# EFFICIENT, LOCALITY-MAINTAINING NAMESPACE OPERATIONS IN A WRITE-OPTIMIZED FILE SYSTEM

Yang Zhan

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2019

Approved by:

Donald E. Porter

Leonard McMillan

F. Donelson Smith

Rob Johnson

Ethan L. Miller

## ABSTRACT

Yang Zhan: Efficient, Locality-Maintaining Namespace Operations in a Write-Optimized File System
(Under the direction of Donald E. Porter)

There is a long-standing trade-off between good locality (fast directory traversals) and efficient namespace operations (efficient file or directory renames) in file systems. Traditional inode-based file systems have good rename performance but can fail to maintain locality, especially in the face of file system aging. On the other hand, full-path-indexed file systems ensure locality, however, renaming a directory needs to update all related full-paths, which is usually implemented as an expensive operation. No existing file system has both good locality and efficient namespace operations.

This dissertation describes a new file system design that has both good locality and efficient namespace operations. In particular, we describe a novel synthesis of write-optimization, full-path indexing, and operations on data structures. By directly manipulating the data structure, a full-path-indexed file system can efficiently update related full-paths in a rename. Moreover, with the technique, a full-path-indexed file system can clone a directory without traversing the directory.

We implement this technique in BetrFS, a full-path-indexed, write-optimized, local file system for Linux. Compared to ext4, the widely used inode-based file system in Linux, the new version of BetrFS traverses the Linux source directory 9.47x faster and renames the same directory 1.09x faster. Meanwhile, the new version of BetrFS clones a directory faster than state-of-the-art file systems that support clones, such as Btrfs and XFS.

## ACKNOWLEDGEMENTS

I would like to thank my advisor, Donald E. Porter. During the past six years in the OSCAR Lab, I have received a great deal of support from him. He has invested enormous time and energy to guide me in the world of system reasearch.

I would also like to thank my committee members, Leonard McMillan, F. Donelson Smith, Rob Johnson, and Ethan L. Miller. I am grateful for all their help and insightful suggestions to this dissertation.

Many thanks to professors and students in the amazing BetrFS project. The project would not have been so successful without them. Special thanks go to William Jannen and Jun Yuan, who laid the foundation of the BetrFS implementation.

In addition, I would like to thank my friends in Stony Brook and Chapel Hill. They supported me greatly and were always willing to help me. Life would have been much harder without them.

Finally, I would like to thank my parents for their everlasting support.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| CoW | Copy-on-Write |
| DAG | Directed Acyclic Graph |
| LCA | Lowest Common Ancestor |
| LCP | Longest Common Prefix |
| VFS | Virtual File System |
| WOD | Write-Optimized Dictionary |

# CHAPTER 1: INTRODUCTION

Today's general purpose file systems fail to utilize the full bandwidth of the underlying hardware. Widely used inode-based file systems, such as ext4, XFS, and Btrfs, can write large files at near disk bandwidth, but typically create small files at less than 3% of the disk bandwidth [50]. Similarly, these file systems can read large files at near disk bandwidth, but traversing directories with many small files is slow, and the performance degrades when the file system ages [10].

At the heart of this issue is how file systems organize metadata and data on disk. The most common design pattern for modern file systems is to use multiple layers of indirection. The inode number of a file or directory connects the name of an entry in the parent directory to its metadata location on disk. The metadata of an inode contains extents that describe the physical location of data at different offsets. Indirection simplifies the implementation of file systems. However, indirection doesn't impose any constraints on how metadata and data are placed on disk. In the worst case, the metadata of entries under a directory and the data of a file can end up scattered over the disk. Because random I/Os are much slower than sequential I/Os on disks, directory traversals and file creations can be extremely slow. Heuristics, such as cylinder groups [26], are designed to mitigate this problem. However, after disk space is allocated and freed over file system aging, the free space on disk becomes scattered, making such heuristics ineffective in the worst case.

One attempt to solve the problem of file creations is the log-structured file system [37]. The log-structured file system treats the whole disk as a log, and all write operations, including file creations, become log appends, which are written to the underlying disk sequentially. Therefore, file creations are much faster on the log-structured file system. However, the log cleaner in a log-structured file system has severe impact on performance [39], especially when the log is full.

Alternatively, a file system can put metadata and data in write-optimized dictionaries (WODs). WODs, such as the LSM-tree [29], consist of multiple levels whose sizes grow exponentially. Writes are treated as log appends to the lowest level, and gradually merged to higher levels in batches. Because all data are written in batches, the amortized cost of each write is much smaller in WODs, despite the fact that each write is written multiple times. Thus, file creations are much faster on WOD-based file systems, such as TableFS [34].

However, previous WOD-based file systems are still inode-based. In other words, these file systems use inode numbers as indexes for metadata and data. Because the inode number of a file or directory doesn't change once allocated, after file deletions and creations over file system aging, files in the same directory can be assigned with inode numbers that are irrelevant to each other in the worst case. In such a scenario, the file system is unable to group the inodes in the same directory close to each other in the WOD. Therefore, a directory scan in inode-based, WOD-based file system can still result in many random I/Os in the worst case.

An alternative design is to use full-path indexing upon WODs in a file system, known to have good performance on nearly all operations. A full-path-indexed file system uses the full-path name of a file or directory as the key for associated metadata and data, and sorted the full-path names in depth-first-search order, that is, lexicographic order by the full-path names of files and directories. With full-path indexing, metadata and data under one directory are close to each other in the key space, which, combined with the sorted order maintained by WODs, leads to good locality and fast directory traversals. Prior work [10, 20, 21, 50, 51] of this design realizes efficient implementation of many file-system operations, such as random writes, file creations and directory traversals, but a few operations have prohibitively high overheads.

The Achilles' heel of full-path indexing is the performance of namespace operations, in particular, renaming large files and directories. In inode-based file systems, renaming a file or directory is just a pointer swing, moving an entry from one directory to another directory, without touching the file or directory being renamed. However, with full-path indexing, renaming a directory involves changing the full-path names of all files and directories under it, updating keys of

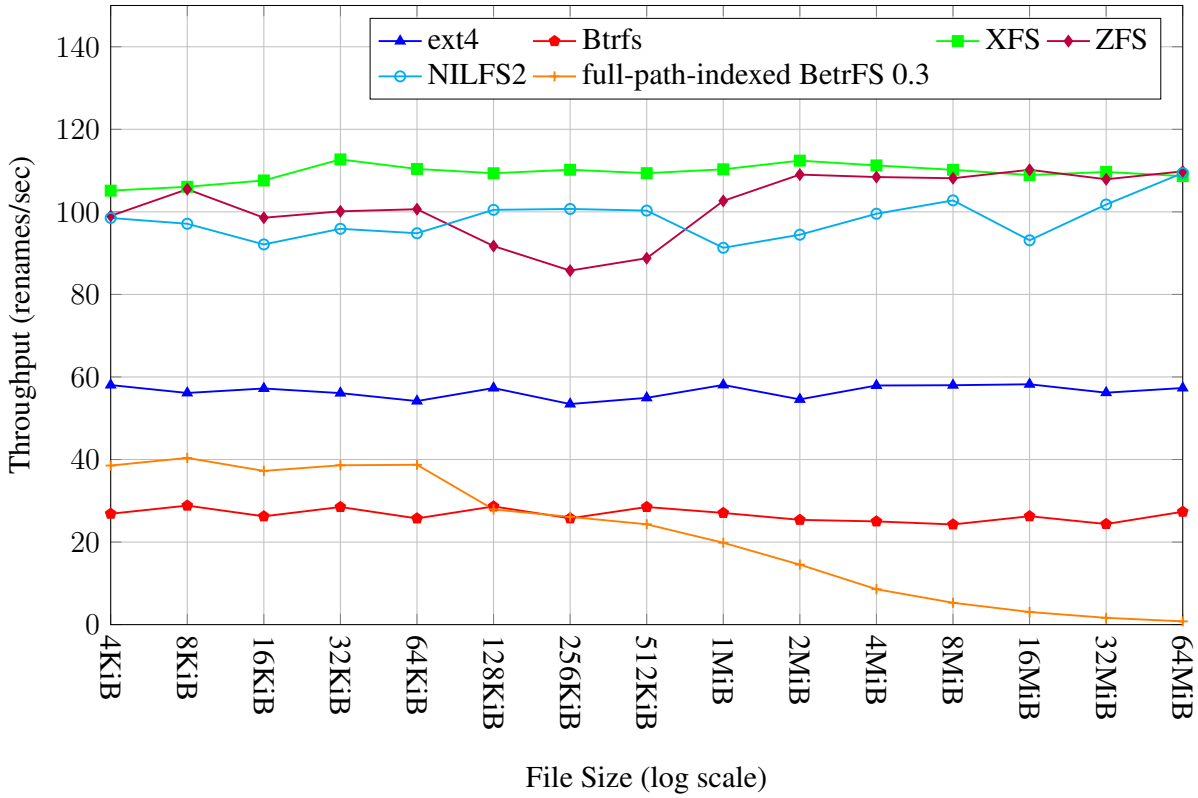



Figure 1.1: Throughput of renaming a file of different sizes (higher is better). The performance of full-path-indexed BetrFS 0.3 degrades when the file being renamed becomes larger.

the metadata and data. Competitive performance for namespace operations in full-path-indexed file systems should complete in an I/O-efficient manner.

However, prior work mainly focuses on the schema level of the file system, i.e., how metadata and data are keyed and indexed in WODs. Renames on those full-path-indexed file systems are implemented by fetching all related key/value pairs of metadata and data, inserting them back with updated keys, and deleting old key/value pairs. In such a design, a file system rename needs to call several operations for each affected full-path name, leading to bad performance, especially when the file or directory being renamed is large. Figure 1.1 compares the throughput of file renames in different file systems. The benchmark renames a file of different sizes 100 times, each followed by an `fsync` of the parent directory. Except full-path-indexed BetrFS 0.3 [1], all file systems are inode-based. As the file size changes, these inode-based file systems have rela-

[1] BetrFS 0.3 [10] is the latest version of BetrFS and can be configured to use full-path-indexing (default BetrFS 0.3 is not full-path-indexed).

tively stable rename throughput. In contrast, file renames in full-path-indexed BetrFS 0.3 become slower and slower when the file size grows, because the file system needs to update more key/-value pairs (full-path-indexed BetrFS 0.3 stores each 4KiB block of a file as a key/value pair, which uses the full-path name and the block number and the key).

This dissertation presents I/O-efficient implementations of namespace operations in a full-path-indexed, write-optimized file system. Specifically, though full-path indexing limits possible changes on the schema level, we observe that it can make all full-path names under a directory contiguous in the key space. And the underlying WODs, $B^\varepsilon$-trees, have a tree structure [2], which makes it possible to move a contiguous key range in a subtree efficiently. Therefore, we dig into the underlying $B^\varepsilon$-trees, and implement a new operation, range-rename, that completes file system renames with a bounded number of I/Os. Moreover, because of the contiguity in the key space, namespace operations that are difficult in inode-based file systems are easier in full-path-indexed file systems. In particular, we expand the range-rename operation into the range-clone operation, which can complete file or directory clones efficiently.

The primary contribution of this dissertation is to show that there is no trade-off between efficient namespace operations and locality. One can implement efficient renames in a full-path-indexed, write-optimized file system while keeping the locality ensured by full-path indexing. In fact, with full-path indexing, one can implement namespace operations that are difficult in inode-based file systems, such as directory clones.

Chapter 2 discusses previously published work related to this work. This chapter organizes related work by topic, and talks in detail about work that is closely related to this work.

Chapter 3 describes the necessary background of this dissertation. This chapter starts with a presentation of the write-optimized $B^\varepsilon$-trees, showing the idea of write-optimization. Then, the chapter describes full-path-indexed BetrFS and relative-path-indexed BetrFS. The full-path-indexed BetrFS shows the benefit of full-path indexing and write-optimization, but suffers from

[2] We choose $B^\varepsilon$-trees in our implementation, but the technique can be used on other WODs, such as LSM-trees, as long as the data structure organizes data in a tree structure.

slow renames. The relative-path-indexed BetrFS has good rename performance, but breaks the full-path indexing and taxes other operations for efficient renames.

Chapter 4 presents the range-rename operation on $B^\varepsilon$-trees. All key/value pairs in the $B^\varepsilon$-tree whose keys have a certain prefix are updated to have keys with another prefix after the range-rename operation. And the range-rename operation is implemented efficiently through two techniques, **key lifting** and **tree surgery**. And with efficient range-rename, full-path-indexed BetrFS can implement file system renames efficiently.

Chapter 5 expands the range-rename operation into the range-clone operation. This chapter first shows how to implement the range-clone operation with range-rename techniques by transforming $B^\varepsilon$-trees into $B^\varepsilon$-DAGs. Then, the chapter introduces a new type of messages to $B^\varepsilon$-DAGs, `GOTO` messages. The `GOTO` message works like other messages in $B^\varepsilon$-DAGs, fitting the range-clone operation into write-optimization. Full-path-indexed BetrFS can use the range-clone operation to implement both file or directory renames and clones.

Chapter 6 evaluates the implementation. This chapter compares full-path-indexed BetrFS with range-rename or range-clone to widely used file systems on micro and application benchmarks. The chapter also puts a particular focus on benchmarking namespace operations.

Chapter 7 summarizes and concludes the dissertation.

# CHAPTER 2: RELATED WORK

This chapter discusses related work to differentiate the contribution of this dissertation over previous research on write-optimization, full-path indexing in file systems, tree surgery and clones in file systems.

## 2.1 Write-optimization

**Write-optimized dictionaries (WODs).** The most commonly used WOD is the log-structured merge tree (LSM-tree) [29]. The LSM-tree partitioned key/value pairs into levels whose sizes grew exponentially. Each level in the LSM-tree was an B-tree. The LSM-tree put new key/value pairs into the first level. When a level was full, the LSM-tree merged all key/value pairs in the level into the next level.

Nowadays, most LSM-tree implementations use the design of LevelDB [18]. Instead of using a tree to store key/value pairs in one level, LevelDB divides a level into multiple 4MiB SSTables and keeps the key range of each SSTable in the metadata file. When a level is full, LevelDB picks an SSTable in the level and merges the SSTable with SSTables in the next level.

There were also cache-oblivious WODs. The cache-oblivious lookahead array (COLA) [2] stored each level in an array and used fractional cascading [8] to improve query performance. The xDict [5] was a cache-oblivious write-optimized dictionary with asymptotic behavior similar to a $B^\varepsilon$-tree. However, only prototypes exist for cache-oblivious WODs.

We use write-optimized $B^\varepsilon$-trees [6] in our implementation. Compared to LSM-trees, $B^\varepsilon$-trees put all key/value pairs in one tree and have better asymptotic query performance [3].

**Key/value stores.** Many key/value stores use WODs as the underlying data structures. For example, BigTable [7], Cassandra [23], HBase [1], LevelDB [18] and RocksDB [14] implement LSM-trees, while *ft-index* [43] and Tucana [30] implement $B^{\varepsilon}$-trees.

Recently, many papers focused on optimizing different aspects of the LSM-tree. LOCS [46] optimized LSM-trees for multi-channel SSDs. WiscKey [25] reduced the size of the LSM-tree so that the LSM-tree fitted into the in-memory cache. PebblesDB [33] optimized the implementation of LSM-trees in LevelDB to reduce write-amplification.

In this dissertation, we provide the upper level file system with key/value store operations. Specifically, for a namespace operation, the file system calls a key/value store operation that finishes most of the work efficiently.

**File systems.** The log-structured file system [37] treated the whole disk as a log. Thus, it performed writes much faster than other file systems. However, garbage collection could be expensive in the file system [39].

TokuFS [13] was an in-application library file system, built on top of *ft-index*. TokuFS showed that a write-optimized file system could support efficient write-intensive and scan-intensive workloads.

KVFS [40] was based on a transactional variation of the LSM-tree, called the VT-tree. Impressively, the performance of their transactional file system was comparable to the performance of the ext4 file system, which does not support transactions. One performance highlight was on random-writes, where they outperformed ext4 by a factor of 2. They also used stitching to perform well on sequential I/O in the presence of LSM-tree compaction.

TableFS [34] used LevelDB to store file-system metadata. They showed substantial performance improvements on metadata-intensive workloads, sometimes up to an order of magnitude. They used ext4 as an object store for large files, so sequential IO performance was com-parable to ext4.

All the WOD-based file systems above were built on FUSE [16], in which the authors can write file systems in userspace. However, FUSE imposes expensive context-switch costs on the

file system. The TableFS paper analyzed the FUSE overhead relative to a library implementation of their file system and found that FUSE could cause a 1000 increase in disk-read traffic.

BetrFS [10, 20, 21, 50, 51] was the first WOD-based file system in kernel. Because of the underlying WOD, BetrFS performed well on benchmarks with small, random writes.

The implementation in this dissertation is a newer version of BetrFS, which inherits the good random write performance from older versions and has a new design for namespace operations.

## 2.2 Full-path indexing in file systems

A number of systems store metadata in a hash table, keyed by full-paths, to look up metadata in one IO. The Direct Lookup File System(DLFS) mapped file metadata to on-disk buckets by hashing full-paths [24]. Hashing full-paths created two challenges, files in the same directory might be scattered across disk, harming locality and DLFS directory renames required 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-paths [17, 31, 42]. In a distributed system, using a hash table for metadata has the advantage of easy load balancing across nodes, as well as fast lookups 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.

The dcache optimization proposed by Tsai et al. demonstrated that indexing the in-memory kernel cache by full-path names can improve several lookup operations, such as `open` [44]. In their system, a rename needed to invalidate all related caches.

The first version of BetrFS [20, 21] used full-path indexing and renames in the first version of BetrFS could be very slow. Therefore, later versions of BetrFS [10, 50, 51] used relative-path indexing. Relative-path indexing divided the directory tree into zones and used full-path indexing within a zone. Relative-path indexing showed good rename performance but penalized other operations.

This dissertation brings full-path indexing back to BetrFS and implements efficient file system renames by doing tree surgery on $B^\varepsilon$-trees.

## 2.3 Tree surgery

Most trees used in storage systems only modify or rebalance node as the result of insertions and deletions. Operations, such as slicing out or relocating a subtree in tree surgery, are uncommon.

Finis et al. [15] introduced ordered index for handling hierarchical data, such as XML, in relational database systems. Similar to a file system rename that moves a subtree in the directory hierarchy, one of their task was moving a subtree in the hierarchical data. However, in their use cases, hierarchy indexes were generally secondary indexes, while in BetrFS, full-path keys encode the directory hierarchy.

Ceph [47] used dynamic subtree partitioning [48] to load balancing metadata servers. In Ceph, a busy metadata server would delegate subtrees of its workload to other metadata servers. Compared to namespace operations of this dissertation, the dynamic subtree partitioning in Ceph didn't change the overall directory hierarchy of the file system.

## 2.4 Clones in file systems

We also investigate file or directory clones in this dissertation. Table 2.1 summarizes the support of clones in file systems.

**File systems with snapshots.** Many modern file systems provide snapshot mechanism, making read-only copies of the whole file system.

The WAFL file system [19] organized all blocks in a tree structure. By copying the "vol_info" block, which was the root of the tree structure, WAFL created a snapshot. Later, WAFL introduced a level of indirection between the file system and the underlying disks [12]. Therefore, multiple active instances of the file system could exist at the same time and WAFL could create

| File System | Whole File System Clone | File Clone | Directory Clone |
|---|---|---|---|
| WAFL [12, 19] | Writable | No | No |
| FFS [26, 27] | Read-only | No | No |
| NILFS2 [22] | Read-only | No | No |
| ZFS [4] | Read-only | No | No |
| GCTree [11] | Read-only | No | No |
| NOVA-Fortis [49] | Read-only | No | No |
| The Episode file system [9] | No | No | Writable* |
| Ext3cow [32] | No | Writable | No |
| Btrfs [35, 36] | Writable | Writable | Writable* |
| EFS [38] | No | Read-only | Read-only |
| CVFS [41] | No | Read-only | Read-only |
| Versionfs [28] | No | Read-only | Read-only |
| BetrFS | Writable | Writable | Writable |

Table 2.1: Clones in file systems (* to enable directory clones, users need to mark the directory as a subvolume (Btrfs) or a fileset (the Episode file system) before putting anything to the directory).

writable snapshots of the whole file system by creating a new instance and copying the vol_info block.

