases to reason about in understanding crash consistency.

Tree surgery walks down the tree and locks the nodes along the path hand-over-hand until either the source LCA or the destination LCA is reached. Then, until tree surgery completes, all fringe nodes, the LCAs, and the parents of LCAs are locked in memory and dirtied. Upon tree surgery completion, these nodes will be unlocked. Because the checkpoint process must grab the locks of all involved nodes before writing them to disk, no intermediate state of slicing will be

exposed to a checkpoint. Dirty nodes may be written back to disk, copy-on-write, before checkpointing under memory pressure. However, because the recovery process always starts from the last checkpoint and the last block table in which these newly allocated nodes is not reachable, these nodes with an uncheckpointed state will be marked as free during crash recovery.

If the system crashes after tree surgery begins but before surgery completes, the recovery code will see a consistent checkpoint of the tree as it was before the tree surgery. The same is true if the system crashes after tree surgery but before dirty nodes are written back to disk. Because a checkpoint flushes all dirty nodes, if the system crashes after a checkpoint, all nodes affected by tree surgery will be on disk. We checked this functionality with crash recovery unit tests. We ran the system in QEMU and crashed the system both before a rename completed and before the rename log had been flushed; we ensured that BetrFS restored the tree to the state before the tree surgery.

Tree surgery maintains the transactional properties of BetrFS. It is performed at the commit stage of a range-rename transaction, and, thus, aborting a range-rename transaction leaves no trace on the tree. In the case of a crash, the tree surgery, which is encoded as a range-rename message in the redo log, will be redone when the recovery process replays the redo log. The key to maintaining failure-atomicity of surgery is that all the involved nodes must be locked in memory in the same batch to ensure the checkpoint has a consistent snapshot of the tree at a point in time before or after, but not during, the tree surgery.

At the file system level, BetrFS has similar crash consistency semantics to metadata-only journaling in ext4. The B^{ε}-tree implementation itself implements full data journaling [55, 56], but BetrFS allows file writes to be buffered in the VFS, weakening this guarantee end-to-end. Specifically, file writes may be buffered in the VFS caches, and are only logged in the recovery journal once the VFS writes back a dirty page (e.g., upon an `fsync` or after a configurable period). Changes to the directory tree structure, such as a `rename` or `mkdir` are persisted to the log immediately. Thus, in the common pattern of writing to a temporary file and then renaming it, it is possible for the rename to appear in the log before the writes. In this situation and in the absence of a crash, the writes will eventually be logged with the correct, renamed key, as the in-memory inode will be up-to-date with the correct B^{ε}-tree key. If the system crashes, these writes can be lost; as with a metadata-journaled file system, the developer must issue an `fsync` before the `rename` to ensure the data is on disk.

Latency. A rename returns to the user once a log entry is in the journal and the root node of the B^{ε}-tree is locked. At this point, the rename has been applied in the VFS to in-memory metadata, and as soon as the log is fsynced, the rename is durable.

We then hand off the rest of the rename work to two background threads to do the cutting and healing. The prototype in this article only allows a backlog of one pending, large rename, since we believe that concurrent renames are relatively infrequent. The challenge in adding a rename work queue is ensuring consistency between the work queue and the state of the tree.

Atomicity and Transactions. The B^{ε}-tree in BetrFS implements multi-version concurrency control by augmenting messages with a logical timestamp. Messages updating a given key range are always applied in logical order. Multiple messages can share a timestamp, giving them transactional semantics.

To ensure atomicity for a range rename, we create an MVCC "hazard": read transactions "before" the rename must complete before the surgery can proceed. Tree nodes in BetrFS are locked with reader-writer locks. We write-lock tree nodes hand-over-hand and left-to-right to identify the LCAs. Once the LCAs are locked, this serializes any new read or write transactions until the rename completes. The lock at the LCA creates a "barrier"—operations can complete "above" or "below"

this lock in the tree, although the slicing will wait for concurrent transactions to complete before write-locking that node. Once the transplant completes, the write-locks on the parents above LCAs are released.

In practice, BetrFS has a reader-writer lock in each in-memory inode to prevent concurrent transactions from happening during a rename, but this would be a concern in applying this technique to a general-purpose key-value store or database that uses MVCC.

For simplicity, before the range rename is applied, we ensure that all messages representing changes in the affected key range(s) that logically occurred before the range rename are flushed below the LCA. All messages that logically occur after the rename follow the new path through the tree to the destination or source. This strategy ensures that when each message is flushed and applied, it sees a point-in-time consistent view of the subtree.

Complexity. In the worst case, at most four slices are performed. Slices go from the parent of the LCA to the leaf, and nodes along this path are dirtied. If the root of the tree is the LCA, a new root is inserted above the current root for slicing. These nodes will need to be read, if not in cache, and written back to disk as part of the checkpointing process. Therefore, the number of I/Os is at most proportional to the height of the B^{ε}-tree, which is logarithmic in the size of the tree.

Let N be the number of entries in a B^{ε}-tree, i.e., the number of files plus directories in the metadata tree and the number of data blocks in the data tree. Let B be the size of a node. And let $\varepsilon \in (0, 1]$ be a design-time tuning parameter that adjusts the fanout (for more details, see Ref. [7]). In the worst case, $O(\frac{\log_B N}{\varepsilon})$ (tree height) I/Os will be performed during a rename as a result of this design.

5 BATCHED KEY UPDATES

After tree surgery completes, there will be a subtree where the keys are not coherent with the new location in the tree. As part of a rename, the prefixes of all keys in this subtree need to be updated. For example, suppose we execute "`mv /foo /bar`". After surgery, any messages and key/value pairs for file `/foo/file` will still have a key that starts with `/foo`. These keys need to be changed to begin with `/bar`. The particularly concerning case is when `/foo` is a very large subtree and has interior nodes that would otherwise be untouched by the tree surgery step; our goal is to leave these nodes untouched as part of rename and, thus, reduce the cost of key changes from the size of the rename tree to the height of the rename tree.

We note that full-path keys in our tree are highly redundant. Our solution reduces the work of changing keys by reducing the redundancy of how keys are encoded in the tree. Consider the prefix encoding for a sequence of strings. In this compression method, if two strings share a substantial longest common prefix (lcp), then that lcp is only stored once. We apply this idea to B^{ε}-trees. The lcp of all keys in a subtree is removed from the keys and stored in the subtree's parent node. We call this approach ***key lifting*** or simply ***lifting*** for short.

At a high level, each parent in a lifted B^{ε}-tree stores each child's common, lifted key prefix alongside the pointer to the child. Child nodes only store differing key suffixes. This approach encodes the complete key in the path taken to reach a given node.