FFS [26, 27] created snapshots by suspending all operations and creating a snapshot file whose inode was stored in the superblock. The size of the snapshot file was the same as the disk. Upon creation, most block pointers in the snapshot inode were marked "not copied" or "not used" while some metadata blocks were copied to new addresses. Reading a "not copied" address in the snapshot file resulted in reading the address on the disk. When a "not copied" block was modified in the file system, FFS copied the block to a new address and updated the block pointer in the snapshot inode.

NILFS [22] was a log-structured file system that organizes all blocks in a B-tree. In NILFS, each logical segment contained modified blocks and a checkpoint block, which was used as the root of the B-tree. NILFS got the current view of the file system from the checkpoint block of the last logical segment. NILFS could create a snapshot by making a checkpoint block permanent.

ZFS [4] also stored the file system in a tree structure and created snapshots by copying the root of the tree (uberblock). To avoid maintaining one block bitmap for each snapshot, ZFS

kept birth time in each pointer. A block should not be freed if its birth time was earlier than any snapshot. In that case, the block was added to the dead list of the most recent snapshot. When a snapshot was deleted, all blocks in its dead list were checked again before being freed.

GCTree [11] implemented snapshots on top of ext4 by chaining different versions of a metadata block with GCTree metadata. Also, each pointer in the metadata block had a "borrowed bit" indicating whether the target block was inherited from the previous version. Therefore, GCTree could check whether to free a block by inspecting GCTree metadata and didn't need to maintain reference counts.

NOVA-Fortis [49] was designed for non-volatile main memory (NVMM). In NOVA-Fortis, each inode had a private log with log entries pointing to data blocks. To enable snapshots, NOVA-Fortis kept a global snapshot ID and added the creating snapshot ID to log entries in inodes. NOVA-Fortis decided whether to free a block based on the snapshot ID in the log entry and active snapshots. NOVA-Fortis also dealt with DAX-style `mmap` by stalling page faults when marking all pages read-only.

Snapshots in file systems provide the basic functionality of cloning files and directories in a coarse granularity. However, users may want to clone only a certain file or directory, in which cloning the whole file system can waste time and space.

**File systems with file or directory clones.** Some file systems go further to support cloning one single file or directory.

The Episode file system [9] grouped everything under a directory into a fileset. Episode could create an immutable fileset clone by copying all the anodes (inodes) and marking all block pointers in the anodes copy-on-write. When modifying a block with a copy-on-write pointer, Episode allocated a new block and updated the block pointer in the active fileset.

Ext3cow [32] created immutable file clones by maintaining a system-wide epoch and inode epochs. When an inode was modified, ext3cow allocated a new inode if the epoch in the inode was older than the last snapshot epoch. Also, each directory stored birth and death epochs for

each entry. Ext3cow could render the view of the file system in a certain epoch by fetching entries alive at that epoch.

Btrfs [36] supported creating writable snapshot of the whole file system by copying the root of the sub-volume tree in its COW friendly B-trees [35]. Btrfs cloned a file by sharing all extents of the file. The extent allocation tree recorded extents with back pointers to the inodes. Therefore, Btrfs was able to move an extent at a later time.

All existing file systems either don't support directory clones or require pre-configuration for directory clones. Or, a file system with file clones can clone a directory by traversing the directory and cloning each file, which can be costly if the directory is huge.

**Versioning file systems** There are also versioning file systems that versions files and directories.

EFS [38] automatically versioned files and directories. By allocating a new inode, EFS created and finalized a new file version when the file was opened and closed. Each versioned file had an inode log that keeps all versions of the file. All entries in directories had creation and deletion time.

CVFS [41] tried to store metadata versions more compactly. CVFS suggested two ways to save space in versioning file systems: 1. journal-based metadata that kept a current version and an undo log to recover previous versions; 2. multiversion B-trees that kept all versions of metadata in a B-tree.

Versionfs [28] built a stackable versioning file system. Instead of building a new file system, Versionfs added the functionality of versioning on an existing file system by transferring certain operations to other operations. In this way, all versions of a file were maintained as different files in the underlying file system.

Versioning file systems study how to automatically provide file or directory clones for users, while this dissertation focuses on how to implement file or directory clones in a full-path-indexed file system.

# CHAPTER 3: BACKGROUND

This chapter gives the background of BetrFS [10, 20, 21, 50, 51]. BetrFS is a file system based on write-optimized B$^\varepsilon$-trees. Section 3.1 introduces the underlying data structure in BetrFS, the B$^\varepsilon$-tree. The B$^\varepsilon$-tree is a write-optimized dictionary (WOD) that treats writes as messages and cascades messages in batches. Then, Section 3.2 describes full-path-indexed BetrFS. Full-path indexing keeps metadata and data under one directory close to each other in the key space, but makes the simple implementation of file system renames expensive. Next, Section 3.3 discusses relative-path indexing, the previous approach that penalizes other operations to mitigate the rename issue in BetrFS. Finally, Section 3.4 explains implementation details in BetrFS.

## 3.1 B$^\varepsilon$-trees

B$^\varepsilon$-trees [3, 6] are B-trees, augmented with buffers in non-leaf nodes. New writes are injected as messages into the buffer of the root node of a B$^\varepsilon$-tree. When a node's buffer becomes full, messages are flushed from that node's buffer to one of its children. If the child is a non-leaf node, messages are injected into the child's buffer. Otherwise, messages take effect on the leaf node. An insert message becomes a key/value pair in the leaf node. If there is an old key/value pair with the same key in the leaf node, the insert message overwrites the old/key value pair. A delete message removes the old key/value pair with the same key in the leaf node. Therefore, in a B$^\varepsilon$-tree, leaf nodes only store key/value pairs, as in a B-tree. Because writes can be messages in non-leaf nodes, point and range queries on a B$^\varepsilon$-tree must check the buffers along the root-to-leaf path, as well as key/value pairs in leaf nodes.

Figure 3.1 shows a B$^\varepsilon$-tree example. Node $D$, $E$, $F$, $G$ and $H$ are leaf nodes that store key/-value pairs. Each leaf node shows key/value pairs in rows, and each row represents a key/value

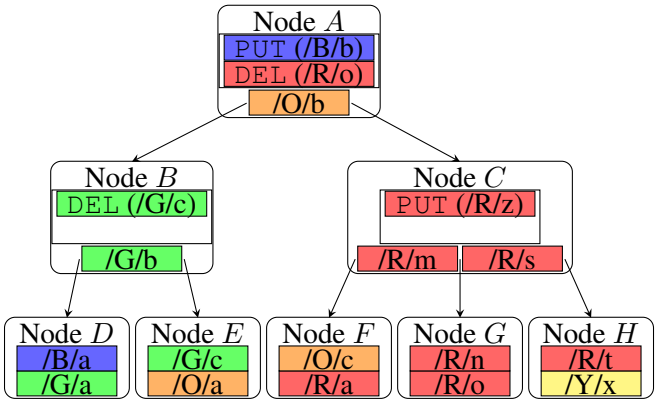



Figure 3.1: In a B$^\varepsilon$-tree, a leaf nodes stores key/value pairs, while a non-leaf node has pivots, parent-to-child pointers and a message buffer. We use different colors to indicate keys with different prefixes throughout the dissertation.

pair, with the left part as the key and the right part as the value. We use different colors to indicate keys with different prefixes and omit the content of the value as dots throughout the dissertation. Node $A$, $B$ and $C$ are non-leaf nodes that contain pivots, parent-to-child pointers and messages in buffers. Parent-to-child pointers connect all nodes into a tree, and the left and right pivots of a parent-to-child pointer bound the key range of the child node. Messages in the buffers of non-leaf nodes are write operations that have been applied to the B$^\varepsilon$-tree but have not taken effect on leaf nodes. For example, the message `DEL` ("/G/c") in Node $B$ indicates that the key "/G/c" has been deleted. Therefore, though a query for "/G/c" can find the key/value pair with the key in Node $E$, that key/value pair is invalidated by the message in Node $B$ and the query returns `NOT_FOUND`. Likewise, a query for "/R/z" returns the value in the message `PUT` ("/R/z") in node $C$.

There can be multiple messages with the same key in a B$^\varepsilon$-tree. For example, in Figure 3.2, the user first inserts a key/value pair with key "/R/z" and then deletes the key "/R/z". This operation generates two messages, `DEL` ("/R/z") in Node $A$ and `PUT` ("/R/z") in Node $C$. Because `DEL` ("/R/z") comes after `PUT` ("/R/z"), the node that contains `DEL` ("/R/z") is closer to the root of the B$^\varepsilon$-tree than the node that contains `PUT` ("/R/z"). Therefore, `PUT` ("/R/z") will be applied to the leaf node before `DEL` ("/R/z") and queries will apply `DEL` ("/R/z") after applying `PUT` ("/R/z") (so queries will not observe any key/value pair with key "/R/z").

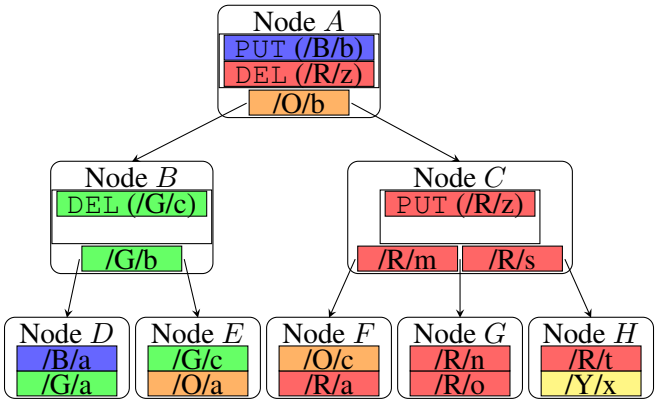



Figure 3.2: There can be multiple messages with the same key in a B$^\varepsilon$-tree. Messages that are closer to the root of the B$^\varepsilon$-tree are newer, and thus override older messages.

| Data Structure | Insert | Point Query | Range Query |
|---|---|---|---|
| B-tree | $O(log_B N)$ | $O(log_B N)$ | $O(log_B N + k/B)$ |
| B$^\varepsilon$-tree | $O(log_B N/\varepsilon B^{1-\varepsilon})$ | $O(log_B N/\varepsilon)$ | $O(log_B N/\varepsilon + k/B)$ |
| B$^\varepsilon$-tree ($\varepsilon = 1$) | $O(log_B N)$ | $O(log_B N)$ | $O(log_B N + k/B)$ |
| B$^\varepsilon$-tree ($\varepsilon = 0.5$) | $O(log_B N/\sqrt{B})$ | $O(log_B N)$ | $O(log_B N + k/B)$ |

Table 3.1: The asymptotic I/O costs of B-trees and B$^\varepsilon$-trees.

B$^\varepsilon$-trees are asymptotically faster than B-trees, as summarized in Table 3.1. Consider a B-tree with $N$ key/value pairs and in which each node can hold $B$ keys (for simplicity, assume keys have constant size and that the value associated with each key has negligible size). The tree has fanout $B$, so its height is $O(log_B N)$. Inserts and point queries need to fetch all nodes along the root-to-leaf path, resulting in $O(log_B N)$ I/Os. A range query for $k$ key/value pairs requires $O(log_B N + k/B)$ I/Os.

For comparison, a B$^\varepsilon$-tree with node size $B$ has fanout $B^\varepsilon$, where $0 < \varepsilon \leq 1$. Therefore, pivot keys in a non-leaf node consume $B^\varepsilon$ space and the remaining $(B - B^\varepsilon)$ space is used for the message buffer. As a result, the B$^\varepsilon$-tree has height $O(log_B N/\varepsilon)$. A point query fetches all nodes along the root-to-leaf path, inspecting messages in the buffers of non-leaf nodes and key/value pairs in the leaf node. Therefore, the total I/O cost is $O(log_B N/\varepsilon)$. Likewise, a range query for $k$ key/value pairs requires $O(log_B N/\varepsilon + k/B)$ I/Os. On the other hand, the cost of an insert consists of injecting the message into the root node with $O(1)$ I/Os and flushing the message down at each level. In each flush, B$^\varepsilon$-trees has $O(B - B^\varepsilon)$ messages and $B^\varepsilon$ children. Thus, at least one child

can receive no less than $O((B - B^\varepsilon)/B^\varepsilon) = O(B^{1-\varepsilon})$ messages. Therefore, the amortized cost of an insert in one flush is $O(1/B^{1-\varepsilon})$. As the insert must be flushed $O(log_B N/\varepsilon)$ (tree height) times, the amortized cost of the insert is $O(log_B N/\varepsilon B^{1-\varepsilon})$. A B$^\varepsilon$-tree with $\varepsilon = 1$ is equivalent to a B-tree. And if we set $\varepsilon = 1/2$, the point and range query costs of the B$^\varepsilon$-tree become $O(log_B N)$ and $O(log_B N + k/B)$, which is the same as a B-tree, but the insert cost becomes $O(log_B N/\sqrt{B})$, which is faster by a factor of $\sqrt{B}$.

One important change in B$^\varepsilon$-trees from B-trees is the asymmetric I/O costs for point queries and inserts. If an application wants to update the old value associated with a key, a B-tree will perform a point query to get the old value and then issue an insert with the updated value. Because both operations take $O(log_B N)$ I/Os, the total cost remains $O(log_B N)$. However, in B$^\varepsilon$-trees, the query cost is $O(log_B N/\varepsilon)$ I/Os while the insert cost is $O(log_B N/\varepsilon B^{1-\varepsilon})$. A query for the old value degrades the update cost to $O(log_B N/\varepsilon)$ I/Os.

To avoid this read-before-write problem, B$^\varepsilon$-trees introduce "upsert" operations. An upsert operation injects an `UPSERT` message with the key and a delta into the buffer of the root node. When the `UPSERT` message is flushed to the leaf, it updates the old value associated with the key with a user-specified function and the delta in the message. With the introduction of `UPSERT` messages, queries need to compute the new value on the fly if `UPSERT` messages have not reached the leaf node, however, this doesn't change the I/O costs of queries. On the other hand, updating an old value becomes as fast as an insert operation, because it doesn't need to fetch the old value.

Figure 3.3 shows an example of an `UPSERT` message in the B$^\varepsilon$-tree. In the example, the application wants to update the value associated with the key "/B/a". Instead of querying the B$^\varepsilon$-tree for the old value and then inserting the new value, the application calls the upsert operation that injects an `UPSERT` message into the root node, Node $A$. Assume the old value associated with the key "/B/a" in Node $D$ is $v$, the delta in the `UPSERT` message is $\Delta$, and the user-specified function is $f$. The `UPSERT` message essentially updates the value associated with the key "/B/a" to $f(v, \Delta)$. Queries for the key "/B/a" need to calculate $f(v, \Delta)$ on the fly. And the `UPSERT`

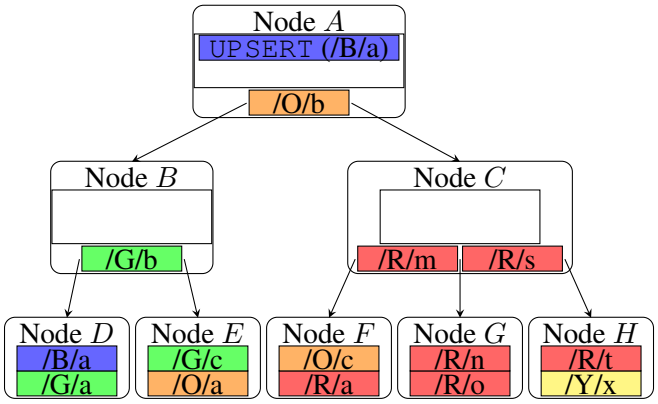



Figure 3.3: An upsert message stores the description of modification on the old value associated with the key.

message will update the value associated with the key "/B/a" to $f(v, \Delta)$ when the B$^\varepsilon$-tree flushes the message to Node $D$.

### 3.2 Full-path-indexed BetrFS

BetrFS [20, 21] is a Linux in-kernel, full-path-indexed, write-optimized file system built upon *ft-index* [43], a key/value store that implements B$^\varepsilon$-trees and exposes a key/value interface similar to Berkeley DB.

Figure 3.4 shows the BetrFS architecture. In Linux, applications interact with file systems through system calls, which invoke the corresponding VFS (Virtual File System) functions. The VFS function then calls the particular function implemented by the underlying file system. BetrFS implements file systems operations by translating them into key/value store operations in *ft-index*. BetrFS interacts with *ft-index* through point operations, such as `put`, `get` and `del`, as well as range queries with cursors (`c_get` with `DB_SET_RANGE` and `DB_NEXT`). BetrFS also uses the transaction interface of *ft-index* to execute multiple operations atomically in one transaction. A redo log and periodic checkpoints (every 60 seconds) in *ft-index* ensure that changes are made persistent on the disk.

*Ft-index* cannot be integrated into a Linux kernel module easily because it is a userspace library that assumes *libc* functions and system calls. To address this issue, we built a shim layer

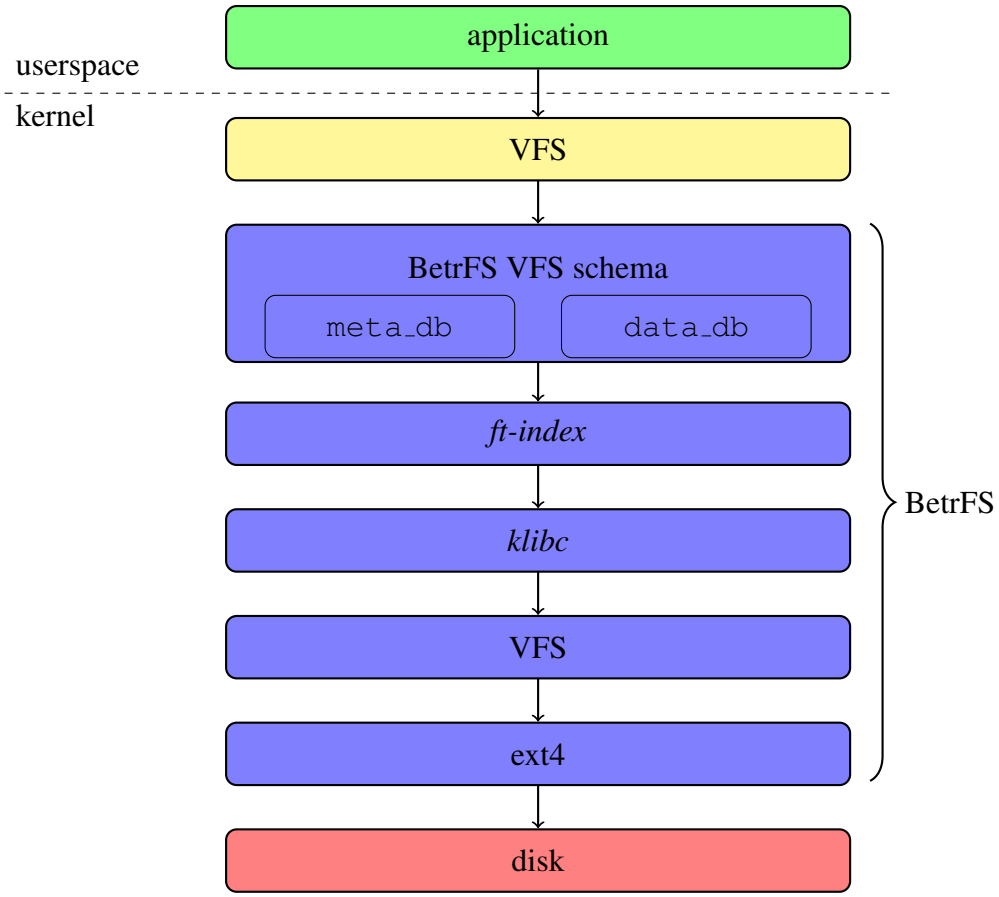



Figure 3.4: The BetrFS architecture.

called *klibc* in BetrFS. *Klibc* implements all functions *ft-index* requires. For example, *ft-index* requires a underlying file system to perform file operations. To this end, *klibc* privately mounts an ext4 file system and calls the corresponding VFS functions when *ft-index* invokes file system operations. Through a shim layer, BetrFS can incorporate *ft-index* into the kernel module without modifying code in *ft-index*.

BetrFS uses two key/value indexes to store metadata and data in the file system. One index, `meta_db`, stores metadata, mapping full-paths to `struct stat` structures. The other index, `data_db`, stores data, mapping (full-path and block number) to 4KiB blocks. When the file system function needs metadata, BetrFS queries the `meta_db` with the full-path key and constructs the corresponding inode from the `struct stat`. Likewise, when a dirty inode needs to be written, the `struct stat` is assembled from the inode and written to the `meta_db` with the

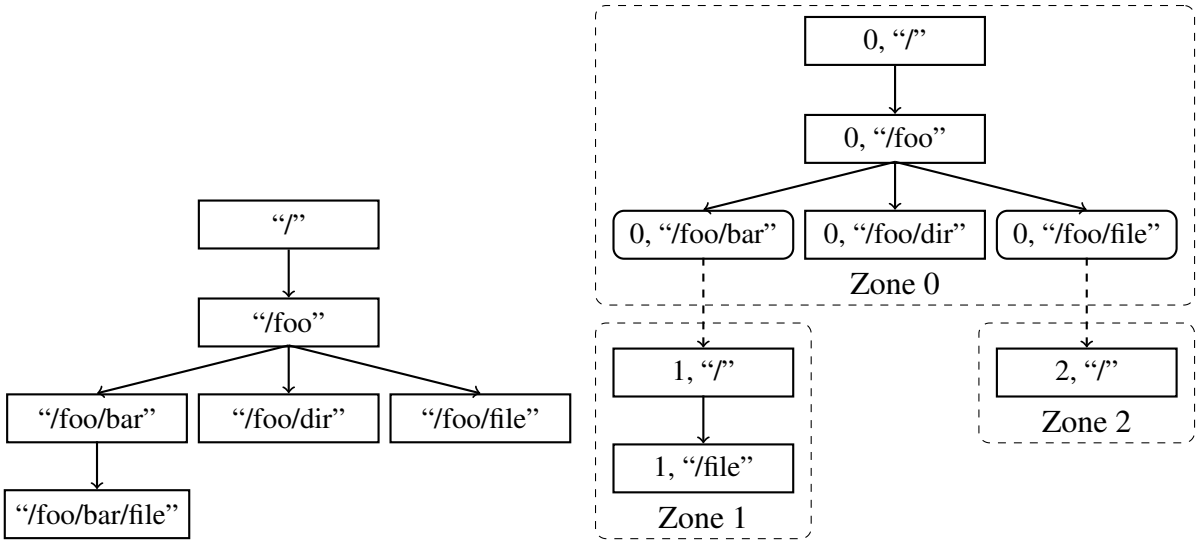
full-path key. Blocks of a file are fetched and written by the full-path and the indexes of blocks. Although other block granularity is possible (we tried 512B for the preliminary implementation of BetrFS), 4KiB is the natural block size because it is the same as the page size in the Linux page cache.

BetrFS can write to a block without fetching the old block into memory, avoiding expensive read-before-write described in Section 3.1. Conventional file systems must read the old block from the disk to the page cache before writing to that block (a complete overwrite can be done without fetching the old block, but it is not implemented in any file system). However, $B^\varepsilon$-trees have asymmetric read and write costs, so read-before-write should be avoided. In BetrFS, if the corresponding block is not in memory, an upsert message, which describes the offset, length and content of this write, is injected into the root node of the $B^\varepsilon$-tree. When this upsert message is flushed to the leaf node, the change is applied to the old block.

Write-optimized $B^\varepsilon$-trees make random writes much faster in BetrFS than conventional file systems. Unlike traditional file systems, which perform each random write at a random location on the disk, BetrFS usually resolves a random write by injecting a message into the root node of the $B^\varepsilon$-tree and flushes messages in batches. Because random I/Os are much slower than sequential I/Os on disks, BetrFS has much better random write performance.

Also, full-path indexing in BetrFS ensures locality over time [10]. After BetrFS fetches one block of a file from the disk, all nodes along the root-to-leaf path are present in memory. And with full-path indexing, all keys under one directory are contiguous in the key space, which means a subsequent fetch of some other block in the same file or another file under than same directory is likely to be resolved in memory, which significantly increases performance and I/O efficiency.

The first implementation of BetrFS (BetrFS 0.1) shows great random write performance. Recursive greps run 3.77x faster than in the best standard file system. File creation runs 12.54x faster. Small, random writes to a file run 68.24x faster [21].




(a) Full-path indexing only keeps full-paths.

(b) Relative-path indexing splits the directory tree into zones and uses indirection between zones.

Figure 3.5: The same directory tree under full-path indexing and relative-path indexing.

However, namespace operations have predictably miserable performance in BetrFS 0.1. Deleting and renaming a Linux source directory take 46.14 and 21.17 seconds, respectively, because the file system has to call one or more key/value store operations for each key.

Later, we fixed the delete problem with range-delete messages [50, 51]. A range-delete message invalidates all key/value pairs within its key range. When flushing a range-delete message, the $B^\varepsilon$-tree injects the message into all children whose key ranges overlap with the range-delete message. Upon reaching a leaf node, the range-delete message deletes all key/value pairs within its key range.

However, the rename problem remains difficult in full-path-indexed BetrFS because a rename needs to update all affected keys in the $B^\varepsilon$-trees, an operation that is slow in key/value stores.

### 3.3 Relative-path-indexed BetrFS

Relative-path-indexed BetrFS [50, 51] backed away from full-path indexing and introduced relative-path indexing, which is also called zoning. Relative-path indexing partitions the directory hierarchy into zones. Each zone has a zone ID (the root zone always has zone ID 0), which

is analogous to an inode number, and a single root file or directory. All files and directories in a zone are indexed relative to the zone root. If the file or directory is the root of another zone, the entry contains the zone ID to redirect queries. With the introduction of zoning, each key in BetrFS contains two parts, a zone ID and the relative path to the zone root.

Figure 3.5 shows an example of the same directory tree under full-path indexing and relative-path indexing. In Figure 3.5b, relative-path indexing partitions the directory into three zones. Zone 0 is the root zone, Zone 1 is rooted at a directory "/foo/bar" and Zone 2 is rooted at a file "/foo/file". When querying the key/value store for file "/foo/file" with key (0, "/foo/file"), the file system gets a special value that indicates the entry is the root of Zone 2. Subsequently, the file system queries the key/value store with key (2, "/") and gets the correct value. Similarly, the file system notices the key for directory "/foo/bar" is (1, "/"). Therefore, when querying for file "/foo/bar/file", it uses key (1, "/file").

Relative-path indexing tries to balance locality and rename performance through a target zone size. On one hand, full-path indexing is still maintained within a zone, so larger zone size gives better locality. On the other hand, smaller zone size imposes a lower bound on rename cost, because no rename needs to mutate more key/value pairs than the zone size. Relative-path-indexing with an infinite zone size is equivalent to full-path-indexing, while relative-path-indexing with a zero zone size is the same as indexing with inode numbers.

Relative-path indexing keeps zones within the target zone size by zone splits and merges. In particular, for each file or directory, BetrFS keeps a counter in its inode to indicate how many key/value pairs will be affected if the file or directory is renamed. When the counter of a directory or file becomes too big, relative-path indexing moves it to its own zone with a zone split, updating all affected keys with the new zone ID and relative-paths. When the counter of a zone root becomes too small (1/4 of the maximum size), relative-path indexing merges it to the parent zone.

The implementation of relative-path-indexed BetrFS, BetrFS 0.2, adopts a 512KiB default zone size. This default zone size is small enough that renames on BetrFS 0.2 are almost as fast as

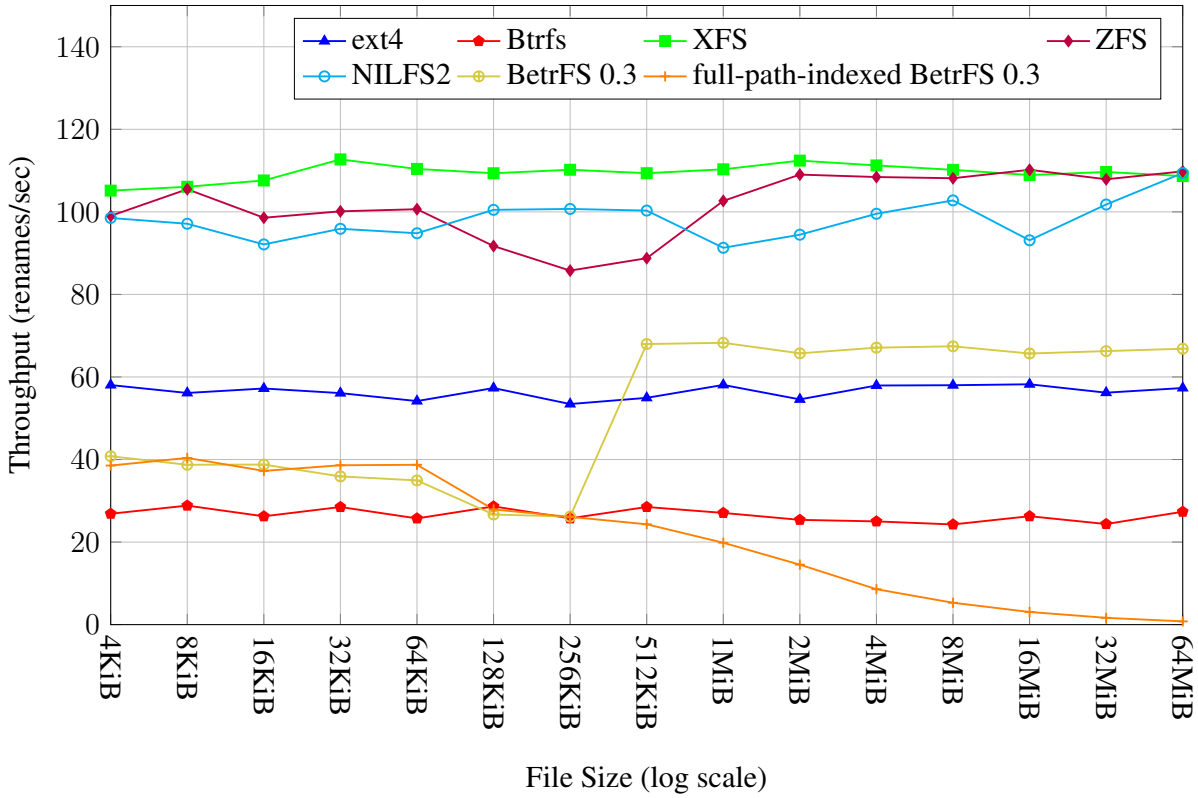



Figure 3.6: Throughput of renaming a file of different sizes (higher is better). The performance of relative-path-indexed BetrFS 0.3 doesn't degrade when the file being renamed becomes larger.

on inode-based file systems. At the same time, the default zone size is large enough that directory traversals are almost as fast as BetrFS 0.1.

Figure 3.6 shows the results of running the file rename benchmark on BetrFS 0.3 (BetrFS 0.3 is BetrFS 0.2 with some bug fixes). The benchmark renames a file of different sizes 100 times, each followed by an `fsync` of the parent directory. We run the benchmark on BetrFS 0.3 twice, one on BetrFS 0.3 (with the default zone size), the other on full-path-indexed BetrFS 0.3, that is, BetrFS 0.3 with an infinite zone size. As shown in the graph, before reaching the target zone size, 512KiB, the performance of relative-path-indexed BetrFS 0.3 is similar to full-path-indexed BetrFS 0.3. However, when the file size reaches the target zone size, the file forms its own zone and file renames become pointer swings. Therefore, after the target zone size, the performance of file renames in relative-path-indexed BetrFS 0.3 doesn't degrade when the file size becomes larger.

| File system | grep (sec) | |
|---|---|---|
| ext4 | 37.795 ± | 1.145 |
| Btrfs | 9.265 ± | 0.139 |
| XFS | 48.130 ± | 0.206 |
| ZFS | 463.184 ± | 33.878 |
| NILFS2 | 8.318 ± | 0.107 |
| BetrFS 0.3 | 5.070 ± | 0.078 |
| full-path-indexed BetrFS 0.3 | 4.022 ± | 0.055 |

Table 3.2: Time to perform recursive grep of the Linux source directory (lower is better). The performance of relative-path-indexed BetrFS 0.3 is still better than conventional file systems.

Table 3.2 shows the performance of directory traversals on different file systems. The benchmark measures the time to `grep` the Linux 3.11.10 source directory. Relative-path-indexed BetrFS 0.3 is slightly slower than full-path-indexed BetrFS 0.3, but still faster than other file systems.

However, relative-path indexing imposes zone maintenance costs on other file system operations. When a file system operation happens to trigger a zone split or merge, in addition to the costs of the operation itself, relative-path indexing charges the zone split or merge to that operation. For instance, Figure 3.7 shows the results of running Tokubench, which creates 3 million small files in a balanced tree structure, on BetrFS 0.3. On the graph, two-thirds of the way through the TokuBench benchmark, BetrFS 0.3 with the default zone size shows a sudden, precipitous drop in cumulative throughput for small file creation, because the file system performs a huge amount of zone splits. At that moment, all benchmarking directories are one file less than the target zone size. Therefore, creating one file in a directory results in a zone split, updating all key/value pairs under the directory. On the contrary, BetrFS 0.3 with an infinite zone size has a smooth curve throughout the benchmark because no zone split happens with a infinite zone size.

Furthermore, relative-path indexing 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), essentially making the file system inode-based. Such a pathological worst case is not possible with

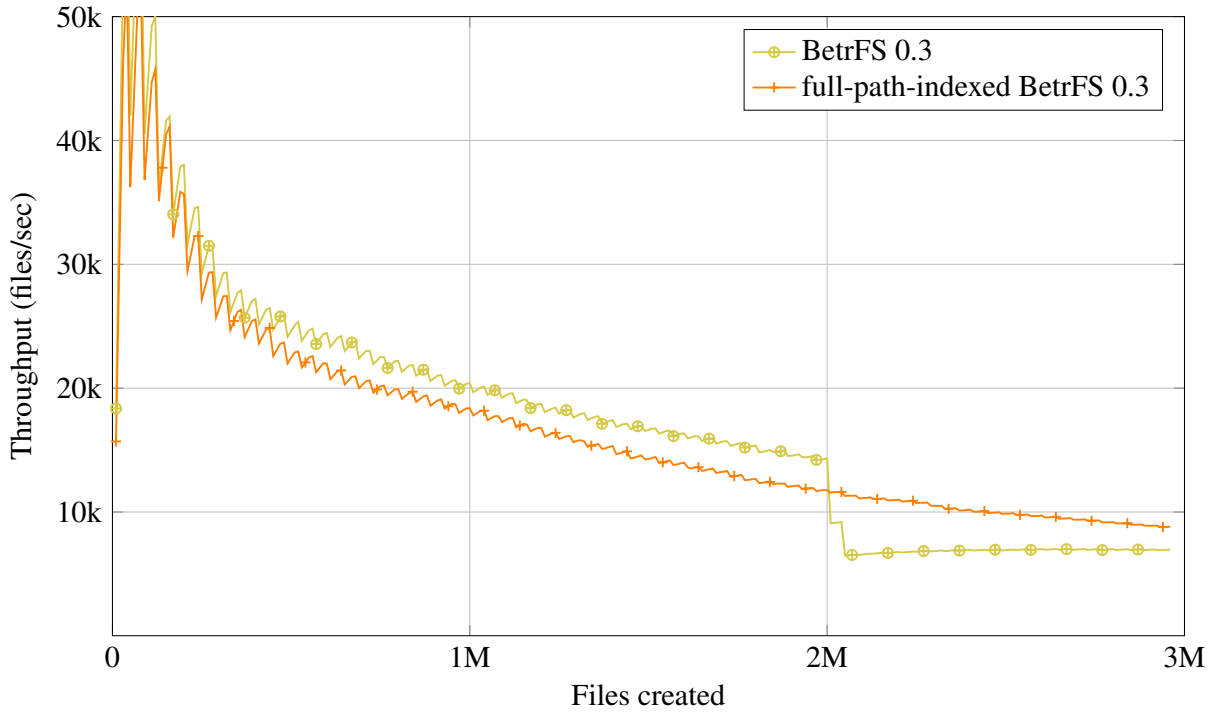



Figure 3.7: Cumulative file creation throughput during the Tokubench benchmark (higher is better). Compared to full-path-indexed BetrFS 0.3, relative-path-indexed BetrFS 0.3 has a sudden performance drop because of zone splits.

full-path indexing in a $B^\varepsilon$-tree, and an important design goal for BetrFS is keeping a reasonable bound on the worst cases.

Finally, relative-path indexing breaks 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 directory. For example, with full-path indexing, we can use range-delete messages not only to delete files, but an entire directory. We could also use range messages to perform a variety of other operations on the whole directory, such as recursive `chmod`, `chown` and timestamp updates. And, as we will eventually see with the range-clone operations, we can clone the whole directory with one operation.

## 3.4 Implementation

This section discusses implementation details of *ft-index*.

### 3.4.1 Synchronization

*Ft-index* keeps in-memory copies of nodes in the cache table to reduce the number of I/Os each operation requires. If an in-memory node is mutated by $B^\varepsilon$-tree operations, *ft-index* marks the node dirty. A dirty node should be written to the disk so that the on-disk version of the node is up to date.

To synchronize operations on the $B^\varepsilon$-tree, *ft-index* maintains a readers-writer lock for each $B^\varepsilon$-tree node. In *ft-index*, a query grabs the read locks of all nodes along the root-to-leaf path, while a write operation grabs the write lock of the root node. A node split or merge grabs the write locks of the parents and children involved in the rebalancing process. Similarly, the flushing process write-locks the parent and the child.

### 3.4.2 The message buffer

*Ft-index* stores messages in a non-leaf node in multiple buffers, called partitions, rather than keeping them in a single large buffer. A non-leaf node in *ft-index* with $n$ children has $n$ partitions, each corresponds to a child of the node, and each partition stores messages in a list, sorted by the logical timestamps. When a message is injected or flushed to a non-leaf node, it is placed at the tail of the list in the partition that corresponds to the child the message will be flushed to later.

Such a design makes implementation simple. *Ft-index* keeps bookkeeping information for each partition and can quickly figure out which child can receive the most messages in the flushing process. Also, because messages are sorted in the timestamp order, the flushing process can simply iterate the list in the parent partition. Moreover, a query doesn't need to fetch the whole non-leaf node, because only one partition can contain messages with the query key. Therefore, the amount of I/Os a query needs to perform is reduced.

A node split also needs to split the parent partition into two new partitions, each for one child generated by the node split. However, splitting a partition with messages is difficult because messages are sorted by the timestamp order, instead of the key order. Therefore, before splitting the child, the node split flushes all messages in the parent partition to the child, emptying the parent

partition. Because a node split holds the write locks of the parent and the child, no message can be injected into the parent partition before the node split completes. And flushing messages in a node split doesn't incur additional I/Os since the node split fetches and dirties the parent and the child anyway. Similarly, a node merge also flushes the parent partitions to the children being merged.