This solution leaves nodes interior to the subtree to be unmodified as part of a range rename. Specifically, part of finding an LCA during tree surgery ensures that the prefix to be renamed is lifted into the parent.

One can then modify the prefix for a lifted subtree by only modifying the parent node. This solution eliminates the need to change key and pivot prefixes in all nodes of a subtree.

Lifting requires a schema-level invariant that keys with a common prefix are adjacent in the sort order. As a simple example, if one uses `memcmp` to compare keys (as BetrFS does), then lifting will

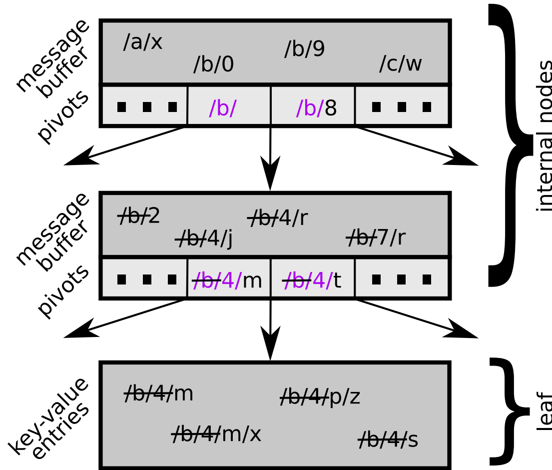


Fig. 2. Example nodes in a lifted B$^{\varepsilon}$-tree. Since the middle node is bounded by two pivots with common prefix "/b/" (indicated by purple text), all keys in the middle node and its descendants must have this prefix in common. Thus, this prefix can be omitted from all keys in the middle node (and all its descendants), as indicated by the strike-through text. Similarly, the bottom node (a leaf) is bounded by pivots with common prefix "/b/4/", so this prefix is omitted from all its keys. The node pivots in a full-path-indexed B$^{\varepsilon}$-tree with and without lifting (per-node buffers that store messages are omitted from the figure). To reconstruct any key, common prefixes are concatenated during tree traversal and the key's suffix is appended.

be correct. This invariant ensures that, if there is a common prefix between any two pivot keys, all keys in that child will have the same prefix, which can be safely lifted. More formally:

Invariant 1. *Let T' be a subtree in a B$^{\varepsilon}$-tree with full-path indexing. Let p and q be the pivots that enclose T'. That is, if T' is not the first or last child of its parent, then p and q are the enclosing pivots in the parent of T'. If T' is the first child of its parent, then q is the first pivot and p is the left enclosing pivot of the parent of T'.*

Let s be the longest common prefix of p and q. Then, all keys in T' begin with s.

Given this invariant, we can strip s from the beginning of every message or key/value pair in T', only storing the non-lifted suffix. Lifting is illustrated in Figure 2, where the common prefix in the first child is "/b/", which is removed from all keys in the node and its children (indicated with strikethrough text). The common prefix (indicated with purple) is stored in the parent. As one moves toward leaves, the common prefix typically becomes longer ("/b/4/" in Figure 2), and each level of the tree can lift the additional common prefix.

Reads can reconstruct the full key by concatenating prefixes during a root-to-leaf traversal. In principle, one need not store the lifted prefix (s) in the tree, as it can be computed from the pivot keys. In our implementation, we do memorize the lifted prefix for efficiency.

As messages are flushed to a child, they are modified to remove the common prefix. Similarly, node splits and merges ensure that any common prefix between the pivot keys is lifted out. It is possible for all of the keys in T' to share a common prefix that is longer than s, but we only lift s because maintaining this amount of lifting hits a sweet spot: *it is enough to guarantee fast key updates during renames, but it requires only local information at a parent and child during splits, merges, and insertions.*

Lifting is completely transparent to the file system. From the file system's perspective, it is still indexing data with a key/value store that is keyed by full-path; the only difference from the file system's perspective is that the key/value store completes some operations faster.

Lifting and Renames. In the case of renames, lifting dramatically reduces the work to update keys. During a rename from a to b, we slice out a sub-tree containing exactly those keys that have a as a prefix. By the lifting invariant, the prefix a will be lifted out of the sub-tree, and the parent of the sub-tree will bound it between two pivots whose common prefix is a (or at least includes a—the pivots may have an even longer common prefix). After we perform the pointer swing, the sub-tree will be bounded in its new parent by pivots that have b as a common prefix. Thus, by the lifting invariant, all future queries will interpret all the keys in the sub-tree as having b as a prefix. Thus, with lifting, the pointer swing implicitly performs the batch key-prefix replacement, completing the rename.

Complexity. During tree surgery, there is lifting work along all nodes that are sliced or merged. However, the number of such nodes is at most proportional to the height of the tree. Thus, the number of nodes that must be lifted after a rename is no more than the nodes that must be sliced during tree surgery, and proportional to the height of the tree.

After slicing out the subtree containing all the source keys involved in the rename, the two source slicing keys become the two pivots bounding the source subtree in the parent. Because the common prefix of the two source slicing keys is the old prefix, the old prefix is lifted from the source subtree. Then, by inserting the subtree to the new location that is bounded by two destination slicing keys with the new prefix, all the keys in the subtree are updated immediately. An end-to-end example of renaming is illustrated in Figure 3. In this example, each slicing step lifts the old-prefix /red. By the time the slice reaches the LCA, the old-prefix /red is lifted out of the entire subtree.

6 IMPLEMENTATION DETAILS

Simplifying Key Comparison. One small difference in the BetrFS 0.4 and BetrFS 0.3 key schemas is that BetrFS 0.4 adjusted the key format so that `memcmp` is sufficient for key comparison. To group all keys in one sub-directory together in a `memcmp`-able way, we add one more slash to the last slash in the full-path and store slashes as `(char)1` in full-paths. For example, "/tmp/foo" and "/tmp/foo/bar" are stored as "\x01tmp\x01\x01foo" and "\x01tmp\x01foo\x01\x01bar", respectively. To specify all keys in a directoy "/dir", we use the range ("\x01dir\x01\x01", "\x01dir\x01\xff").

We found that this change simplified the code, especially around lifting, and helped CPU utilization, as it is hard to compare bytes faster than a well-tuned `memcmp`.

Zone Maintenance. A major source of overheads in BetrFS 0.3 is tracking metadata associated with zones. Each update involves updating significant in-memory bookkeeping; splitting and merging zones can also be a significant source of overhead (c.f., Figure 4). BetrFS 0.4 was able to delete zone maintenance code, consolidating this into the B^{ε}-tree's internal block management code.

Hard Links. BetrFS 0.4 does not support hard links. In future work, for large files, sharing subtrees could also be used to implement hard links. For small files, we expect that broadcasting updates to each link could be efficient in a write-optimized dictionary. Alternatively, zones could be reintroduced solely for hard links.

Garbage Collection. BetrFS 0.4 swaps the source and destination subtrees as the last step of a *range-rename* instead of discarding the destination subtree right away. In theory, the destination subtree should be recycled after the rename. However, deleting a pivot in an internal node may

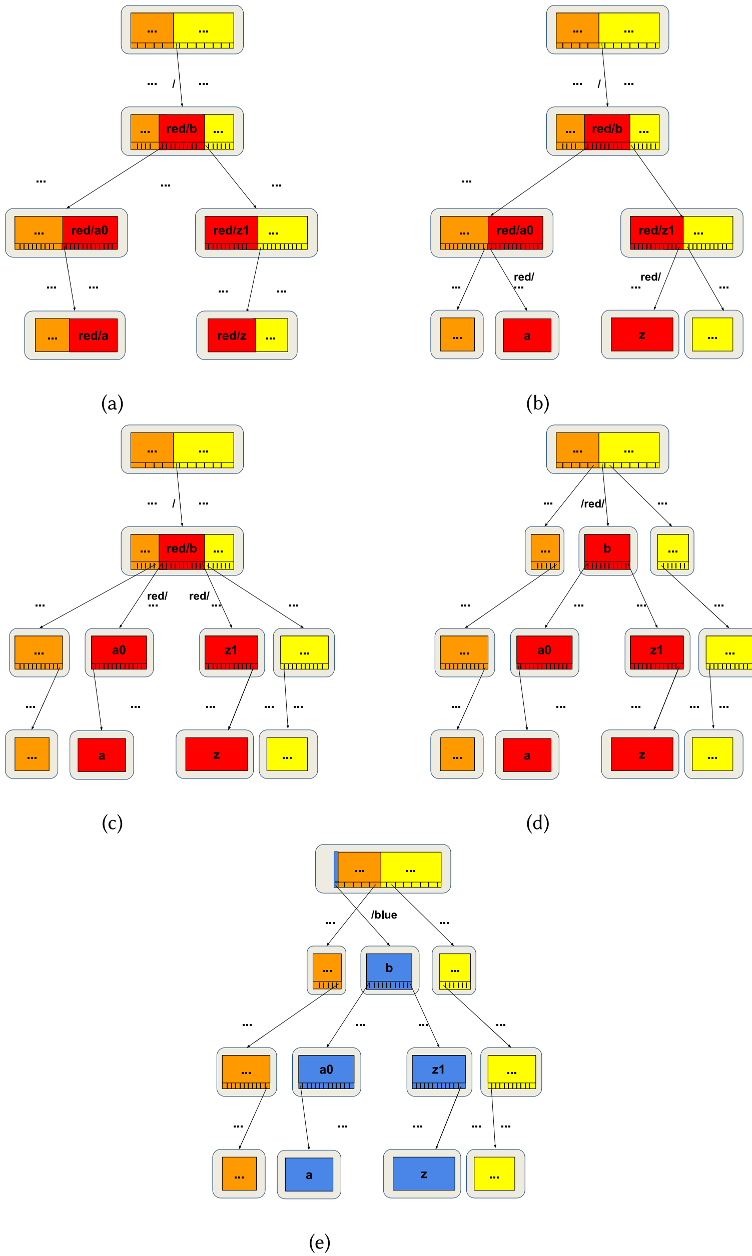


Fig. 3. An end-to-end example of renaming directory "/red" to "/blue". For simplicity, the process of slicing the destination and healing are skipped. (a) There are three directories or files, "/orange", "/red", "/yellow" in the file system, and the tree is already in a lifted state. (b) As the fringe leaves of "/red" are sliced, the common prefix "red/" are lifted out of the leaf as a part of split operation. (c) The slice walks up and lifts the "red/" prefix furthur up. (d) The LCA is sliced and the "red/" is lifted out of the subtree. The resulting isolated source subtree no longer contains prefix "/red/". (e) The sub-tree is moved to the new location with prefix "/blue/" by a pointer swing.

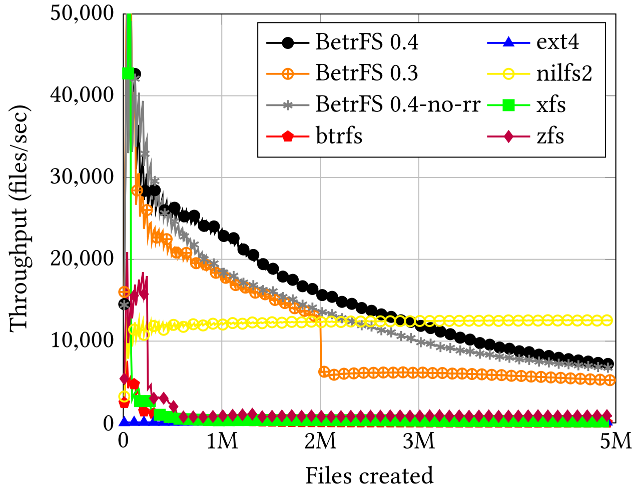


Fig. 4. Cumulative file creation throughput during the Tokubench benchmark (higher is better). BetrFS 0.4 outperforms other file systems by orders of magnitude and avoids the performance drop that BetrFS 0.3 experiences due to its zone-maintenance overhead.

require the lifting and unlifting of a child, which might cause additional IOs. Thus, BetrFS 0.4 relies on the range delete messages to garbage collect the destination subtree, which means the destination subtree is not garbage collected until the range delete message is flushed to its leaves.

7 EVALUATION

Our evaluation seeks to answer the following questions:

— (Section 7.1) Does full-path indexing in BetrFS 0.4 improve overall file system performance, aside from renames? Does our new rename implementation really yield performance improvements, compared to relative-path indexing, on other file system operations? For example, how does BetrFS 0.4 compare to BetrFS 0.3 on recursive directory traversals and file creations, e.g., in Tokubench?
— (Section 7.2) Are rename costs acceptable in BetrFS 0.4 ?
— (Section 7.3) What other opportunities does full-path indexing in BetrFS 0.4 unlock?
— (Section 7.4) How is the performance of BetrFS 0.4 compared to other file systems on application benchmarks?