### 3.4.3 Transactions

*Ft-index* performs multiple operations atomically in transactions. To prevent transactions from touching the same key, *ft-index* maintains locks for key ranges in an interval tree. Before querying a key, the operation should read-lock the specific key range so that no other transaction can modify keys in the key range. Likewise, before modifying a key, the operation should write-lock the specific key range. The key locks are released only when the transaction commits or aborts. When a transaction fails to lock a key range for its operation, it should abort.

Transaction commits or aborts are implemented with transaction-commit or transaction-abort messages. When a transaction commits or aborts, it injects one transaction-commit or transaction-abort message for each operation it performs. For example, consider an insert for key $k$ in a transaction. During the transaction, the insert injects an `PUT` message with key $k$ to the $B^\varepsilon$-tree. When the transaction commits, the transaction injects a transaction-commit message with key $k$ for the insert, finalizing the insert. If the transaction aborts, the transaction injects a transaction-abort message with key $k$ for the insert, invalidating the insert. Queries need to ignore uncommitted or aborted messages.

### 3.4.4 Queries

The textbook way of implementing queries on $B^\varepsilon$-trees is to traverse the root-to-leaf path, collecting related messages in non-leaf nodes and related key/value pairs in the leaf node, and figure out the results. However, when we have transactions, upsert messages and range messages, writing query code to distinguish different scenarios is difficult, especially for range queries.

Therefore, *ft-index* implements queries in a different way. Specifically, a query traverses the root-to-leaf path, lock all nodes along the path. Then, the query applies messages in non-leaf nodes to the leaf node in the reverse order, i.e., from the parent of the leaf node to the root node. With pending messages applied to the leaf node, the query can figure out its result by searching the leaf node. Note this doesn't dirty the leaf node, therefore, the asymptotic I/O costs of $B^\varepsilon$-trees are maintained.

### 3.4.5 Recovery

*Ft-index* ensures crash consistency by keeping a redo log of pending messages and applying messages to BetrFS nodes with copy-on-write. At periodic intervals (60 seconds in BetrFS), *ft-index* checkpoints its lifted $B^\varepsilon$-trees by writing all dirty nodes to the disk. Then, *ft-index* can garbage collection the space used by the redo log, which stores operations after the previous checkpoint. Application can ensure all changes are persistent on the disk by force a log flush. When a machine fails, after rebooting, *ft-index* simply replays the redo log on the lifted $B^\varepsilon$-trees at the last checkpoint.

## 3.5 Conclusion

BetrFS is a general file system designed for all operations, with a particular focus on random writes and locality. The underlying data structure, $B^\varepsilon$-trees, performs random writes much faster than B-trees by cascading writes in batches. The full-path indexing schema of BetrFS 0.1 ensures good locality. However, the preliminary implementation of namespace operations in BetrFS 0.1 is slow because they need to iterate all affected keys. The relative-path indexing schema of BetrFS 0.2 enables fast renames by bounding the maximum number of affected in a rename. However, it imposes zone maintenance costs on other file system operations. Also, it breaks the full-path indexing, preventing us from implementing other efficient namespace operations that are difficult on other file systems.

# CHAPTER 4: RANGE-RENAME

This chapter describes the range-rename operation [52, 53] on B$^\varepsilon$-trees. Full-path-indexed BetrFS can implement a file system rename by invoking the range-rename operation on the B$^\varepsilon$-tree that updates all related key in the rename. For example, renaming "/foo" to "/bar" invokes a range-rename operation that updates all keys with prefix "/foo" to have prefix "/bar". The goal of this design is to get good locality from full-path indexing while having good rename performance through an efficient range-rename implementation.

Section 4.1 describes the range-rename interface and shows how BetrFS can implement file system renames by calling range-rename operations. Then, Section 4.2 discusses how to implement the range-rename operation in an I/O-efficient way on B$^\varepsilon$-trees. In particular, the range-rename operation introduces two key techniques, **tree surgery** (Section 4.2.1) and **key lifting** (Section 4.2.2). Finally, Section 4.3 explains implementation details of the range-rename operation in *ft-index* and BetrFS.

## 4.1 The range-rename interface

Range-rename is a new key/value store operation defined as range-rename(*src_prefix*, *dst_prefix*). The range-rename operation takes two prefixes, *src_prefix* and *dst_prefix*, as arguments and updates source and destination key/value pairs (a source/destination key/value pair has the prefix *src_prefix*/*dst_prefix* in its key). Range-rename(*src_prefix*, *dst_prefix*) does three things atomically:

- the range-rename operation deletes all destination key/value pairs from the key/value store
- then, for each source key/value pair $(k, v)$ in the key/value store, the range-rename operation creates a key/value pair $(k', v)$ in the key/value store, where $k$ is the concatenation of *src_prefix* and some suffix $s$ and $k'$ is the concatenation of *dst_prefix* and the same suffix $s$;

- at last, the range-rename operation deletes all source key/value pairs from the key/value store.

In other words, the range-rename deletes all destination key/value pairs and updates all source key/value pairs, changing the key prefix from *src_prefix* to *dst_prefix*.

To see how range-rename accomplishes file system renames in full-path-indexed BetrFS, consider renaming file "/foo" to "/bar". In `meta_db`, BetrFS needs to insert the destination key "/bar" (this insert overwrites the old value of "/bar", if it exists) and delete the source key "/foo", resulting in two operations on the $B^\varepsilon$-tree. On the other hand, in `data_db`, BetrFS needs to delete all data keys of "/bar" (POSIX allows file renames to overwrite the destination file) and update all data keys of "/foo" to be data keys of "/bar". The whole work in `data_db` can be done by calling range-rename("/foo","/bar") (a data key is the concatenation of the full-path and an 8-byte block number).

Similarly, consider renaming directory "/baz" to "/qux". In `meta_db`, BetrFS also needs to insert the destination key "/qux" and delete the source key "/baz" with two operations on the $B^\varepsilon$-tree. Additionally, BetrFS needs to update all keys with prefix "/baz/" to have prefix "/qux/" (POSIX only allows directory renames to overwrite an empty directory, which means there cannot be any key with prefix "/qux/"), which is handled in range-rename("/baz/", "/qux/"). Likewise, in `data_db`, BetrFS needs to update all keys with prefix "/baz/" to have prefix "/qux/", which is done by range-rename("/baz/", "/qux/") (directory doesn't have any data key).

Also, BetrFS puts all operations in a file system rename in a transaction so that all changes are committed atomically.

Table 4.1 summarizes how file system renames can be done with range-rename operations in full-path-index BetrFS. Briefly speaking, full-path-indexed BetrFS can finish a file rename by invoking three operations on $B^\varepsilon$-trees: an insert, a delete and a range-rename operation. And a directory rename can be done with an insert, a delete and two range-rename operations.

In fact, previous full-path-indexed BetrFS implemented the range-rename operation with existing $B^\varepsilon$-tree operations, `del`, `get` and `put`. However, this range-rename implementation

| Type of Rename | Key/Value Store Operations |
|---|---|
| File Rename | transaction_begin();<br>`meta_db` →put(*dst*);<br>`meta_db` →del(*src*);<br>`data_db` →range-rename(*src*, *dst*);<br>transaction_end(); |
| Directory Rename | transaction_begin();<br>`meta_db` →put(*dst*);<br>`meta_db` →del(*src*);<br>`meta_db` →range-rename(*src/*, *dst/*);<br>`data_db` →range-rename(*src/*, *dst/*);<br>transaction_end(); |

Table 4.1: Full-path-indexed BetrFS renames *src* to *dst* by invoking range-rename and other operations on B$^\varepsilon$-trees in a transaction.

updates all source and destination key/value pairs. The I/O-cost of this range-rename implementation increases when the number of source and destination key/value pairs grows. Efficient file system renames in full-path-indexed BetrFS require an efficient range-rename implementation, that is, a range-rename implementation that completes in $O(tree\ height)$ I/Os.

## 4.2 The range-rename operation

This section describes the range-rename implementation on the B$^\varepsilon$-tree that completes in $O(tree\ height)$ I/Os. The efficient range-rename operation requires lexicographic key order so that keys with the same prefix are contiguous in the key space. With contiguous keys, the range-rename operation can create an isolated subtree, which contains all keys of a certain prefix in the B$^\varepsilon$-tree, through **tree surgery**. Then, the range-rename operation moves the isolated subtree to another location in the B$^\varepsilon$-tree. The prefixes of keys in the subtree are updated automatically through **key lifting**, which transforms B$^\varepsilon$-trees into lifted B$^\varepsilon$-trees.

Section 4.2.1 describes tree surgery, which splits nodes in the B$^\varepsilon$-tree to create an isolated subtree of a prefix. Tree surgery also flushes messages in the B$^\varepsilon$-tree to push all related messages into the isolated subtree. Then, Section 4.2.2 introduces key lifting, which transforms the B$^\varepsilon$-trees into lifted B$^\varepsilon$-trees that update prefixes of keys in a subtree automatically when the subtree is

moved to another location. At last, Section 4.2.3 combines the two techniques and shows how to implement the efficient range-rename operation on lifted B$^\varepsilon$-trees.

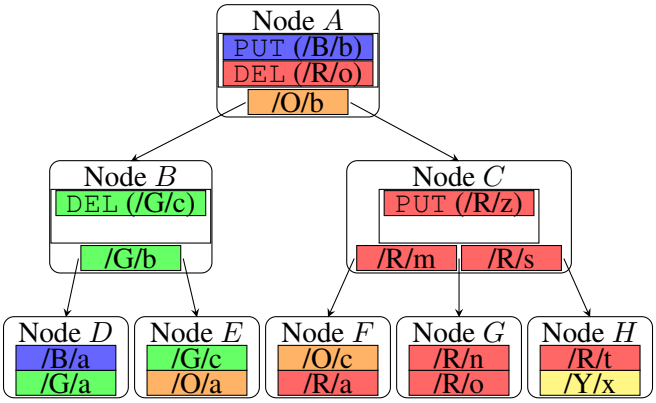
### 4.2.1 Tree surgery

The goal of tree surgery is to slice out an isolated subtree of a certain prefix $p$ in the B$^\varepsilon$-tree. In the B$^\varepsilon$-tree, each node covers a certain key range, bounded by the key range and pivots of its parent. For a certain key range $(p_{min}, p_{max})$ ($p_{min}$ and $p_{max}$ are the minimum and maximum keys with prefix $p$, respectively), there are three types of nodes in the B$^\varepsilon$-tree: nodes whose key ranges are completely out of the key range (exterior nodes), nodes whose key ranges are completely in the key range (interior nodes), and nodes whose key ranges partly overlap with the key range (fringe nodes).

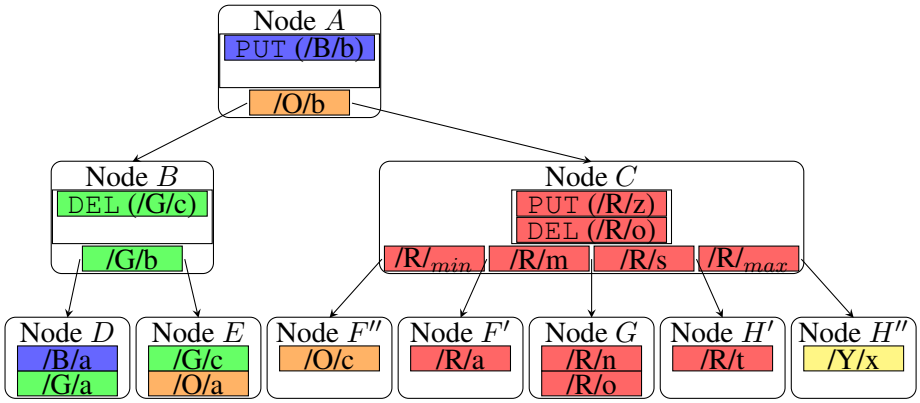
In the B$^\varepsilon$-tree shown in Figure 4.1a, consider prefix "/R/" with key range ("/R/$_{min}$", "/R/$_{min}$"), Node $B$, $D$ and $E$ are exterior nodes, Node $G$ is an interior node, and the other nodes are fringe nodes.

**Identifying fringe nodes.** The first step of tree surgery is to identify all fringe nodes. Because the key range of a fringe node partly overlaps with key range $(p_{min}, p_{max})$, a fringe node must include either $p_{min}$ or $p_{max}$ in its key range. Therefore, tree surgery can perform two root-to-leaf traversals with two keys, $p_{min}$ and $p_{max}$, to identify all fringe nodes. For example, in Figure 4.1a, we walk down the B$^\varepsilon$-tree with "/R/$_{min}$" and "/R/$_{max}$" to identify all fringe nodes of prefix "/R/", Node $A$, $C$, $F$ and $H$.

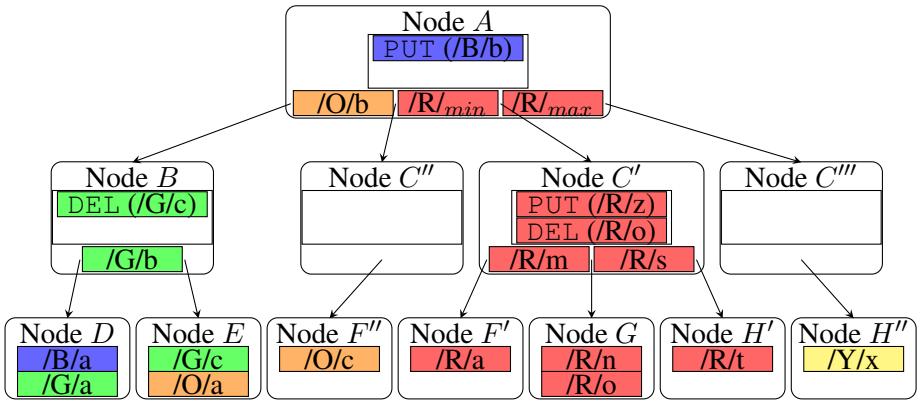
An important fringe node for tree surgery is the **LCA** (Lowest Common Ancestor) of the two traversing keys, that is, the lowest (the most distant from the root node) B$^\varepsilon$-tree node whose key range includes both keys. For example, in Figure 4.1a, Node $C$ is the LCA of prefix "/R/". The subtree rooted at the LCA is the lowest subtree in the B$^\varepsilon$-tree that covers the whole range $(p_{min}, p_{max})$. The goal of tree surgery is to generate an isolated subtree rooted at the LCA that contains all keys with prefix $p$. Therefore, before reaching the LCA, the traversals also flush messages from the parent to the child so that there is no pending message above the LCA.




(a) The B$^\varepsilon$-tree before tree surgery.




(b) Tree surgery splits leaf nodes.




(c) Tree surgery splits the LCA.

Figure 4.1: Tree surgery slices out an isolated subtree of prefix "/R/".

**Slicing.** The goal of slicing is to separate unrelated keys in the fringe nodes from keys in the key range (thus with prefix $p$). With all related messages and key/value pairs in the subtree rooted at the LCA, tree surgery starts slicing out the isolated subtree by splitting fringe nodes from the bottom up. Slicing uses the same code as standard B$^\varepsilon$-tree node splits, but, rather than picking a key in the middle of the node, divides the node at one of the slicing keys, $p_{min}$ or $p_{max}$.

Figure 4.1b shows the B$^\varepsilon$-tree after the bottom-up slicing splits leaf nodes. Tree surgery splits Node $F$ with key "/R/$_{min}$", generating an interior node, Node $F'$, and an exterior node, Node $F''$. Likewise, tree surgery splits Node $H$ into an interior node, Node $H'$, and an exterior node, Node $H'$, with key "/R/$_{max}$". Note, the message `DEL` ("/R/o") has been flushed from Node $A$ to Node $C$ before slicing.

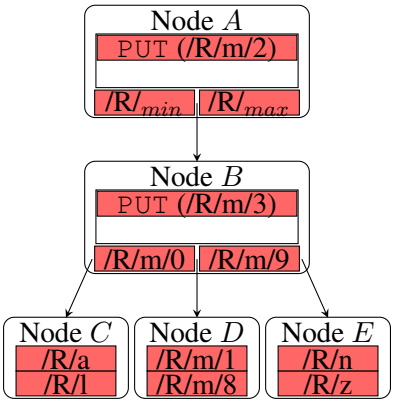
At last, in Figure 4.1c, tree surgery splits the LCA, Node $C$, into Node $C'$, $C''$ and $C'''$ with both keys, "/R/$_{min}$" and "/R/$_{max}$". The subtree rooted at Node $C'$ is the isolated subtree that contains and only contains all keys with the prefix "/R/".

A special case in tree surgery is when the LCA is the root node of the B$^\varepsilon$-tree. In such a scenario, splitting the root node divides the tree into a forest. To avoid such case, we create a new root node as the parent of the old root node before slicing.

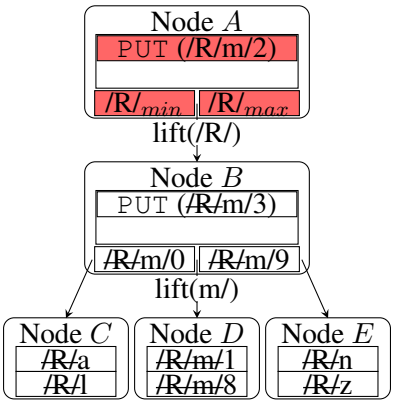
### 4.2.2 Key lifting

After tree surgery, one can move the isolated subtree to another location in the B$^\varepsilon$-tree. However, the keys in the subtree will not be coherent with the new location in the tree. Thus, as a part of the range-rename operation, the prefixes of keys in this subtree need to be updated. A naive approach will traverse the subtree and update all keys. However, this process costs a lot of I/Os, rewriting every node in the source subtree. The particularly concerning case is when the subtree is very large.

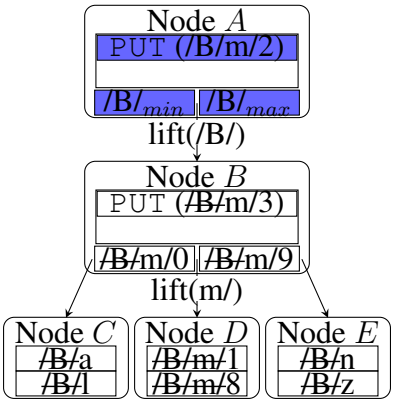
Key lifting eliminates the need to update prefixes of keys in the subtree by transforming B$^\varepsilon$-trees into lifted B$^\varepsilon$-trees. The idea of key lifting comes from the observation that with lexicographic key order, all keys bounded in the key range of two pivots must have the same prefix that




(a) A B$^{\varepsilon}$-tree without key lifting.




(b) The lifted B$^{\varepsilon}$-tree with the same keys. Lifted prefixes are marked as strike-through in keys.




(c) When the subtree rooted at Node $B$ is bounded by two different pivots, all keys in the subtree change their prefixes.

Figure 4.2: Key lifting lifts prefixes from the subtree. Keys in the same subtree are viewed with different prefixes when the pivots in its parent change.

is the **LCP** (Longest Common Prefix) of the pivots. Therefore, this LCP is redundant information in the subtree of the two pivots and can be removed from the subtree. In particular, the subtree generated by tree surgery is bounded by two pivots, $p_{min}$ and $p_{max}$ and with lexicographic key order, all keys in the subtree must have prefix $p$.

In lifted B$^\varepsilon$-trees, a parent-to-child pointer lifts the LCP of the two pivots from the subtree rooted at the child. Child nodes only store differing key suffixes. This approach encodes the complete key in the path taken to reach a given node and one can then modify the prefix for a lifted subtree by only modifying the parent node, eliminating the need to change key and pivot prefixes in all nodes of a subtree.

Figure 4.2 shows an example of key lifting. Figure 4.2a and Figure 4.2b show the same B$^\varepsilon$-tree with and without key lifting, respectively. In the interest of clarity, irrelevant nodes are omitted from the figure. In Figure 4.2b, the subtree rooted at Node $B$ is bounded by two pivots, "/R/$_{min}$" and "/R/$_{max}$", in Node $A$. Therefore, all keys in the subtree must have prefix "/R/" and key lifting removes this prefix from the subtree (we show the lifted prefix on the parent-to-child pointer and mark the lifted prefix as strike-through in the subtree). Because keys in Node $B$ don't have prefix "/R/" physically in the node, we show transparent keys ("/R/" keys are red in previous examples) in the node. Likewise, Node $D$ is bounded by "m/a" and "m/z" in Node $B$ (note "/R/" is already lifted from the sutbree rooted at Node $B$), so key lifting removes prefix "m/" from Node $D$. In Figure 4.2c, the same subtree, which contains exactly the same key/value pairs, is moved to a different location, bounded by two new pivots, "/B/$_{min}$" and "/B/$_{max}$", in Node $A'$. Because the prefix lifted through the parent-to-child pointer in Node $A'$ becomes "/B/", all keys in the subtree have prefix "/B/" instead. For examples in the rest of the dissertation, we will not show lifted prefixes in keys.

A query on a lifted B$^\varepsilon$-tree needs to track lifted prefixes during the root-to-leaf traversal and reconstruct the full key by concatenating these prefixes and the suffix in the leaf. For example, consider a query for key "/R/m/1" in Figure 4.2b. When following the parent-to-child pointer from Node $A$ to Node $B$, the query notices the lifted prefix "/R/". Therefore, the query searches

for key "m/1" in Node $B$ and follows the parent-to-child pointer to Node $D$. Again, the query removes the lifted prefix, "m/", from the search key. After fetching the key/value pair of key "1" in Node $D$, the query prepends prefixes lifted long the root-to-leaf path and recovers the full key "/R/m/1". Note the recovering process is not necessary for point queries because they know their keys beforehand, but range queries must reconstruct the resulting keys.

Also, a lifted B$^\varepsilon$-tree must remove the lifted prefix from a message before flushing the message from parent to child. For example, in Figure 4.2b, when flushing `PUT` ("/R/m/2",...) from Node $A$ to Node $B$, the lifted B$^\varepsilon$-tree remove the prefix "/R/" from the message and injects a message `PUT` ("m/2",...) into the buffer of Node $B$.

A node split in a lifted B$^\varepsilon$-tree adds a pivot to the parent, which may change the lifted prefix associated with the parent-to-child pointer, so it may need to update keys in the resulting children. Similarly, a node merge needs to re-lift keys in the resulting node.

However, the extra work described above for B$^\varepsilon$-tree operations can be resolved in memory. Therefore, key lifting doesn't incur additional I/Os for other B$^\varepsilon$-tree operations.

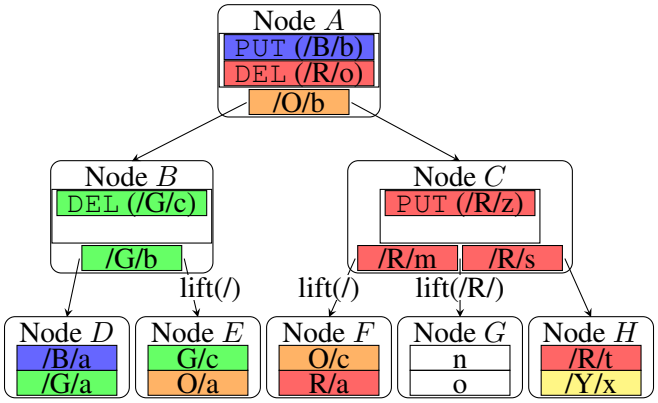
Compared to zoning, in which the file system removes certain prefixes from keys, key lifting is completely transparent to BetrFS. BetrFS still stores key/value pairs with full-path keys, just with a slightly different data structure.

### 4.2.3 The range-rename operation on lifted B$^\varepsilon$-trees

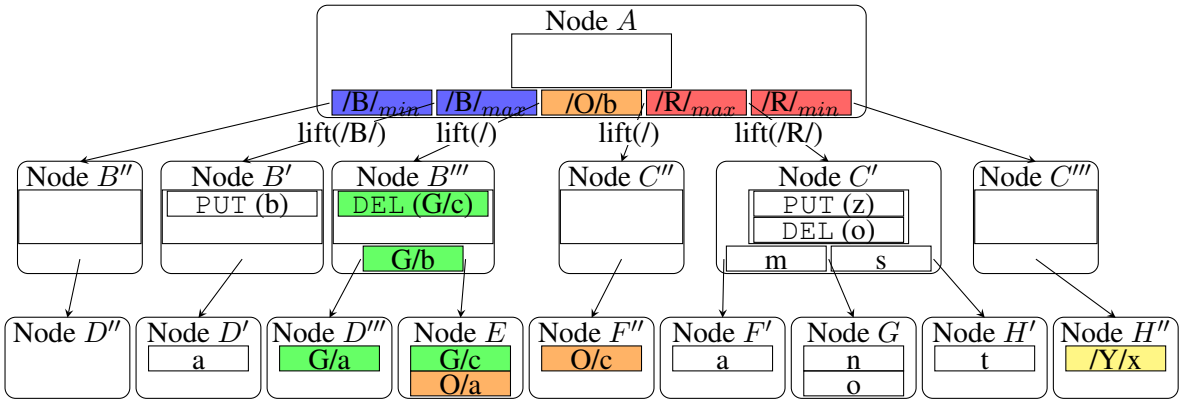
On a lifted B$^\varepsilon$-tree, range-rename(*src_prefix*, *dst_prefix*) can be done with two steps:

**Tree Surgery.** The range-rename operation first slices out two isolated subtrees in the lifted B$^\varepsilon$-tree simultaneously, one of *src_prefix* and the other of *dst_prefix*.

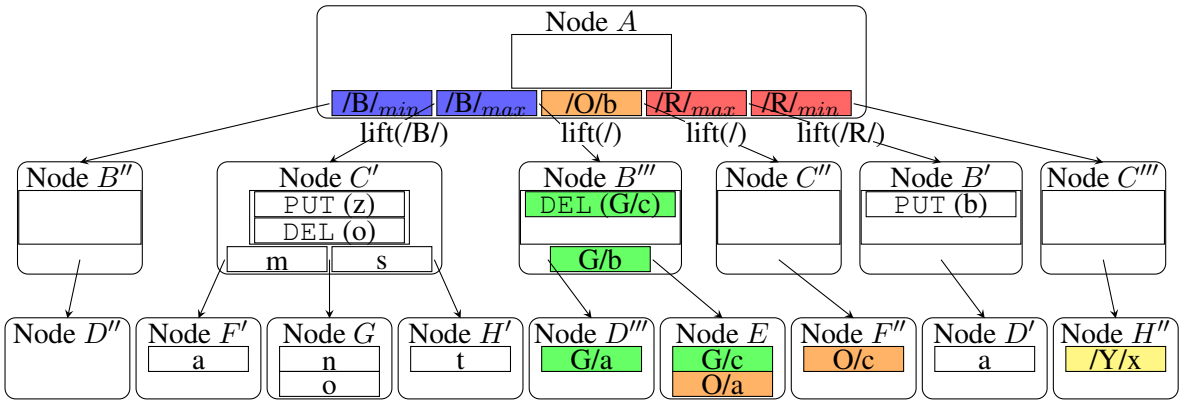
**Transplant.** Then, the range-rename operation swaps these two subtrees and injects a range-delete message for source keys. Alternatively, one can garbage collect the destination subtree and merge nodes at the source. However, garbage collecting the destination subtree requires




(a) The B$^\varepsilon$-tree before the range-rename operation.




(b) The range-rename operations performs tree surgery to slice out the source and destination subtrees.




(c) The range-rename operation swaps the subtrees and injects a range-delete message for source keys.

Figure 4.3: An example of range-rename("/R/", "/B/") on the B$^\varepsilon$-tree.

traversing the subtree. Instead, we leverage the existing range-delete message to reclaim the space of the destination subtree.

Similar to a B-tree, a B$^\varepsilon$-tree should keep all leaf nodes at the same distance from the root node. Otherwise, the B$^\varepsilon$-tree becomes unbalanced, breaking the asymptotic I/O cost analysis. In order to keep all leaf nodes at the same distance after transplanting, the range-rename operation requires the source and destination subtrees to be at the same height. Therefore, in tree surgery, for the subtree with a lower LCA, the range-rename operation keeps slicing its ancestors by adding empty nodes with only one child.

Figure 4.3 shows an example of range-rename("/R/", "/B/") on the B$^\varepsilon$-tree. Figure 4.3a shows the lifted B$^\varepsilon$-tree before the range-rename operation. In Figure 4.3b, the range-rename operation performs tree surgery to slice out the source subtree, rooted at Node $C'$, and the destination subtree, rooted at Node $B'$. Though the destination LCA in Figure 4.3a is Node $D$, tree surgery still splits Node $B$ to keep the source and destination subtrees at the same height. Also, prefix "/R/" is lifted from the source subtree and prefix "/B/" is lifted from the destination subtree. Figure 4.3c shows the lifted B$^\varepsilon$-tree after swapping the source and destination subtrees, with a range-delete message for "/R/" keys in Node $A$. At this point, no key with prefix "/R/" exists in the lifted B$^\varepsilon$-tree because of the range-delete message, and all keys with prefix "/R/" in Figure 4.3a now have prefix "/B/".

**Healing.** Tree surgery may create undersized B$^\varepsilon$-tree nodes, i.e., non-leaf nodes without enough children, or leaf nodes without enough key/value pairs. In a normal node split, the B$^\varepsilon$-tree evenly splits a node that is oversized (a non-leaf node with too many children or a leaf node with too many key/value pairs), so the resulting two nodes will not be undersized. However, tree surgery splits normal nodes with the minimum and maximum keys of a certain prefix, which can split the node unevenly. Moreover, to meet the tree height invariant, tree surgery may create "stalks", non-leaf nodes with only one child, so that the source and destination subtrees are at the same height.

The range-rename operation handles this situation by triggering a rebalancing within the tree. Specifically, if a node has only one child, the slicing process will merge the node after completing the work of the range-rename operation. After the transplant completes, there may be a number of B$^\varepsilon$-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$^\varepsilon$-tree node merge.

**Complexity.** During tree surgery, at most 4 root-to-leaf paths (2 for source and 2 for destination) are traversed, dirtying all nodes along the paths. 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 required in tree surgery is at most proportional to the height of the B$^\varepsilon$-tree, which is logarithmic in the size of the tree.

The healing process only merges nodes that are split during the tree surgery process. Therefore, the number of I/Os required in the healing process is also $O(log_B N/\varepsilon)$ (tree height).

There is also lifting work along all nodes that are sliced in tree surgery or merged in healing. 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 during a range-rename is no more than the nodes that must be sliced during tree surgery, and proportional to the height of the tree ($O(log_B N/\varepsilon)$).

In summary, the I/O cost of a range-rename operation is $O(log_B N/\varepsilon)$. Therefore, the cost of a range-rename operation is bounded by the B$^\varepsilon$-tree height, which is determined by the total size of key/value pairs in the B$^\varepsilon$-tree. No matter how many key/value pairs are involved in the range-rename operation, the cost of a range-rename operation doesn't change logarithmically.

### 4.3 Implementation

This section discusses implementation details of the range-rename operation.

### 4.3.1 Synchronization

The range-rename operation synchronizes with other B$^\varepsilon$-tree operations using the readers-writer locks of B$^\varepsilon$-tree nodes (described in Section 3.4.1). Starting from the root node, the range-rename operation write-locks B$^\varepsilon$-tree nodes hand-over-hand until reaching the LCAs. To avoid newer messages being flushed to the subtrees rooted at the LCAs, the write locks of the LCAs and the parents of the LCAs are held until the range-rename operation completes. Then, the range-rename operation write-locks all fringe nodes below the LCAs and releases the write locks after the bottom-up node splits. After transplanting, the range-rename operation finally releases the write locks of the LCAs and the parents of the LCAs.

### 4.3.2 Re-lifting a non-leaf node

Re-lifting a node requires updating prefixes of all keys in the node. Because *ft-index* has a large node size (4 MiB), re-lifting a node can cause a large amount of computation.

However, as described in Section 3.4.2, *ft-index* stores messages in a non-leaf node in partitions, each corresponding to a child. Therefore, we can lift the prefixes of keys in a partition by the LCP of two pivots bounding it in the non-leaf node. In other words, keys in the partition are not only lifted by pivots in ancestors but also the two pivots in the node. For example, consider a non-leaf node that contains keys in ("/R/a/1", "R/b"), key lifting lifts prefix "/R/" from all keys in the node. Assume there are two adjacent pivots "a/2" and "a/3" in the node (prefix "/R/" is lifted). There is a partition in the node storing all keys in ("/R/a/2", "/R/a/3"), with prefix "/R/a/" lifted.

To see how it helps, consider a partition bounded by two pivots in the non-leaf node. All keys in the partition are lifted by the total lifted prefix in ancestors, $p$, and the LCP of the two pivots, $q$. Therefore, the total amount of prefix lifted from keys in the partition is $(p + q)$. Now, assume re-lifting further lifts some prefix $s$ from the node, making the total lifted prefix by ancestors $(p+s)$ and the LCP of the two pivots $(q - s)$. The total amount of prefix lifted for keys in the partition remains $(p + q)$. Similarly, assume re-lifting reduces the lifted prefix from ancestors to $(p - s')$. Now the LCP of the two pivots become $(q + s')$, and the total amount of prefix lifted for keys in

the partition is still $(p + q)$. In the previous example, assume we split the node with a new pivot "/R/a/9". The resulting node that contains keys in ("/R/a/1", "/R/a/9") has "/R/a/" lifted. The two pivots bounding the partition become "2" and "3". However, for all keys in the partition, the total lifted prefix is still "/R/a/".

Therefore, re-lifting a non-leaf node only needs to update pivots. It is not necessary to touch any messages in partitions. However, re-lifting a leaf node still requires updating all keys.

### 4.3.3 Splitting fringe nodes

As discussed in Section 3.4.2, because partitions store messages in the timestamp order, a node split needs to empty the parent partition through flushing before splitting the child. Since tree surgery splits fringe nodes from bottom up, all partitions involved must be empty. To this end, in the process that identifies all fringe nodes, after flushing all related messages to the LCA, tree surgery keeps flushing messages along the two root-to-leaf paths that contain fringe nodes. Because tree surgery holds the write lock of the parent of the LCA, no message will be injected into the partitions involved in tree surgery.

### 4.3.4 Transactions

The range-rename operation fits into the transaction framework described in Section 3.4.3. When invoked, the range-rename operation first write-locks the source and destination key range. However, the range-rename operation doesn't generate messages that can be committed or aborted with transaction-commit or transaction-abort message. Therefore, we perform the whole work of the range-rename operation when the transaction commits. If the transaction aborts, nothing happens to the lifted $B^\varepsilon$-tree.

### 4.3.5 Recovery

After a crash, the range-rename operation can be recovered if the log entry is written to the redo log of *ft-index* (Section 3.4.5). A range-rename operation is *logically applied* as soon as

the range-rename and its corresponding transaction commit entries are inserted into the redo log. The range-rename operation is durable as soon as the redo log entry is written to disk. If the system crashes after a range-rename operation is logged, the recovery will see a prefix of the message history that includes the range-rename and its transaction commit entries, and performs the corresponding range-rename operation on the lifted $B^\varepsilon$-trees of the last checkpoint.

### 4.3.6 Latency

The range-rename operation returns to the user once the range-rename and its transaction commit entry is in the log and the root node of the $B^\varepsilon$-tree is write-locked. No read or write operation to the $B^\varepsilon$-tree can start before the range-rename operation releases the root lock. The rest of the range-rename work is handed off to background threads that perform slicing, transplanting and healing.

### 4.3.7 BetrFS key order

There are 2 constraints on the key order of full-path-index BetrFS with the range-rename operation:

- the readdir constraint. Because BetrFS uses range-queries to fetch all child entries for readdirs, all files and directories immediately under one directory must be contiguous in key space.
- the lexicographic constraint. Key lifting only works properly with lexicographic key order. In order to use the range-rename operation, BetrFS must have lexicographic key order.

Simple `memcmp` key order fails the readdir constraint. Consider entries "/bar/dir", "/bar/dir/-file" and "/bar/file" in the that order, a readdir for directory "/bar" needs to skip "/bar/dir/file" in its range queries. In the worst case, a readdir might need to skip almost all keys in the key/value store.

The old BetrFS key order sorts full-path keys first by the number of slashes and then by a `memcmp`. This key order satisfies the readdir constraint but fails the lexicographic constraint, so BetrFS needs a new key order for range-rename.

In order to use the range-rename operation, BetrFS tweaks its full-path keys by adding one additional slash alongside the last slash. Now, "/foo" and "/foo/bar" become "//foo" and "/foo//bar", respectively (for correct ordering, '\x01' is used as slashes). With the new full-path keys, BetrFS can use `memcmp` as the key comparison function while satisfying the readdir constraint.

## 4.4 Conclusion

This chapter presents the new range-rename operation on $B^\varepsilon$-trees. File system renames in full-path-indexed BetrFS can be done with an insert, a delete and one or two range-rename operations. And on lifted $B^\varepsilon$-trees, a range-rename operation can be done efficiently with tree surgery.

BetrFS with range-rename demonstrates the possibility of consolidating efficient renames into full-path-indexed file systems, showing the possibility of building file systems that are good at locality and namespace operations. Moreover, full-path indexing ensures all metadata or data in a directory are contiguous in the key space, creating more opportunity for namespace operations.

Key lifting and tree surgery can be applied to other tree-style data structures. For example, one can build a full-path-indexed file system on B-trees with the same range-rename design. However, the generality means the design doesn't fully utilize the write-optimization of $B^\varepsilon$-trees.

Also, the technique is not limited to the range-rename operation that updates keys with certain prefix. For a key/value store operation that updates a lot of keys, one can use a similar technique as long as the key/value store can group all related keys in a contiguous key range and the bounding keys of the key range can "lift" the parts of related keys being updated.

# CHAPTER 5: RANGE-CLONE

This chapter describes how to implement another type of namespace operations, the file or directory clone, in a full-path-indexed file system. File or directory clones are useful in many cases. For example, many people build versioning file systems [28, 38, 41] which constantly make read-only clones of files and directories. Therefore, a user can get an old version of a file or directory after committing unwanted changes to the file or directory. Also, container applications usually clone the image file before booting the container.

Similar to file or directory renames, we implement file or directory clones with the range-clone operation on $B^\varepsilon$-trees. Unlike a range-rename operation, which completes all its work at once, a range-clone operation injects a new type of message, the `GOTO` message, into the root node of the $B^\varepsilon$-tree. The $B^\varepsilon$-tree then flushes `GOTO` messages with other messages in batches, gradually finishing the range-clone work. Therefore, the range-clone operation fits into the write-optimized framework of $B^\varepsilon$-trees, and the I/O cost of a range-clone operation is amortized with other operations.

Section 5.1 describes the range-clone interface and how to implement file or directory renames and clones in full-path-indexed BetrFS by calling the range-clone operation. Section 5.2 discusses how to implement the range-clone operation. In particular, we first describe an implementation of the range-clone operation with techniques introduced by the range-rename operation, finishing all work on the critical path. Then, we introduce the `GOTO` message that delays most of the range-clone work to the flushing process in the data structure. Finally, Section 5.3 explains some implementation details in the range-clone operation.

## 5.1 The range-clone interface

Range-clone is defined as range-clone(*src_prefix*, *dst_prefix*). Range-clone(*src_prefix*, *dst_prefix*) does the following things atomically:

- the range-clone operation deletes all destination key/value pairs from the key/value store;
- then, for each source key/value pair $(k, v)$ in the key/value store, the range-clone operation creates a key/value pair $(k', v)$ to the key/value store, where $k$ is the concatenation of *src_prefix* and some suffix $s$ and $k'$ is the concatenation of *dst_prefix* and the same suffix $s$;

Therefore, range-clone(*src_prefix*, *dst_prefix*) is equivalent to range-rename(*src_prefix*, *dst_prefix*) without deleting source key/value pairs.