We compare BetrFS 0.4 with several file systems, including BetrFS 0.3 [13], BetrFS 0.4 without *range-rename* (as BetrFS 0.4-no-rr), Btrfs [43], ext4 [33], nilfs2 [27], XFS [48], and ZFS [37]. Each file system's block size is 4096 bytes. We use the versions of XFS, Btrfs, and ext4 that are part of the 3.11.10 kernel, and ZFS 0.6.5.11, downloaded from www.zfsonlinux.org. We use default recommended file system settings unless otherwise noted. For ext4 (and BetrFS), we disabled lazy inode table and journal initialization, as these features accelerate file system creation but slow down some operations on a freshly-created file system; we believe this configuration yields more representative measurements of the file system in steady-state. BetrFS 0.4 without *range-rename* represents the original BetrFS (there are many unrelated optimizations after BetrFS 0.1) with full-path indexing and serves to show how zoning and *range-rename* affect performance in BetrFS. Each experiment was run a minimum of five times. Error bars indicate minimum and maximum times over all runs. Similarly, error ± terms bound minimum and maximum times over all runs.

Table 1. Time to Perform Recursive Directory Traversals of the Linux 3.11.10 Source Tree (Lower Is Better)

File system	find (sec)	grep (sec)
BetrFS-0.4	0.227 ± 0.0	3.655 ± 0.1
BetrFS-0.3	0.240 ± 0.0	5.436 ± 0.0
BetrFS-0.4-no-rr	0.230 ± 0.0	3.600 ± 0.1
Btrfs	1.311 ± 0.1	8.068 ± 1.6
ext4	2.333 ± 0.1	42.526 ± 5.2
xfs	6.542 ± 0.4	58.040 ± 12.2
zfs	9.797 ± 0.9	346.904 ± 101.5
nilfs2	6.841 ± 0.1	8.399 ± 0.2

BetrFS 0.4 is Significantly Faster than Every Other File System, Demonstrating the Locality Benefits of Full-Path Indexing.

Unless noted, all benchmarks are cold-cache tests and finish with a file-system sync. For BetrFS 0.3, we use the default zone size of 512KiB.

In general, we expect BetrFS 0.3 to be the closest competitor to BetrFS 0.4 , and focus on this comparison. We include other file system measurements for context. Relative-path indexing is supposed to get most of the benefits of full-path indexing, with affordable renames; comparing BetrFS 0.4 with BetrFS 0.3 shows the cost of relative-path indexing and the benefit of full-path indexing.

All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40GHz Intel Core i7 CPU, 4GB RAM, and a 500GB, 7200RPM SATA disk, with a 4096-byte block size. The system runs Ubuntu 14.04.5, 64-bit, with Linux kernel version 3.11.10. We boot from a USB stick with the root file system, isolating the file system under test to only the workload.

7.1 Non-Rename Microbenchmarks

Tokubench. The tokubench benchmark creates five million 200-byte files in a balanced directory tree, where no directory is allowed to have more than 128 children.

As Figure 4 shows, zone maintenance in BetrFS 0.3 causes a significant performance drop around 2 million files. This drop occurs all at once because, at that point in the benchmark, all the top-level directories are just under the zone size limit. As a result, the benchmark goes through a period where each new file causes its top-level directory to split into its own zone. If we continue the benchmark long enough, we would see this happen again when the second-level directories reach the zone-size limit. In experiments with very long runs of Tokubench, BetrFS 0.3 never recovers this performance.

With our new rename implementations, zone maintenance overheads are eliminated. As a result, BetrFS 0.4 has no sudden drop in performance and remains close to BetrFS 0.4 without *range-rename*. By creating 3 million files, only nilfs2 comes close to matching BetrFS 0.4 on this benchmark, in part because nilfs2 is a log-structured file system. From 3 million files to 5 million files, BetrFS 0.4 drops slowly to around 7,000 files/second. BetrFS 0.4 has over 80× higher cumulative throughput than ext4 throughout the benchmark.

Recursive Directory Traversals. In these benchmarks, we run `find` and recursive `grep` on a copy of the Linux kernel 3.11.10 source tree. The times taken for these operations are given in Table 1. BetrFS 0.4 outperforms BetrFS 0.3 by about 5% on find and almost 30% on grep. In the case of grep,

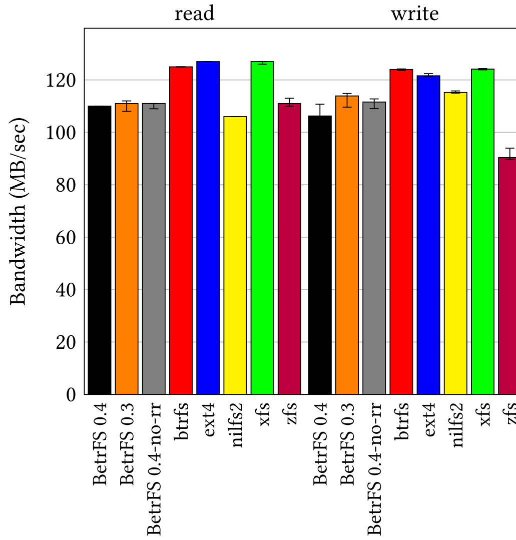


Fig. 5. Sequential IO bandwidth (higher is better). BetrFS 0.4 performs sequential IO at over 100MB/s but up to 19% slower than the fastest competitor. Lifting introduces some overheads on sequential writes.

for instance, we found that roughly the same total number of bytes were read from disk in both versions of BetrFS, but that BetrFS 0.3 issued roughly 25% more I/O transactions. For this workload, we also saw higher disk utilization in BetrFS 0.4 (40MB/s vs. 25MB/s), with fewer worker threads needed to drive the I/O. Lifting also reduces the system time by 5% on grep, but the primary savings are on I/Os. In other words, this demonstrates the locality improvements of full-path indexing over relative-path indexing. BetrFS 0.4 is anywhere from 2 to almost 100 times faster than conventional file systems on these benchmarks.

Sequential IO. Figure 5 shows the throughput of sequential reads and writes of a 10GiB file (more than twice the size of the machine's RAM). All file systems measured, except ZFS, are above 100MB/s, and the disk's raw read and write bandwidth is 132MB/s.

Sequential reads in BetrFS 0.4 are essentially identical to those in BetrFS 0.3 and roughly competitive with other file systems. Both versions of BetrFS do not realize the full performance of the disk on sequential I/O, leaving up to 20% of the throughput compared to ext4 or Btrfs. This is inherited from previous versions of BetrFS and does not appear to be significantly affected by range rename. Profiling indicates that there is not a single culprit for this loss but several cases where writeback of dirty blocks could be better tuned to keep the disk fully utilized. This issue has improved over time since version 0.1, but in small increments.

Writes in BetrFS 0.4 are about 5% slower than in BetrFS 0.3. Comparing to BetrFS 0.4 without *range-rename* and profiling indicates this is because node splitting incurs additional computational costs to re-lift a split child. We believe this can be addressed in future work by either better overlapping computation with I/O or integrating key compression with the on-disk layout, so that lifting a leaf involves less memory copying.