Table 5.1 summarizes how BetrFS implements file or directory renames and clones by invoking the range-clone operation. Because a range-clone operation is the same as a range-rename operation without deleting the source key/value pairs, BetrFS can complete a range-rename operation with a range-clone operation and a range-delete operation (described in Section 3.2) that deletes all source key/value pairs. Therefore, BetrFS implements file or directory renames by replacing the range-rename operation in Table 4.1 with a range-clone operation and a range-delete operation. And if BetrFS calls the range-clone operation without the range-delete operation, the source key/value pairs of metadata and data stay in the key/value store. In such a scenario, the file system completes file or directory clones if it doesn't delete the metadata for the source file or directory (in rename, this delete is not covered in the range-rename operation).

Also, BetrFS puts all operations in a file system rename in a transaction so that all changes are committed atomically.

## 5.2 The range-clone operation

This section shows the implementation of the range-clone operation.

Section 5.2.1 shows all the changes needed if we perform all range-clone work on the critical path. In particular, we show that the range-clone operation can be implemented by modifying the

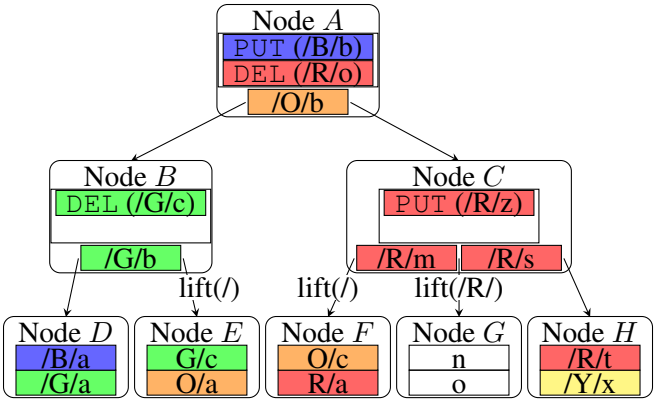
| Type of File System Operation | Key/Value Store Operations |
|---|---|
| File Rename | transaction_begin();<br>meta_db →put(*dst*);<br>meta_db →del(*src*);<br>data_db →range-clone(*src*, *dst*);<br>data_db →range-delete(*src*)<br>transaction_end(); |
| Directory Rename | transaction_begin();<br>meta_db →put(*dst*);<br>meta_db →del(*src*);<br>meta_db →range-clone(*src/*, *dst/*);<br>meta_db →range-delete(*src/*);<br>data_db →range-clone(*src/*, *dst/*);<br>data_db →range-delete(*src/*);<br>transaction_end(); |
| File Clone | transaction_begin();<br>meta_db →put(*dst*);<br>data_db →range-clone(*src*, *dst*);<br>transaction_end(); |
| Directory Clone | transaction_begin();<br>meta_db →put(*dst*);<br>meta_db →range-clone(*src/*, *dst/*);<br>data_db →range-clone(*src/*, *dst/*);<br>transaction_end(); |

Table 5.1: Full-path-indexed BetrFS renames or clones *src* to *dst* by invoking range-clone and other operations on $B^\varepsilon$-trees.

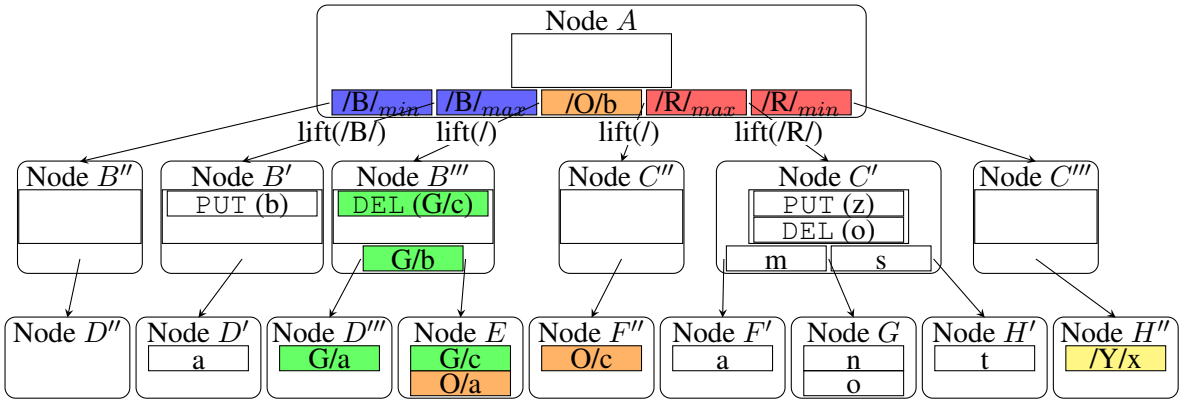
range-rename operation. This implementation shows all the work in the range-clone operation and helps the understanding of the write-optimized implementation described in later sections. Then, Section 5.2.2 introduces `GOTO` messages. The range-rename operation can return to the application immediately after injecting a `GOTO` message into the root node, without slicing out the subtrees. Finally, Section 5.2.3 shows how `GOTO` messages are flushed in the data structure, gradually finishing the range-clone work.

### 5.2.1 Range-clone, on the critical path

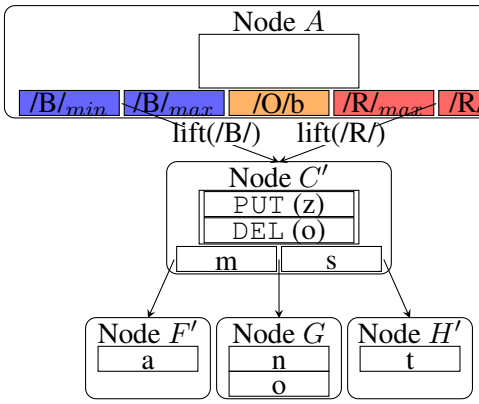
The range-clone operation can be implemented in the $B^\varepsilon$-tree by modifying the range-rename operation. In particular, the range-rename operation swaps the two isolated subtree after slicing.




(a) The B$^{\varepsilon}$-tree before the range-clone operation.




(b) The range-clone operation slices out the source and destination subtrees.




(c) The range-clone operations shares the source subtree at both source and destination and garbage-collects the destination subtree.

Figure 5.1: An example of range-clone("/R/", "/B/") that completes all work in the critical path.

Instead, a range-clone implementation can share the source subtree in both source and destination, and garbage collect the destination subtree.

Figure 5.1 shows an example of performing the work of range-clone("/R/", "/B/") on the lifted B$^\varepsilon$-tree in Figure 5.1a. Figure 5.1b shows the lifted B$^\varepsilon$-tree after tree surgery. Tree surgery slices out the source and destination subtrees, after flushing pending messages to the LCAs. This tree surgery is completely identical to that in the range-rename operation (Figure 4.3b). In Figure 5.1c, the lifted B$^\varepsilon$-tree garbage collects the destination subtree, rooted at Node $B'$, and sets the parent-to-child pointer between "/B/$_{min}$" and "/B/$_{max}$" ("/B/$_{min}$" and "/B/$_{max}$" are the minimum and maximum keys with prefix "/B/", respectively) to Node $C'$, which is the root node of the source subtree. In the interest of clarity, irrelevant nodes are omitted from the figure.

The subtree rooted at Node $C'$ is now shared by two parent-to-child pointers, and queries for both "/B/" and "/R/" keys will fetch messages and key/value pairs in that subtree. However, because queries need to reconstruct the key by prepending lifted prefixes on the root-to-leaf path, they get different keys following different parent-to-child pointers.

Sharing a subtree among multiple parent-to-child pointers transforms a lifted B$^\varepsilon$-tree into a lifted B$^\varepsilon$-DAG (Directed Acyclic Graph) with some constraints. Because the B$^\varepsilon$-DAG is generated by sharing subtrees in a B$^\varepsilon$-tree, there is still one root node in the B$^\varepsilon$-DAG, which can reach all nodes in the B$^\varepsilon$-DAG through parent-to-child pointers. And since the source and destination subtrees generated by tree surgery are at the same height, the length of any root-to-leaf path in the B$^\varepsilon$-DAG is still logarithmic in the size of the graph.

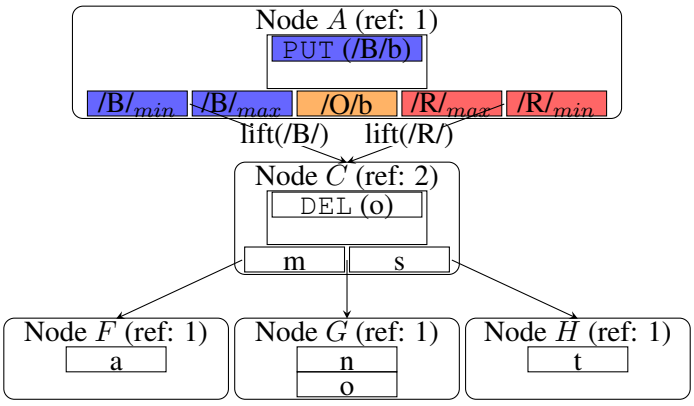
The lifted B$^\varepsilon$-DAG adds reference counts for B$^\varepsilon$-tree nodes to track the sharing status. In particular, *ft-index* has a node table that a maps node ID to the actual location of the node. We store the reference count of each node in the node table alongside the mapping. Before flushing to a node, the B$^\varepsilon$-DAG must check the reference count of the node. A reference count greater than 1 means the node is shared by multiple parent-to-child pointers. The lifted B$^\varepsilon$-DAG must break the sharing because other paths should not see the messages this flush is going to inject into the node.

In a lifted B$^\varepsilon$-DAG, the root-to-leaf traversals of different queries may end up reaching the same leaf node, fetching the same key/value pair. However, because key lifting requires queries to reconstruct the full key by concatenating lifted prefixes, different queries treat the same key/-value pairs with different prefixes. For example, in Figure 5.1c, queries for key "/R/n" and "/B/n" follow the same root-to-leaf path and get the same key/value pair from the leaf node, Node $G$. However, when walking down the tree from Node $A$ to Node $C$, the query for key "/R/n" follows the parent-to-child pointer that lifts prefix "/R/". Therefore, it gets key "/R/n" after prepending lifted prefixes along the root-to-leaf path. Similarly, the query for key "/B/n" gets a result with key "/B/n".

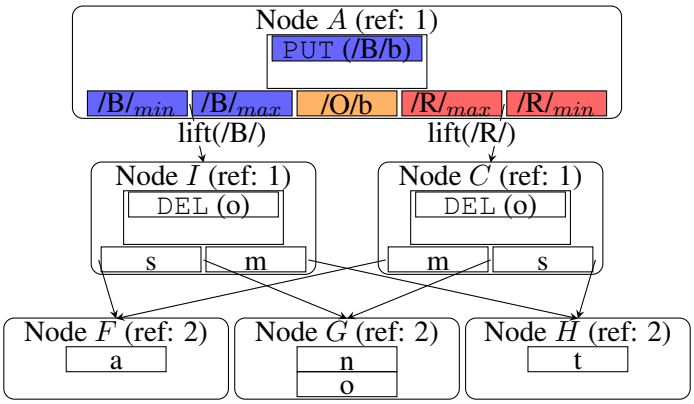
The lifted B$^\varepsilon$-DAG must break the sharing of a node when flushing messages through one of its parent-to-child pointer. Because these messages are injected into the tree after the range-clone operation, they are not shared by all parent-to-child pointers. Thus, the node can no longer be shared by multiple parents after flushing. However, before accumulating enough messages to flush to the subtree, the subtree is shared by multiple parent-to-child pointers, which saves a lot of space compared to the naive approach that clones the source subtree in the range-clone operation.

The lifted B$^\varepsilon$-DAG breaks the sharing of a node using Copy-on-Write (CoW). For instance, after the range-clone operation, one parent of the shared node might choose to flush its messages to the node. At this moment, the lifted B$^\varepsilon$-DAG creates a new node that is identical to the shared node, sharing the content and all children of the node. The lifted B$^\varepsilon$-DAG then sets the parent-to-child pointer in the parent to the new node and performs the flush.

Figure 5.2 shows an example of the CoW process. In this example, we also show the reference count of a node alongside the node ID. In Figure 5.2a, Node $A$ has two parent-to-child pointers to Node $C$. Therefore, the reference count of Node $C$ is 2 and Node $F$, $G$ and $H$ have reference count 1. When Node $A$ wants to flush messages with through the parent-to-child pointer between "/B/$_{min}$" and "/B/$_{max}$", the flushing process finds out that Node $C$ is a shared node because its reference count is greater than 1. Therefore, it clones Node $C$ to break the sharing. As shown in Figure 5.2b, the B$^\varepsilon$-DAG creates a new node, Node $I$, that is identical to Node $C$. Now,




(a) Node $C$ is a shared by two parent-to-child pointers in Node $A$.




(b) Before Node $A$ flushes "/B/" messages, it breaks the sharing by cloning Node $C$ to Node $I$.

Figure 5.2: When a parent want to flush messages to a shared node, it breaks the sharing by cloning the shared node.

both Node $C$ and $I$ have reference count 1, while Node $F$, $G$, and $H$ have reference count 2. With the sharing broken, the B$^\varepsilon$-DAG can flush message `PUT` ("/B/b",...) from Node $A$ to Node $I$ without affecting the other root-to-leaf path that includes Node $C$.

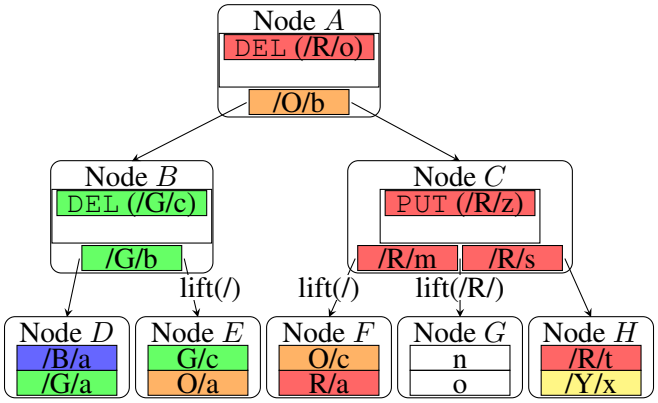
### 5.2.2 Range-clone with `GOTO` messages

Instead of completing all work on the critical path, range-clone can use a new type of message, the `GOTO` message, that delays tree surgery in the lifted B$^\varepsilon$-DAG.

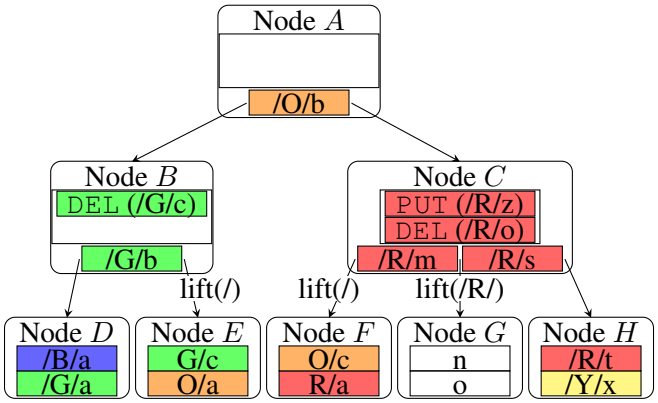
This subsection describes the `GOTO` message and how the range-clone operation can return to the user by injecting a `GOTO` message into the root node. The next subsection (Section 5.2.3) shows how the B$^\varepsilon$-tree flushes a `GOTO` message, completing all work in the range-clone operation.

**The `GOTO` message.** A `GOTO` message consists of 3 parts: a *dst_prefix*, a *src_prefix*, a node ID (with its height in the B$^\varepsilon$-DAG). Generally speaking, a `GOTO` message serves as an additional parent-to-child pointer in the B$^\varepsilon$-tree node with an `xf` (translate-and-filter) function. Rather than following normal parent-to-child pointers in the node, a query whose search key falls in the key range (*dst_prefix* $_{min}$, *dst_prefix* $_{max}$) must follow the `GOTO` message, that is, the next node the query visits should be the node whose node ID is in the `GOTO` message. Also, the query should update its search key by replacing prefix *dst_prefix* with prefix *src_prefix*. Later, the query needs to reconstruct the result by replacing prefix *src_prefix* with prefix *dst_prefix* in the key. Therefore, the `GOTO` message should filter out keys that are not in the key range (*src_prefix* $_{min}$, *src_prefix* $_{max}$) in the subsequent search.

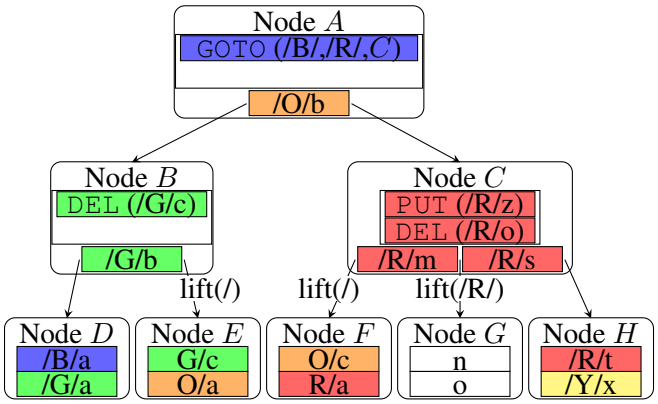
For example, in Figure 5.3c, consider a query for key "/B/a". The query first visits the root node, Node $A$, and finds a `GOTO` message, `GOTO` ("/B/", "/R", $C$). This query should follow the `GOTO` message and visit Node $C$, instead of the normal parent-to-child pointer to Node $B$. Additionally, the `GOTO` message acts as an `xf` function that translates the search key from "/B/a" to "/R/a" and bounds the query result to ("/R/$_{min}$", "/R/$_{max}$"). After fetching the key/value pair with key "/R/a" in Node $F$, the query reconstructs the key by replacing prefix "/R/" with prefix




(a) The B$^\varepsilon$-tree before the range-clone operation.




(b) The range-clone operation reaches the source LCA, Node $C$, flushing messages along the way.




(c) The range-clone operation injects a GOTO message for destination ("/B/") keys into the root node.

Figure 5.3: An example of completing range-clone("/R/", "/B/") with a GOTO message.

"/B/". Therefore, the query returns the key/value pair with key "/B/a". To see why the `GOTO` message needs to bound the query result, consider a range query for a key that is greater than "/B/z". The `GOTO` message in Node $A$ translates the search key to "/R/z". However, searching for a key greater than "/R/z" in the subtree rooted at Node $C$ returns key "/Y/x", which the query cannot reconstruct by replacing prefix "/R/" with prefix "/B/". In fact, the `GOTO` message is only for results whose keys fall in the key range ("/B/$_{min}$", "/B/$_{max}$"), therefore, any key that is not in the key range ("/R/$_{min}$", "/R/$_{max}$") is invalid after following the `GOTO` message.

Because a `GOTO` message redirects queries whose search keys have a certain prefix, it also acts as a range-delete message that invalidates all old messages and key/value pairs following other pointers (normal parent-to-child pointers and older `GOTO` messages). Thus, if we set the node ID in a `GOTO` message to a special value (for example, 0), the `GOTO` message acts as a range-delete message. Therefore, we can use `GOTO` messages to implement range-delete.

Though a `GOTO` message acts as an additional parent-to-child pointer in the node, the child of the `GOTO` message doesn't need to be at the same height as normal children of the node. Therefore, the `GOTO` message let us share a subtree between two parents at different heights.

Also, the `xf` function associated with the `GOTO` message means the target subtree can have a larger key range than specified in the `GOTO` message. Thus, the range-clone operation can lazily slice out the subtrees, amortizing the slicing cost with other operations.

**Range-clone with `GOTO` messages.** Range-clone(*src_prefix*, *dst_prefix*) is implemented with `GOTO` messages through the following steps:

- the range-clone operation does a root-to-leaf traversal with *src_prefix* until reaching the source LCA, the lowest node that covers the whole key range (*src_prefix* $_{min}$, *src_prefix* $_{max}$). During the traversal, the range-clone operation also flushes messages from parent to child. At this point, all messages with keys in the cloned key range are in the subtree rooted at the source LCA.

- the range-clone message then injects a `GOTO` message into the root node of the lifted B$^\varepsilon$-DAG with the source LCA as the node ID in the `GOTO` message, increasing the reference count of the source LCA by 1.

Briefly speaking, on the critical path, the range-clone operation flushes messages until the source LCA, increments the reference count of the source LCA, and injects a `GOTO` message into the root node. Note, this range-clone operation doesn't do tree surgery or pointer swings on the critical path.

Figure 5.3 shows an example of range-clone("/R/", "/B/") that generates a `GOTO` message. Figure 5.3a shows the B$^\varepsilon$-tree before the range-clone operation. Starting from the root node, Node $A$, the range-clone operation traverses to the source LCA, Node $C$. At the same time, it flushes the message, `DEL` ("/R/z"), to Node $C$. Figure 5.3b shows the B$^\varepsilon$-tree after the range-clone operation reaches the source LCA. Finally, the range-clone operation injects a `GOTO` message into the root node, with "/B/" as its *dst_prefix*, "/R/" as its *src_prefix* and Node $C$ as its node ID. Figure 5.3c shows the result. Now, consider a query for key "/B/a". The query starts at the Node $A$. Then, instead of following the parent-to-child pointers to Node $B$, it follows the `GOTO` message and visits Node $C$. The `GOTO` message also acts as an `xf` function that translates the search key to "/R/a" and bounds the result in range ("/R/$_{min}$", "/R/$_{max}$"). At last, the query finds the correct key/value pair in Node $F$.

### 5.2.3 Flushing `GOTO` messages

A `GOTO` message prevents the B$^\varepsilon$-tree from flushing future messages that fall in the key range of the `GOTO` message, because queries will not follow the normal parent-to-child pointer and find the messages once flushed. Therefore, the lifted B$^\varepsilon$-DAG should flush `GOTO` messages along with other messages.

However, flushing a `GOTO` message to a node at the same height as the source LCA breaks the asymptotic I/O costs of queries in the lifted B$^\varepsilon$-DAG, because such a `GOTO` message redirects queries to a node at the same height. Therefore, when flushing a `GOTO` message to a node at the

same height as the source LCA, the lifted B$^\varepsilon$-DAG transfers the GOTO message to pivots and a parent-to-child pointer.

This subsection explains how the lifted B$^\varepsilon$-DAG flushes GOTO messages. We start from the simple case, where a GOTO message is being flushed to nodes higher than the source LCA. Then, we describe how a GOTO message becomes pivots and a parent-to-child pointer when the lifted B$^\varepsilon$-DAG flushes the GOTO message to a node at the same height as the source LCA.

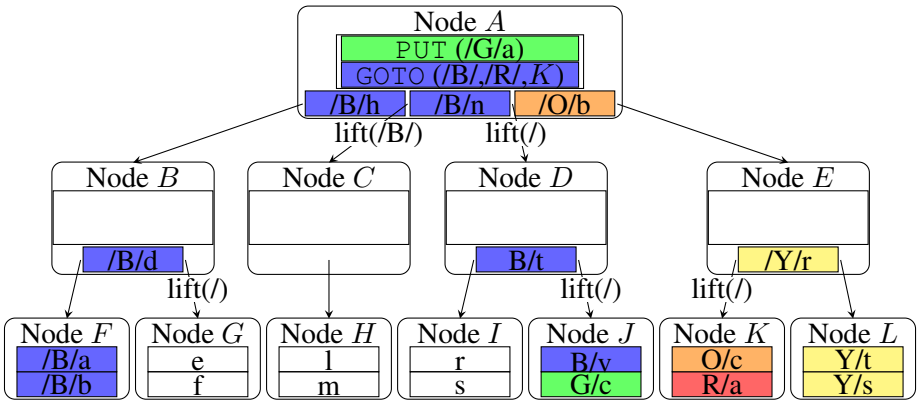
**The lifted B$^\varepsilon$-DAG flushes a GOTO message to a node higher than the source LCA.** In the simple case, the node that tries to flush a GOTO message has a single child that covers the key range of the GOTO message and the child is higher than the source LCA of the GOTO message. Flushing the GOTO message simply lifts the *dst_prefix* of the GOTO message and moves the GOTO message from the node's buffer to the child's buffer.

For example, in Figure 5.4b, one child of Node $A$, Node $B'$, covers the whole key range of the GOTO message, ("/B/$_{min}$", "/B/$_{max}$"). Therefore, the lifted B$^\varepsilon$-DAG can flush the GOTO message to Node $B'$, as shown in Figure 5.4c.

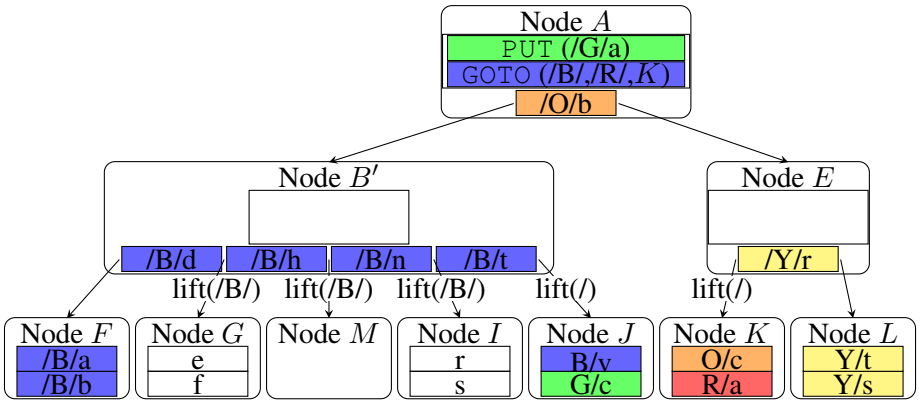
In the more complicated case, a GOTO message can overlap with the key ranges of multiple children. One solution is to duplicate the GOTO message and flush to all children. However, because duplicating the GOTO message increases the reference count of the source LCA, the source LCA ends up being shared by many duplicated GOTO messages, making the sharing complicated.

Our solution is to generate a single child that can accommodate the GOTO message. An easy way is to merge all children whose key range overlaps with the GOTO message. However, this may create a node with a huge fanout. In fact, because the GOTO message invalidates all old keys with *dst_prefix*, we can decrease the reference counts (and potentially garbage collect) of all children whose key ranges fall completely in the key range of the GOTO message (we call these children *interior children*) and merge the two children whose key ranges partly overlap with the GOTO message (we call these two children *fringe children*).

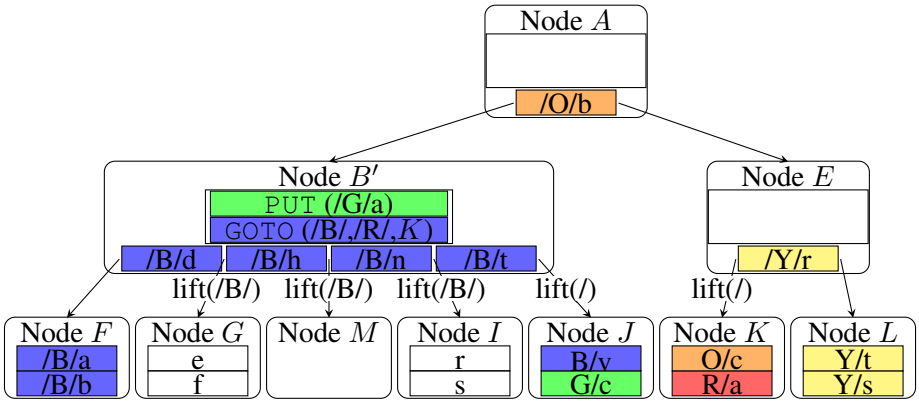
Figure 5.4 shows an example of merging children before flushing a GOTO message. In Figure 5.4a, Node $B$, $C$ and $D$ all overlap with the range of the GOTO message. Node $B$ AND $D$




(a) The lifted B$^\varepsilon$-DAG cannot not flushed the GOTO message to a single child of Node $A$.




(b) The lifted B$^\varepsilon$-DAG garbage collects Node $C$ and merges Node $B$ and $D$ into Node $B'$.




(c) The lifted B$^\varepsilon$-DAG flushes the GOTO message to Node $B'$.

Figure 5.4: When a node tries to flush a GOTO message, there may be more than one child that overlaps with the key range of the GOTO message. In this case, the B$^\varepsilon$-DAG must merge children to create a single child that can accommodate the GOTO message.

are fringe children, while Node $C$ is an interior child. Note, though the range-rename operation that generates the GOTO message has "/R/" as the *src_prefix*, the *src_prefix* in the GOTO message is "R/" because "/" is lifted from the source LCA, Node $K$. To flush the GOTO message, in Figure 5.4b, the lifted B$^\varepsilon$-DAG garbage collects Node $C$ and merges Node $B$ and $D$ into Node $B'$. Note in the example, the merge creates an empty node, Node $M$, to cover the key range ("/B/h", "/B/q"). After the merge, Node $A$ flushes the GOTO message to Node $B'$, as shown in Figure 5.4c.

In the exmaple, we create an empty node, Node $M$, to cover the key range of garbage-collected nodes. Otherwise, we need to enlarge the key range of either Node $G$ or Node $I$. However, with key lifting, changing the key range of a node means re-lifting the node and its descendants, which means additioanl I/Os during a flush.

In special cases, there can be only one or no fringe child. If there is only one fringe child, the flush process removes the interior children and adds an empty subtree to cover the key range as one child of the fringe child. If there is no fringe child, the flush process removes the interior child and adds an empty subtree to cover the key range as the child of the node.

**Converting GOTO messages to pivots and parent-to-child pointers.** After the merging process, there is only one child whose key range overlaps with the key range of the GOTO message. If the child is higher than the source LCA of the GOTO message, we can simply flush the GOTO message to the child buffer. However, if the child is at the same height as the source LCA, we cannot flush the GOTO message. In such scenarios, the GOTO message is converted into pivots and a parent-to-child pointer in the node.

The lifted B$^\varepsilon$-DAG completes this conversion by adding two new pivots, *dst_prefix* $_{min}$ and *dst_prefix* $_{max}$ and setting the new parent-to-child pointer to the source LCA of the GOTO message. Assume *dst_prefix* $_{min}$ and *dst_prefix* $_{max}$ are in the key range of child $i$ of the node, that is, $\textit{dst\_prefix}_{min}, \textit{dst\_prefix}_{max} \in (pivot_i, pivot_{i+1})$. Adding *dst_prefix* $_{min}$ and *dst_prefix* $_{max}$ as two new pivots creates 3 key ranges, $(pivot_i, \textit{dst\_prefix}_{min})$, $(\textit{dst\_prefix}_{min}, \textit{dst\_prefix}_{max})$ and $(\textit{dst\_prefix}_{max}, pivot_{i+1})$. Key range $(\textit{dst\_prefix}_{min}, \textit{dst\_prefix}_{max})$ is covered in the source LCA of the GOTO message, while the other two key range are covered by child $i$.

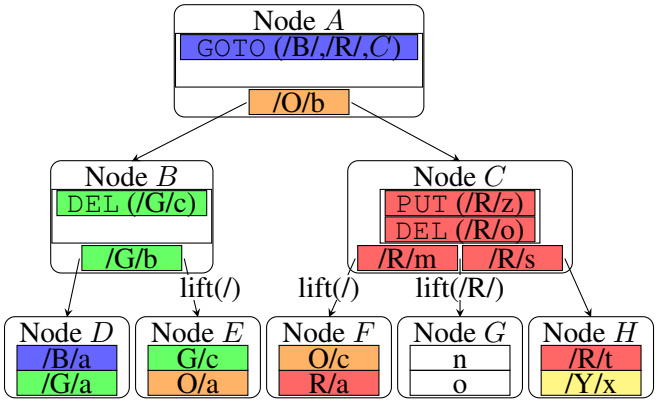
All three parent-to-child pointers now point to children whose key ranges are different than those specified in the node. In particular, child $i$ has range $(pivot_i, pivot_{i+1})$ but bounded by $(pivot_i, \textit{dst\_prefix}_{min})$ or $(\textit{dst\_prefix}_{max}, pivot_{i+1})$. The source LCA has a key range that might be larger than $(\textit{src\_prefix}_{min}, \textit{src\_prefix}_{max})$ but bounded by $(\textit{dst\_prefix}_{min}, \textit{dst\_prefix}_{max})$. A smaller key range may lift a longer prefix through the parent-to-child pointer.

The problem is solved by augmenting parent-to-child pointers with `xf` functions, which are the same as the `xf` function described in `GOTO` messages. Each parent-to-child pointer now has a `xf` function with a prefix. After key lifting lifts the search keys of queries by the LCP of two pivots, the `xf` function prepends its prefix to the search key. Also, `xf` functions serves as filters, bounding the results queries may return.

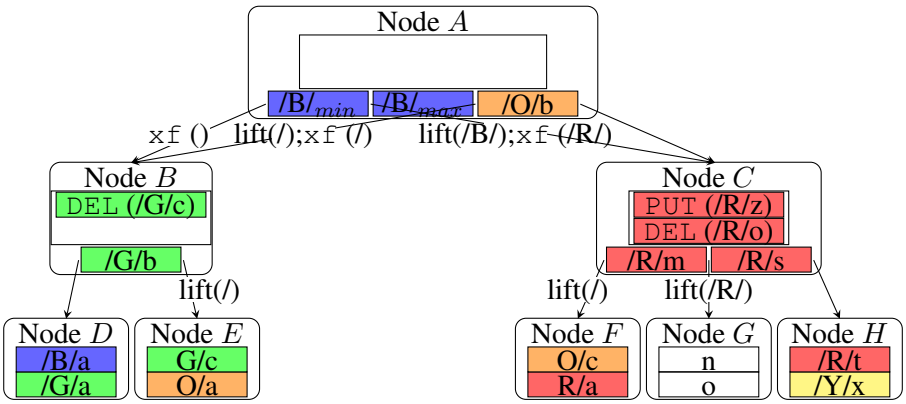
Figure 5.5 shows an example of the transformation. The lifted B$^\varepsilon$-DAG cannot flush the `GOTO` message from Node $A$ to Node $B$, because Node $B$ is at the same height as its source LCA, Node $C$. Therefore, two pivots, "/B/$_{min}$" and "/B/$_{max}$", are added to Node $A$. The parent-to-child pointer between "/B/$_{min}$" and "/B/$_{max}$" points to the source LCA, Node $C$, and has a `xf` function of "/R/", indicating a key range mismatch and the additional prefix "/R/" for keys in the child. Queries following that parent-to-child pointer should prepend prefix "/R/" after the normal lifting of "/B/" from its search key. Also, queries remember only keys between "/R/$_{min}$" and "/R/$_{max}$" in the child are valid. Similarly, two `xf` functions are added to the other two pointers. Note, for the leftmost parent-to-child pointer of Node $A$, though the `xf` function contains no prefix, it still serves as a filter for queries.

With the help of `xf` functions in parent-to-child pointers, the B$^\varepsilon$-DAG can transfer a `GOTO` message into two pivots and a parent-to-child pointer. Now, let's look at how `xf` functions are resolved during flushes, finishing the rest work in the range-clone operation.

**Fixing `xf` functions in parent-to-child pointers.** `GOTO` messages add `xf` functions to parent-to-child pointers. The `xf` function transforms the search keys of queries by prepending its prefix after normal key lifting through parent-to-child pointers. Also, the `xf` function indicates that the

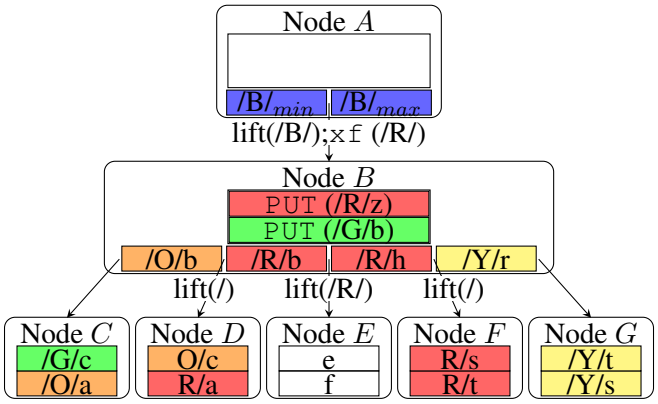



(a) The GOTO message cannot be flushed to Node $B$, which is at the same as Node $C$.

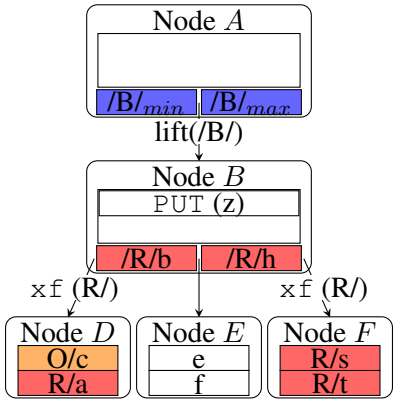



(b) The GOTO message becomes pivots and a parent-to-child pointer, and adds xf functions to the parent-to-child pointers.

Figure 5.5: When the lifted B$^\varepsilon$-DAG cannot flush a GOTO message deeper, it transforms the GOTO message into pivots and a parent-to-child pointers and adds xf functions to the parent-to-child pointers.




(a) The parent-to-child pointer from Node $A$ to Node $B$ has `xf` ("/R/").




(b) To remove the `xf` function, it discards exterior children and adds `xf` functions to parent-to-child pointers of fringe children.

Figure 5.6: During node flushes, the lifted B$^\varepsilon$-DAG resolves the `xf` function associated with the parent-to-child pointer.

child contains keys outside of the key range specified by pivots in the parent and thus, queries should ignore those keys following the parent-to-child pointers.

The lifted B$^\varepsilon$-DAG resolves `xf` functions through node flushes. Specifically, when flushing from a parent to a child and the parent-to-child pointer has an `xf` function, the lifted B$^\varepsilon$-DAG garbage collects exterior children of the child whose key ranges are completely outside of the key range specified by the pivots in parent. Also, the lifted B$^\varepsilon$-DAG removes messages whose keys are outside of the key range specified in the parent from the child buffer. Then, for fringe children whose key ranges partly overlap with that specified in the parent, the B$^\varepsilon$-DAG propagates the `xf` function to their parent-to-child pointers. Finally, if the `xf` function has a prefix, the B$^\varepsilon$-DAG lifts the prefix from keys in the child.

Figure 5.6 shows an example of fixing `xf` functions in node flushes. In Figure 5.6a, the parent-to-child pointer from Node $A$ to Node $B$ has an `xf` function, `xf` ("/R/"). The key range specified by Node $A$ is ("/B/$_{min}$","/B/$_{max}$"), and the `xf` function translates the key range to translates to ("/R/$_{min}$","/R/$_{max}$") in Node $B$. Also, the `xf` function tells queries to ignore pivots and messages that are out of the key range ("/R/$_{min}$","/R/$_{max}$"). In Figure 5.6b, the lifted B$^{\varepsilon}$-DAG garbage collects exterior nodes, Node $C$ and $G$ and adds `xf` functions to pointers to fringe nodes, Node $D$ and $F$. Also, it updates keys in Node $B$ by removing the prefix, "/R/", in the `xf` function. Note, the message `PUT` (/G/a,...) is also removed from Node $B$ because it is out of the key range specified in Node $A$.

## 5.3 Implementation

This section discusses implementation details of the range-clone operation.