Random Writes. Table 2 shows the execution time of a microbenchmark that issues 256K 4-byte overwrites at random offsets within a 10GiB file, followed by an `fsync`. This is 1MiB of total data written, sized to run for at least several seconds on the fastest file system. BetrFS 0.4 performs small random writes approximately 400 to 650 times faster than conventional file systems and about 19% faster than BetrFS 0.3.

Table 2. Time to Perform 256K 4-byte Random Writes (1MiB Total Writes, Lower Is Better)

File system	random write (sec)
BetrFS-0.3	6.2 ± 0.1
BetrFS-0.4	4.9 ± 0.5
BetrFS-0.4-no-rr	4.9 ± 0.1
Btrfs	2147.5 ± 7.4
ext4	2776.0 ± 40.2
xfs	2835.7 ± 7.9
zfs	3288.9 ± 394.7
nilfs2	2013.1 ± 19.1

BetrFS 0.4 is up to 600 Times Faster Than Other File Systems on Random Writes.

Summary. These benchmarks show that lifting and full-path indexing can improve performance over relative-path indexing for both reads and writes, from 5% up to 2×. The only case harmed is sequential writes. This is primarily attributable to reducing complexity in the code. In short, lifting is generally more efficient than zone maintenance in BetrFS.

7.2 Rename Microbenchmarks

Rename Performance as a Function of File Size. We evaluate rename performance by renaming files of different sizes and measuring the throughput. For each file size, we rename a file of this size 100 times within a directory and `fsync` the parent directory to ensure that the file is persisted on disk. We measure the average across 100 runs and report this as throughput in Figure 6(a).

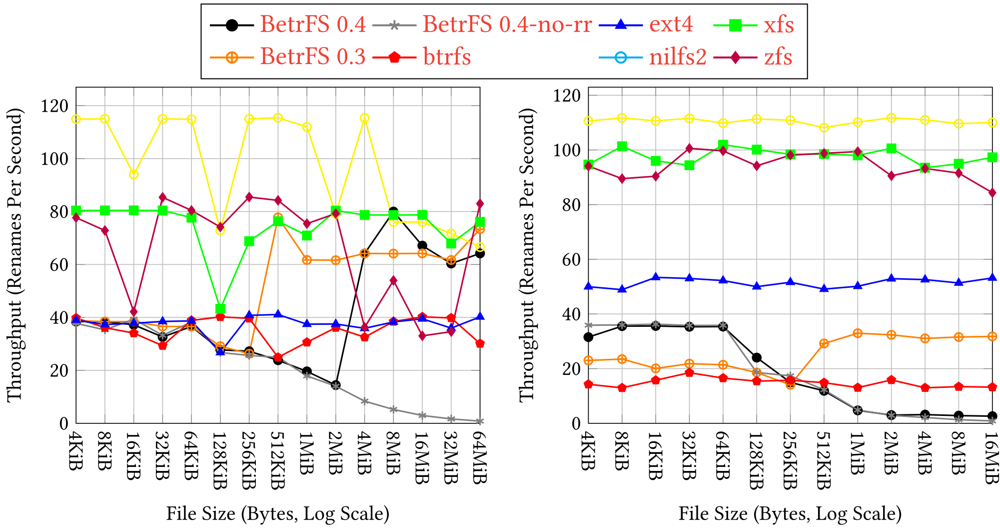
In both BetrFS 0.3 and BetrFS 0.4, there are two modes of operation. For smaller objects, both versions of BetrFS simply copy the data. At 512KiB and 4MiB, BetrFS 0.3 and BetrFS 0.4, respectively, switch modes—this is commensurate with the file matching the zone size limit and node size. For files larger than these thresholds, both file systems see comparable throughput of simply doing a pointer swing.

More generally, the rename throughput of all of these file systems is somewhat noisy, but ranges from 30–120 renames per second, with nilfs2 being the fastest. Both variants of BetrFS are within this range, except when a rename approaches the node size in BetrFS 0.4. The result of BetrFS 0.4 without *range-rename* shows how the performance of a naive rename implemenation with full-path indexing drops when the file size gets larger.

Figure 6(b) shows rename performance in a setup carefully designed to incur the worst-case tree surgery costs in BetrFS 0.4. In this experiment, we create two directories, each with 1,000 files of the given size. The benchmark renames the interleaved files from the source directory to the destination directory so that they are also interleaved with the files of the given size in the destination directory. Thus, when the interleaved files are 4MB or larger, every rename requires two slices at both the source and destination directories. We fsync after each rename.

Performance is roughly comparable to the previous experiment for small files. For large files, this experiment shows the worst-case costs of performing four slices. Further, all slices will operate on different leaves.

Although this benchmark demonstrates that rename performance has potential for improvement in some carefully constructed worst-case scenarios, the cost of renames in BetrFS 0.4 is nonetheless bounded to an average cost of 454ms. We also note that this line flattens, as the slicing



(a) Rename throughput as a function of file size. This experiment was performed in the base directory of an otherwise empty file system.

(b) Rename throughput as a function of file size. This experiment interleaves the renamed files with other files of the same size in both the source and destination directories.

Fig. 6. Rename throughput.

overheads grow logarithmically in the size of the renamed file. In contrast, renames in BetrFS 0.4 without *range-rename* keep dropping and are unboundedly expensive, easily getting into minutes; bounding this worst case is significant progress for the design of full-path-indexed file systems.

Recall that for small files, BetrFS 0.4 implements rename by re-inserting the file's data and metadata under the new name, i.e., a physical copy. Once the file size reaches the B^{ε}-tree node size (4MiB), BetrFS 0.4 switches to performing the rename via tree surgery. As a result, there is much less data movement, so performance jumps up to somewhere between ext4 and XFS. Overall, we conclude that rename performance in BetrFS 0.4 is comparable to other general-purpose file systems.

7.3 Full-path Performance Opportunities

As a simple example of other opportunities for full-path indexing, consider deleting an entire directory (e.g., `rm -rf`). POSIX semantics require checking permission to delete all contents, bringing all associated metadata into memory. Other directory deletion semantics have been proposed. For example, HiStar allows an untrusted administrator to *delete* a user's directory but not *read* the contents [57].

We implemented a system call that uses range-delete messages to delete an entire directory subtree. This system call, therefore, accomplishes the same goal as `rm -rf`, but it does not need to traverse the directory hierarchy or issue individual unlink/rmdir calls for each file and directory in the tree. The performance of this system call is compared to the performance of `rm -rf` on multiple file systems in Table 3. We delete the Linux 3.11.10 source tree using either our recursive-delete system call or by invoking `rm -rf`.

A recursive delete operation is orders of magnitude faster than a brute-force recursive delete on all file systems in this benchmark. This is admittedly an unfair benchmark, in that it foregoes POSIX semantics but is meant to illustrate the *potential* of range updates in a full-path-indexed

Table 3. Time to Delete the Linux 3.11.10 Source Tree (Lower Is Better)

File system	recursive delete (sec)
BetrFS 0.4 (range delete)	0.053 ± 0.001
BetrFS-0.4	3.343 ± 0.5
BetrFS-0.3	2.683 ± 0.4
BetrFS-0.4-no-rr	4.110 ± 1.0
BtrFS	2.762 ± 0.1
ext4	3.693 ± 2.2
xfs	7.971 ± 0.8
zfs	11.492 ± 0.1
nilfs2	9.472 ± 0.3

Full-path Indexing in BetrFS 0.4 can Remove a Subtree in a Single Range Delete, Orders-of-Magnitude Faster than the Recursive Strategy of `rm -rf`.

file system. With relative-path indexing, a range of keys cannot be deleted without first resolving the indirection underneath. With full-path indexing, one could directly apply a range delete to the directory and garbage collect nodes that are rendered unreachable.

There is a regression in regular `rm -rf` performance for BetrFS 0.4, making it slower than Btrfs and BetrFS 0.3. However, BetrFS 0.4 is still faster than BetrFS 0.4 without *range-rename*. This regression is attributable to inefficiencies in flushing a large number of range delete messages, which is a relatively new feature in the BetrFS code base. We found that re-enabling zones on BetrFS 0.4, even with lifting, made the performance comparable to BetrFS 0.3. An interesting side-effect of zoning is that range renames end up more evenly distributed in the tree and are flushed less often; turning off zones leads to more flushes of range delete messages. We believe this can be mitigated with additional engineering. This experiment also illustrates how POSIX semantics that require reads before writes can sacrifice performance in a write-optimized storage system.

More generally, full-path indexing has the potential to improve many recursive directory operations, such as changing permissions or updating reference counts.

7.4 Macrobenchmark Performance

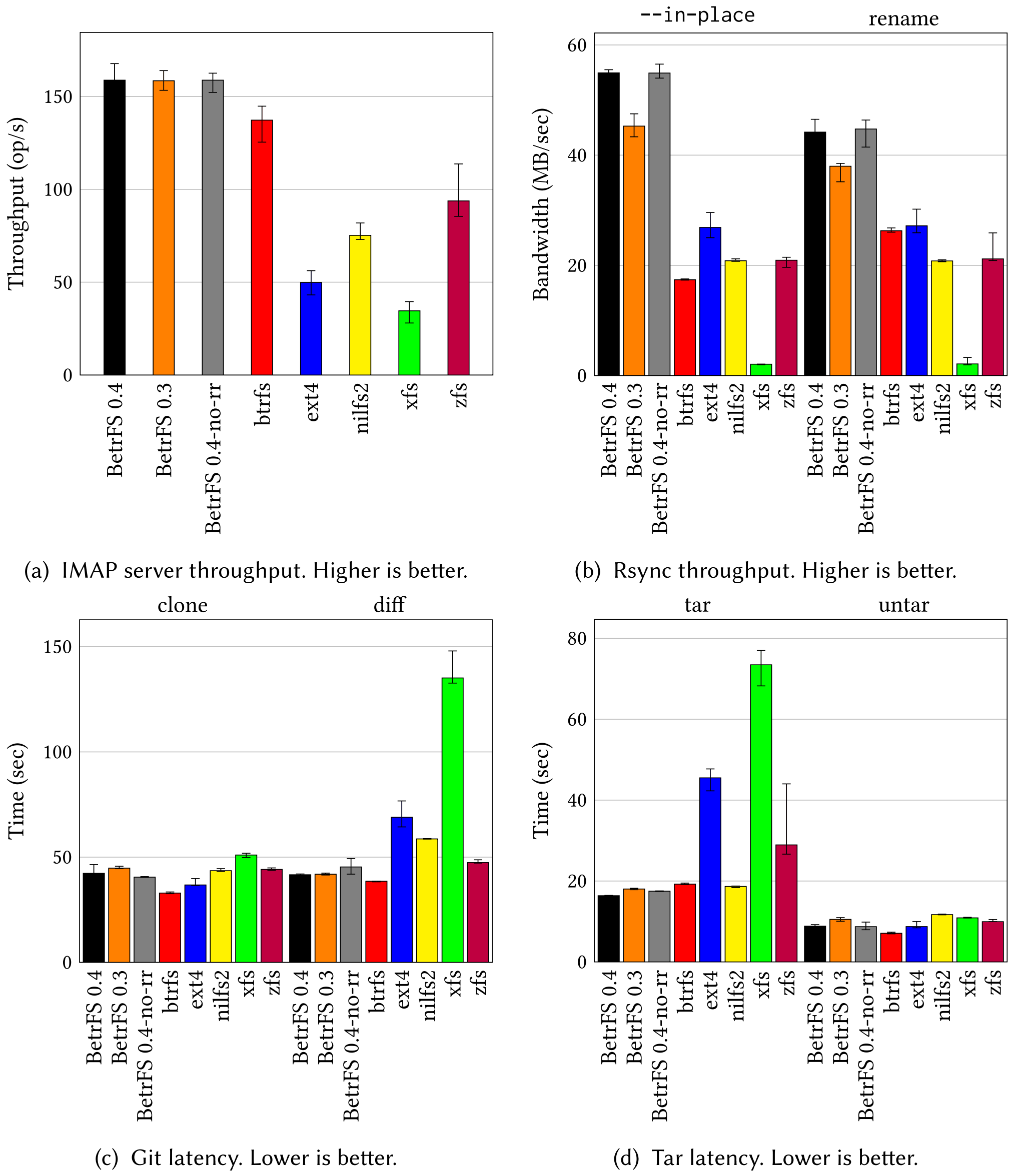
Figure 7(a) shows the throughput of four threads on the Dovecot 2.2.13 mailserver. We initialize the mailserver with 10 folders (each contains 2,500 messages) and use 4 threads (each performs 1,000 operations with 50% reads and 50% updates (marks, moves, or deletes)).

Figure 7(b) measures `rsync` performance. We copy the Linux 3.11.10 source tree from a source directory 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.

Figure 7(c) reports the time to clone the Linux kernel source code repository [30] from a clone on the local system. The `git diff` workload reports the time to diff between the v4.14 and v4.07 Linux source tags.

Finally, Figure 7(d) reports the time to `tar` and un-`tar` the Linux 3.11.10 source code.

BetrFS 0.4 is either the fastest or a close second for five of the seven application workloads. The high-level message is that performance is excellent across the board: BetrFS 0.4 is in first place or essentially tied for first place in five of the seven benchmarks. No other file system matches that breadth of performance.



(a) IMAP server throughput. Higher is better.

(b) Rsync throughput. Higher is better.

(c) Git latency. Lower is better.

(d) Tar latency. Lower is better.

Fig. 7. Application benchmarks. BetrFS 0.4 is the fastest file system, or essentially tied for fastest, in four out of the seven benchmarks. No other file system offered comparable across-the-board performance. Furthermore, BetrFS 0.4's improvements over BetrFS in the in-place rsync, git clone, and untar benchmarks demonstrate that eliminating zone-maintenance overheads can benefit real application performance.

BetrFS 0.4 represents a strict improvement over BetrFS 0.3 for these workloads. In particular, we attribute the improvement in the `rsync --in-place`, `git`, and un-`tar` workloads to eliminating zone maintenance overheads and improved sequential write performance. These results show that, although zoning represents a balance between full-path indexing and inode-style indirection, full-path indexing can improve application workloads by 3–13% over zoning in BetrFS without incurring unreasonable rename costs.

8 RELATED WORK

WODs. Write-Optimized Dictionaries, or WODs, including LSM-trees [38] and B^ε-trees [9], are widely used in key-value stores. For example, BigTable [11], Cassandra [28], LevelDB [21] and RocksDB [16] use LSM-trees; TokuDB [50] and Tucana [39] use B^ε-trees.

A number of projects have enhanced WODs, including in-memory component performance [4, 20, 45], write amplification [32, 54], and fragmentation [35]. Like the lifted B^ε-tree, the LSM-trie [54] also has a trie structure; the LSM-trie was applied to reducing write amplification during LSM compaction rather than fast key updates. Their purpose is to have distinct key ranges at each level with hashed keys so that there is a bound on write amplification, while the goal of lifted B^ε-tree is to achieve fast key updates in *range-rename*.

Several file systems are built on WODs through Filesystem in Userspace (FUSE) [18]. TableFS [42] puts metadata and small files in LevelDB and keeps larger files on ext4. KVFS [46] uses stitching to enhance sequential write performance on VT-trees, variants of LSM-trees. TokuFS [15], a precursor to BetrFS, uses full-path indexing on B^ε-trees, showing good performance for small writes and directory scans. This work was extended in BetrFS [23, 24], an in-kernel file system that uses B^ε-trees.

Trading Writes for Reads. IBM Virtual Storage Access Method (VSAM) storage system, in the Key Sequenced Data Set (KSDS) configuration, can be thought of as an early key-value store using a B+ tree. One can think of using KSDS as a full-path-indexed file system, optimized for queries. Unlike a POSIX file system, KSDS does not allow keys to be renamed, only deleted and reinserted [31].

In the database literature, a number of techniques have been developed that optimize for read-intensive workloads, but make schema changes or data writes more expensive [1–3, 12, 22, 25]. For instance, denormalization stores redundant copies of data in other tables, which can be used to reduce the costs of joins during query, but make updates more expensive. Similarly, materialized views of a database can store incremental results of queries, but keeping these views consistent with updates is more expensive. These techniques have been applied in a number of database systems

BetrFS demonstrates the advantage of WODs on random writes and also shows great locality from full-path indexing. However, full-path indexing also destroys the performance of deletes and renames. These are fixed by range-delete messages and relative-path indexing in the second version of BetrFS [55, 56]. However, relative-path indexing reduces locality and introduces maintenance costs at the file system level.

Tree Surgery. Most trees used in storage systems only modify or rebalance nodes as the result of insertions and deletions. Violent changes, such as tree surgery, are uncommon. Order Indexes [17] introduce relocation updates, which move nodes in the tree, to support dynamic indexing of hierarchical data. Ceph [52] performs dynamic subtree partitioning [53] on the directory tree to adaptively distribute metadata to different servers.

Hashing Full Paths. A number of systems store *metadata* in a hash table, keyed by full path, to look up metadata in one I/O. The Direct Lookup File System (DLFS) maps file metadata to on-disk buckets by hashing full paths [29]. Hashing full paths creates two challenges: files in the same directory may be scattered across disk, harming locality, and DLFS directory renames require deep recursive copies of both data and metadata.

A number of distributed file systems have stored file metadata in a hash table, keyed by full path [19, 40, 49]. In a distributed system, using a hash table for metadata has the advantage of easy load balancing across nodes, as well as fast lookups. We note that the concerns of indexing *metadata* in a distributed file system are quite different from keeping logically contiguous *data*

physically contiguous on disk. Some systems, such as the Google File System, also do not support common POSIX operations, such as listing a directory.

CalvinFS [49] is a replicated file system that stores its metadata in a distributed database. Like DLFS, CalvinFS indexes files by full path, hash-partitioning file metadata for load balancing and to simplify data tracking. This use of hashing similarly eliminates the locality benefits of full-path indexing and makes multi-file operations like recursive renames costly. CalvinFS is optimized for single-file operations, and multi-file operations like recursive renames are implemented as compound transactions.

In order to speed up lookups in the virtual file system's directory cache, Tsai et al. [51] demonstrate that indexing the *in-memory* kernel directory cache by full paths can improve path lookup operations, such as open. This work observes similar overheads as this article from comparing long keys; rather than use lifting, their proposed directory cache uses a collision-resistant hash for faster comparisons of hashes of long keys. Since the directory cache is an in-memory mapping from file names to inodes, the locality of file data is not a concern. Nonetheless, hashing full paths introduces many challenges for multi-file operations that, through hashing, are scattered throughout the table.

To compactly store an index of keywords, de la Briandais [14] proposed the *trie* data structure (originally referred to as the "letter tables" approach) [14]. Descending from the root, keywords are built component-by-component, with references to values stored at the keyword's completion. BetrFS 0.4 lifting uses this strategy, storing full-path keys in a lifted B^{ε}-tree. Each node of the lifted B^{ε}-tree stores the common substring shared by its descendents' keys. By decomposing full paths into substring components stored in nodes, BetrFS 0.4 minimizes the keyspace modifications required to complete a rename and preserves locality.

9 CONCLUSION

This article presents a new on-disk indexing structure, the lifted B^{ε}-tree, which can leverage full-path indexing without incurring large rename overheads. Our prototype, BetrFS 0.4, is a nearly strict improvement over BetrFS 0.3. The main cases where BetrFS 0.4 does worse than BetrFS 0.3 are where node splitting and merging is on the critical path, and the extra computational costs of lifting harm overall performance. We believe these costs can be reduced in future work.

BetrFS 0.4 demonstrates the power of consolidating optimization effort into a single framework. A critical downside of zoning is that multiple, independent heuristics make independent placement decisions, leading to sub-optimal results and significant overheads. By using the keyspace to communicate information about application behavior, a single codebase can make decisions such as when to move data to recover locality, and when the cost of indirection can be amortized. In future work, we will continue exploring additional optimizations and functionality unlocked by full-path indexing.

Source code for BetrFS is available at betrfs.org.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Ethan Miller, for their insightful comments on earlier drafts of the work. Part of this work was done while Yuan was at Farmingdale State College of SUNY.

REFERENCES

[1] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endow.* 5, 10 (2012), 968–979.

[2] Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *Proc. VLDB Endow.* 2, 2 (2009), 1566–1569.

[3] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. 2003. STREAM: The Stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. 665–665.

[4] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*. 80–94.

[5] Michael A. Bender, Richard Cole, Erik D. Demaine, and Martin Farach-Colton. 2002. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02)*. 139–151.

[6] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious streaming B-trees. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*. 81–92.

[7] 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^{ε}-trees and write-optimization. *:login; Magazine* 40, 5 (Oct. 2015), 22–28.

[8] Gerth Stølting Brodal, Erik D. Demaine, Jeremy T. Fineman, John Iacono, Stefan Langerman, and J. Ian Munro. 2010. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'10)*. 1448–1456.

[9] Gerth Stolting Brodal and Rolf Fagerberg. 2003. Lower bounds for external memory dictionaries. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*. 546–554.

[10] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. 2000. On external memory graph traversal. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*. 859–860.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.* 26, 2 (2008), 4:1–4:26.

[12] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A. N. Soules, and Alistair Veitch. 2012. LazyBase: Trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. 169–182.

[13] 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 (FAST'17)*. 45–58.

[14] Rene De La Briandais. 1959. File searching using variable length keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference (IRE-AIEE-ACM'59 (Western))*. 295–298.

[15] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2012. The TokuFS streaming file system. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'12)*. 14–14.

[16] Facebook, Inc.RocksDB. Retrieved April 26, 2018 from http://rocksdb.org/.

[17] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Faerber. 2015. Indexing highly dynamic hierarchical data. *Proc. VLDB Endow.* 8, 10 (2015), 986–997.

[18] FUSE. Retrieved April 26, 2018 from https://github.com/libfuse/libfuse.

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 29–43.

[20] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*. 32:1–32:14.

[21] Google, Inc.LevelDB. Retrieved April 26, 2018 from https://github.com/google/leveldb.

[22] Mingsheng Hong, Alan J. Demers, Johannes E. Gehrke, Christoph Koch, Mirek Riedewald, and Walker M. White. 2007. Massively multi-query join processing in publish/subscribe systems. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. 761–772.

[23] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 301–315.

[24] 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: Write-optimization in a kernel file system. *ACM Trans. Storage* 11, 4 (2015), 18:1–18:29.

[25] Charles Johnson, Kimberly Keeton, Charles B. Morrey, Craig A. N. Soules, Alistair Veitch, Stephen Bacon, Oskar Batuner, Marcelo Condotta, Hamilton Coutinho, Patrick J. Doyle, Rafael Eichelberger, Hugo Kiehl, Guilherme Magalhaes, James McEvoy, Padmanabhan Nagarajan, Patrick Osborne, Joaquim Souza, Andy Sparkes, Mike Spitzer,

Sebastien Tandel, Lincoln Thomas, and Sebastian Zangaro. 2014. From research to practice: Experiences engineering a production metadata database for a scale out file system. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. 191–198.

[26] Sangman Kim, Michael Z. Lee, Alan M. Dunn, Owen S. Hofmann, Xuan Wang, Emmett Witchel, and Donald E. Porter. 2012. Improving server applications with system transactions. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. 15–28.

[27] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. 2006. The Linux implementation of a log-structured file system. *SIGOPS Operating Systems Review* 40, 3 (2006), 102–107.

[28] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2 (2010), 35–40.

[29] Paul Hermann Lensing, Toni Cortes, and André Brinkmann. 2013. Direct lookup and hash-based metadata placement for local file systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*. 5:1–5:11.

[30] Linux kernel source tree. Retrieved April 26, 2018 from https://github.com/torvalds/linux.

[31] Mary Lovelace, Jose Dovidauskas, Alvaro Salla, and Valeria Sokai. 2004. VSAM Demystified. (2004). Retrieved April 26, 2018 from http://www.redbooks.ibm.com/abstracts/sg246105.html.

[32] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 133–148.

[33] Avantika Mathur, MingMing Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: Current status and future plans. In *Ottowa Linux Symposium (OLS)*, Vol. 2. 21–34.

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

[35] Fei Mei, Qiang Cao, Hong Jiang, and Lei Tian Tintri. 2017. LSM-tree managed storage for large-scale key-value store. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. 142–156.

[36] Jason Olson. 2007. Enhance your apps with file system transactions. *MSDN Magazine* (July 2007). http://msdn2.microsoft.com/en-us/magazine/cc163388.aspx.

[37] ZFS on Linux. Retrieved April 26, 2018 from http://zfsonlinux.org/.

[38] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (1996), 351–385.

[39] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of the 2016 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'16)*. 537–550.

[40] Christopher Peery, Francisco Matias Cuenca-Acuna, Richard P. Martin, and Thu D. Nguyen. 2005. Wayfinder: Navigating and sharing information in a decentralized world. In *Proceedings of the Second International Conference on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P'04)*. 200–214.

[41] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. 2009. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. 161–176.

[42] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. 145–156.

[43] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The linux B-tree filesystem. *ACM Trans. Storage* 9, 3 (2013), 9:1–9:32.

[44] Russell Sears, Mark Callaghan, and Eric Brewer. 2008. Rose: Compressed, log-structured replication. *Proc. VLDB Endow.* 1, 1 (2008), 526–537.

[45] 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)*. 217–228.

[46] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. 17–30.

[47] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. 2009. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the 7th Conference on File and Storage Technologies (FAST'09)*. 29–42.

[48] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (ATEC'96)*. 1–1.

[49] Alexander Thomson and Daniel J. Abadi. 2015. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 1–14.

[50] Tokutek, Inc.TokuDB. Retrieved April 26, 2018 from https://github.com/Tokutek/ft-index.

[51] 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 (SOSP'15)*. 441–456.

[52] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 307–320.

[53] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. 2004. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*. 4–15.

[54] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'15)*. 71–82.

[55] 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. 2017. Writes wrought right, and other adventures in file system optimization. *ACM Trans. Storage* 13, 1 (2017), 3:1–3:26.

[56] 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 *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 1–14.

[57] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 263–278.

Received May 2018; revised July 2018; accepted July 2018