### 5.3.1 Synchronization

The range-clone operation synchronizes with other B$^{\varepsilon}$-tree operations using the readers-writer locks of B$^{\varepsilon}$-tree nodes (Section 3.4.1). The range-clone operation first grabs the write lock of the root node. Then, while flushing messages along the root-to-leaf path until the LCA, the range-clone operation write-locks B$^{\varepsilon}$-tree nodes, hand-over-hand, holding the write lock of the root node. Finally, the range-clone operation injects the `GOTO` message into the root node and releases the write locks of the root node and the LCA.

Compared to the range-rename operation that unlocks the root node after flushing messages (Section 4.3.1), the range-clone operations locks the root node for a longer time. However, the range-clone operation returns to the caller earlier, while the range-rename operation needs to lock the LCAs during bottom-up slicing, preventing concurrent operations to the subtrees.

### 5.3.2 Preferential splitting

Most of the work in range-clone and range-rename is about splitting fringe nodes, separating keys with the prefix and keys without the prefix. If all nodes are either interior nodes or exterior nodes, much less work is needed. Therefore, it is beneficial to reduce the number of fringe nodes.

To this end, we introduce preferential splitting in node splits. Originally, *ft-index* splits leaf nodes evenly. When a leaf node needs to be split, the middle key in the leaf is picked as the new pivot that separates the two new leaf nodes.

Preferential splitting generates pivots that are potential splitting keys in file system renames and maximizes the common prefix under the leaf, subject to the constraint that both leaves should be at least 1/4 full. This strategy reduces the likelihood of having fringe nodes in range-clone and bounds how unbalanced leaves can be.

A naive approach would compare all keys in the range of [1/4, 3/4] of the leaf node and pick the pair of two adjacent keys that share the shortest common prefix. But this scan can be costly. For example, in BetrFS, a full leaf node is 4 MiB in size and each key/value pair in `meta_db` is less than 200 Bytes, which means more than 10000 key comparisons are required in this naive preferential splitting.

In fact, we can do preferential splitting that only requires the reading of two keys. Because the shortest common prefix of adjacent keys is the same as the common prefix of the smallest (at 1/4 of the leaf) candidate key, $k_s$, and the largest (at 3/4 of the leaf) candidate key, $k_l$, a good pivot can be constructed from these two keys. In particular, preferential splitting generates a pivot that is the maximum key with prefix $p_s$, where $p_s$ is the shortest prefix of $k_s$ that contains the LCP of $k_s$ and $k_l$ and has a slash or a end-of-string as the last character. For instance, if $k_s$ is "/foo/bar" and $k_l$ is "/fuz/bar", preferential splitting set $p_s$ as "/foo/", which is the shortest prefix of $k_s$ that contains the LCP of $k_s$ and $k_l$, "/f", and has a slash at the end. Therefore, the pivot generated by preferential splitting is the maximum key with prefix "/foo/" (adding a '\xff').

Non-leaf-node splits can be viewed as promoting a pivot from the child to the parent. Because the number of pivots in a non-leaf node is small (at most 16), we can afford one-to-one comparisons.

As we will see in the evaluation, preferential splitting also helps other operations, such as sequential writes, because the $B^{\varepsilon}$-tree is more likely to generate a pivot that is the maximum data key of a file or directory. In the sequential write workload, all pivots generated by node splits are data keys of the file being written. Generating the maximum data key of the file at the beginning of the workload reduces node re-lifting costs.

### 5.3.3 Queries

As described in Section 3.4.4, during a query, *ft-index* applies messages in non-leaf nodes along the root-to-leaf path to the leaf nodes.

However, in a $B^{\varepsilon}$-DAG, there can be multiple paths from the root node to a leaf node, which will accumulate a different set of pending messages. Therefore, in our implementation, we must cache different versions of the same node that is shared on disk — one per path. In addition to the original identifier, the node ID, in the cahe table, we add the key range as a sub-identifier. However, this approach can potentially create more cache pressure, which can lead to more node evictions and I/Os.

We adopt a hybrid approach in the implementation. If a leaf has only one version, i.e., all nodes along any path to it have reference count 1, messages are applied to the leaf directly. Otherwise, the leaf node maintains one additional copy for each version.

### 5.3.4 Garbage collection

When a leaf node's reference count drops to zero, it is simply marked as free in the block table, and the space on disk can be re-allocated after the next checkpoint. If the system crashes before the next checkpoint, crash recovery will replay the redo log on the previous checkpoint. Therefore, it is only safe to reuse space in the block table after a checkpoint (which includes a

snapshot of the block table) and we defer re-allocation of physical space until after a checkpoint When an interior node's reference count drops to zero, the complication is that, as part of freeing the node, the reference count on each of its children must be lowered, and the children recursively freed.

Our $B^\varepsilon$-DAG implementation does this work with background threads, although, in future work, we could also trigger foreground garbage collection as part of flushing if free space falls below a threshold. We track the pending work with an in-memory work queue and an extra flag in the block table; the flag in the block table ensures that checkpointing the system is not obstructed by garbage collection, and ensures that pending garbage collection work and space is not lost upon a crash.

## 5.4 Conclusion

This chapter presents the new range-clone operation. We first describe a range-clone implementation that finishes all works on the critical path and transforms lifted $B^\varepsilon$-trees into lifted $B^\varepsilon$-DAGs. Then, we introduce the `GOTO` message that delays tree surgery, fitting the implementation into the write-optimized framework of lifted $B^\varepsilon$-DAGs. At last, we show how `GOTO` messages should be flushed down the lifted $B^\varepsilon$-DAGs. A `GOTO` message will eventually become pivots and a parent-to-child pointer in the lifted $B^\varepsilon$-DAG.

Full-path-indexed BetrFS clones a directory without traversing the directory because full-path indexing specifies the key range inside the directory. In contrast, to finish a directory clone, an inode-based file system needs to traverse the directory to collect all inodes inside the directory. When the number of inodes in the directory is large, cloning the directory is slower in inode-based file systems than that in full-path-indexed BetrFS. Additionally, the directory clones in full-path-indexed BetrFS still maintains full-path indexing, which ensures good locality.

Therefore, full-path indexing is not an obstacle to namespace operations. In fact, full-path indexing creates more opportunities for namespace operations.

Similar to the range-rename operation, the technique is not limited to the range-clone operation that clones keys with certain prefix. For a key/value store operation that clones a lot of keys, one can use a similar technique as long as the key/value store can group all related keys in a contiguous key range and there is an `xf` function that translates sources keys to destination keys.

# CHAPTER 6: EVALUATION

This chapter evaluates the performance of BetrFS with range-rename (BetrFS 0.4) and BetrFS with range-clone (BetrFS 0.5), comparing them with widely used file systems and BetrFS 0.3. BetrFS 0.3 is the relative-path-indexed BetrFS that penalizes other file system operations for good namespace operations. The evaluation includes both micro and macro benchmarks.

We seek to answer the following questions in the evaluation:

- How do non-namespace operations perform on BetrFS 0.4 and BetrFS 0.5?
- How do namespace operations perform on BetrFS 0.4 and BetrFS 0.5?
- Do applications perform well on BetrFS 0.4 and BetrFS 0.5?
- Do BetrFS 0.4 and BetrFS 0.5 maintain good performance over time?

**Experimental Setup.** All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7-2600 CPU, 4GB RAM, and a 128 GiB partition (the rest of the disk is not used) of a 500 GB, 7200 SATA disk with a 4096-byte block size (Seagate Barracuda ST500DM002). The system runs 64-bit Ubuntu server 14.04.6 on a USB stick to prevent interference from the root file system. BetrFS 0.3 and BetrFS 0.4 run on a modified Linux 3.11.10 kernel that enlarges the size of the kernel stack, while BetrFS 0.5 and all other file systems run on unmodified Linux 4.9.142 kernel. The evaluation uses ZFS 0.6.5.11 from `zfsonlinx.org` and ext4, Btrfs, XFS and NILFS2 as parts of the Linux kernel. Each experiment runs a minimum of 5 times and reports the average number. Error bars and error $\pm$ terms indicat 95% confidence intervals over all runs. Unless noted, all benchmarks are cold-cache tests.

We run benchmarks on empty file systems and aged file systems. We age the file system by emulating user behaviors. First, we fill the file system with files and directories from the root

file system, roughly taking up 31 GiB out of 128GiB space. Then, we clone the git repository of the Linux kernel with the release tag "v3.12" (released on 2013/11/03) into the home directory in the file system. Afterwards, we repeat the process of building the kernel and pulling the next elease of the repository until we build the Linux kernel with the release tag "v4.20" (released on 2018/12/13). This process takes several days to complete (about 5 days on ext4), building the kernel 240 times (though there are only 29 major releases, there are 7 to 9 release candidates between two consecutive major releases) and pulling 397,481 git commits. The directory of the git repository takes up about 3.6 GiB space and after building the kernel, it takes up about 15 GiB space. Therefore, after the aging process, the whole file system takes up about 46 GiB space, about 36% of the partition size. Unless otherwise noted, when benchmarking on aged file systems, all operations are performed in the aged git repository.

## 6.1 Microbenchmarks

In this section we run file systems on microbenchmarks, each measures the performance of one file system operations. We divide file system operations into non-namespace operations (Section 6.1.1) and namespace operations (Section 6.1.2).

### 6.1.1 Non-namespace operations

**Sequential writes and reads.** We measure the throughput of sequentially writing and reading a file. This benchmark first writes a 10GiB file, 40MiB at a time, with an `fsync` to flush the file to the disk. Then, after clearing the kernel page cache, the benchmark reads the file from the disk using `dd` with a 10MiB block size.

Figure 6.1 shows the results of running the benchmark on empty file systems. Ext4, Btrfs and XFS can sequentially write close to disk bandwidth, while BetrFS 0.5, similar to NILFS2, is about 6.5% slower than the fastest file system. In the `blktrace`, we find that, while most of the I/Os generated by ext4 write 2048 sectors (256MiB), BetrFS 0.5 generates small I/Os (4KiB or 8KiB) in between large I/Os due to log writes. BetrFS 0.4 is slightly slower than BetrFS 0.3

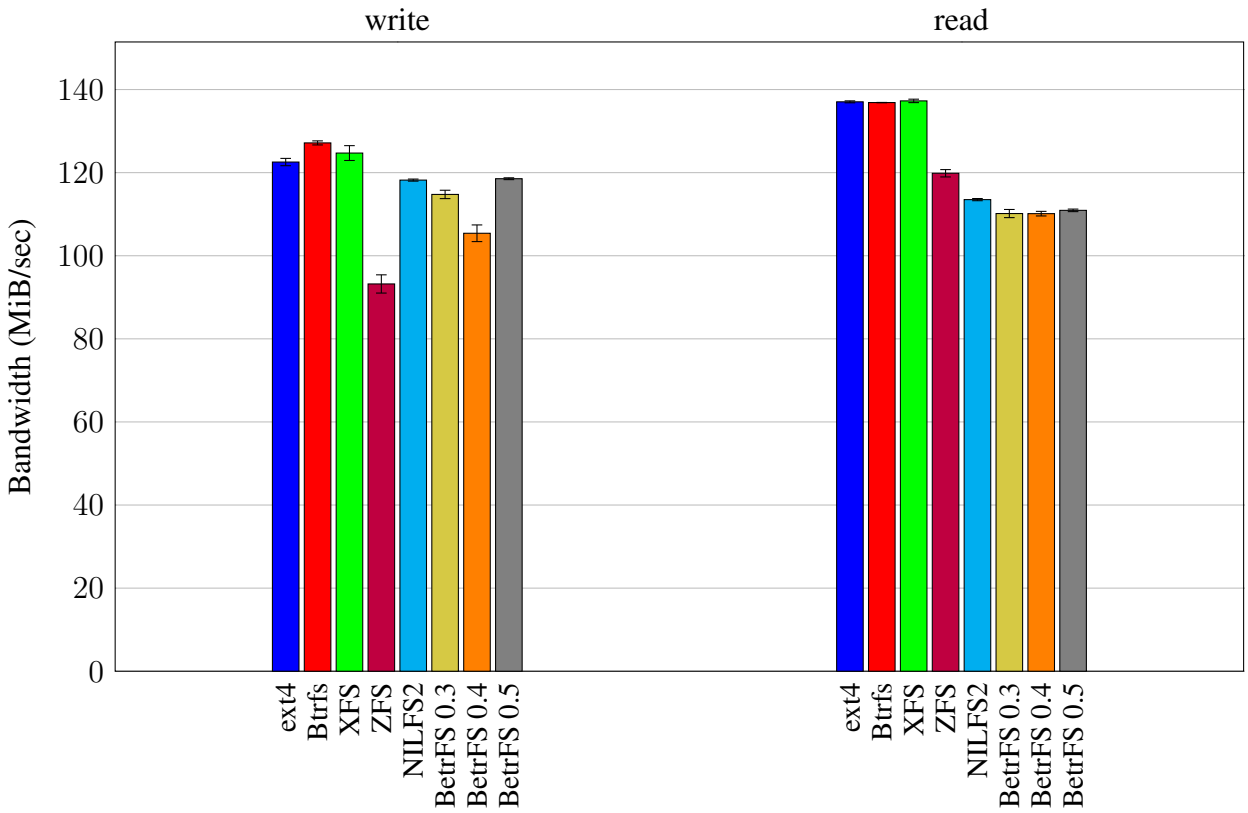



Figure 6.1: Bandwidth to sequentially read and write a 10 GiB file on empty file systems (higher is better).

because of the re-lifting cost in node splits. The performance increase of BetrFS 0.5 from BetrFS 0.4 is from preferential splitting (Section 5.3.2), which creates a pivot matching the maximum file data key at the beginning of the workload, avoiding future node re-lifting in subsequent node splits as the file grows. For sequential reads, Ext4, Btrfs, and XFS run at disk bandwidth, while BetrFS 0.5 is 19.1% slower than the fastest file system, which is close to BetrFS 0.4 and NILFS2. The `blktrace` shows most of the I/Os issued by both ext4 and BetrFS 0.5 read 256 sectors (32 MiB). However, the I/Os generated by ext4 are contiguous in the address space, while BetrFS 0.5 constantly issues two consecutive I/Os that are far away from each other when BetrFS 0.5 switches to read a different node.

Figure 6.2 shows the results of running the benchmark on aged file systems. The performances of ext4 and XFS are similar to those on empty file systems, showing that most of the free spaces are not fragmented by the aging process. Btrfs, ZFS, BetrFS 0.3, BetrFS 0.4, and BetrFS

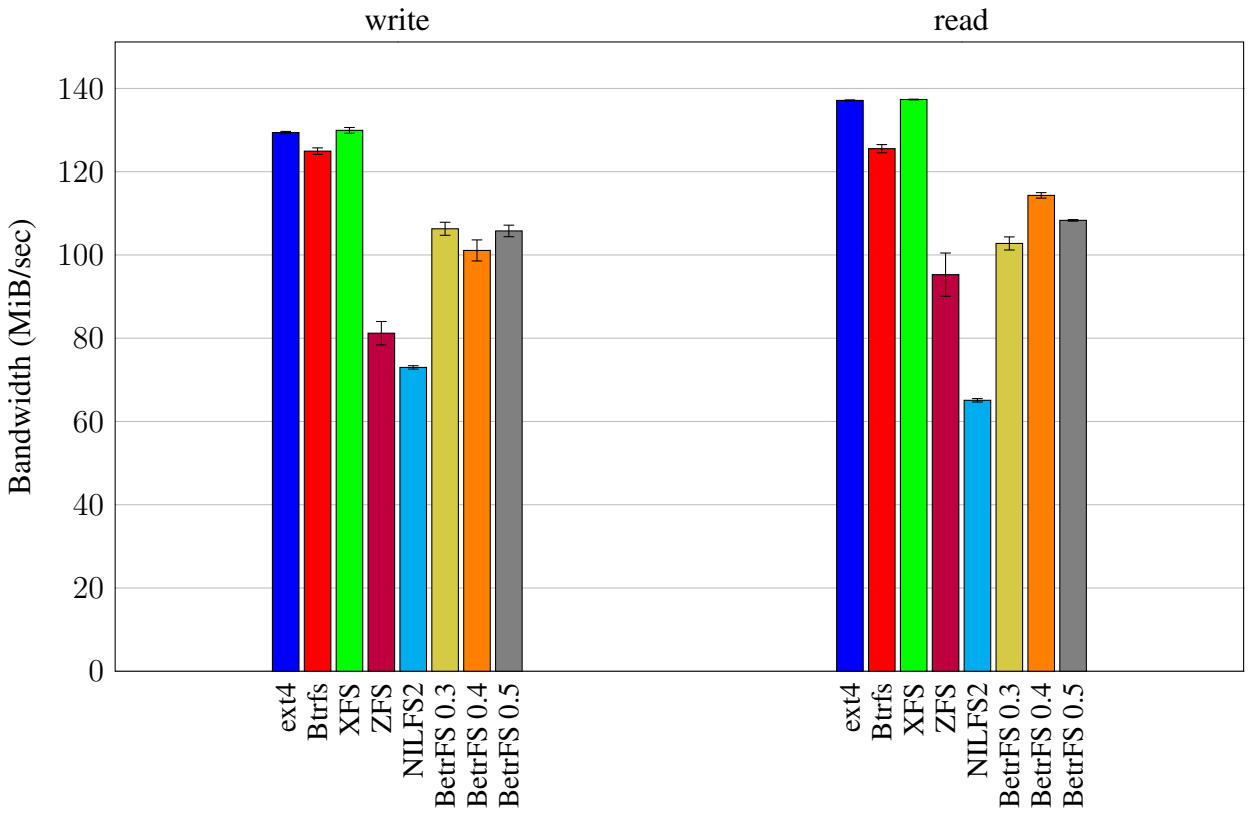



Figure 6.2: Bandwidth to sequentially read and write a 10 GiB file on aged file systems (higher is better).

0.5 store parts of metadata and data in trees, which are taller for non-empty file systems. Because operations on taller trees are more expensive, Btrfs, ZFS, BetrFS 0.3, BetrFS 0.4, and BetrFS 0.5 become slightly slower in the aged setting. NILFS2 becomes much slower because the aging process fills the log multiple times. To generate free spaces, NILFS2 needs to garbage collect the log, resulting in additional I/Os. Moreover, the free spaces generated by garbage collection scatter over the disk, therefore, NILFS2 needs to issue more random I/Os during the benchmark.

**Random writes.** We then measure the performance of random writes on the file generated by the sequential write benchmark. The benchmark issues 256K (262,144) 4-Byte overwrites (in total 1 MiB data) at random offsets within the 10GiB file, followed by an `fsync`.

Table 6.1a shows the results of running the benchmark on empty file systems. Because BetrFS performs upserts, which doesn't read the old data, for random writes, performing the 1MiB random writes only takes about 5 seconds on all versions of BetrFS. However, it takes at least

| File system | random write (sec) |
|---|---|
| ext4 | 2809.415 ± 6.273 |
| Btrfs | 2065.304 ± 7.767 |
| XFS | 2888.394 ± 26.987 |
| ZFS | 2881.419 ± 44.429 |
| NILFS2 | 2023.282 ± 1.902 |
| BetrFS 0.3 | 4.936 ± 0.109 |
| BetrFS 0.4 | 4.871 ± 0.075 |
| BetrFS 0.5 | 5.478 ± 0.070 |

(a) Benchmarking on empty file systems.

| File system | random write (sec) |
|---|---|
| ext4 | 2787.738 ± 6.130 |
| Btrfs | 2264.590 ± 13.842 |
| XFS | 3141.709 ± 68.312 |
| ZFS | 3238.939 ± 72.113 |
| NILFS2 | 2174.144 ± 139.910 |
| BetrFS 0.3 | 5.207 ± 0.109 |
| BetrFS 0.4 | 4.837 ± 0.027 |
| BetrFS 0.5 | 4.593 ± 0.058 |

(b) Benchmarking on aged file systems.

Table 6.1: Time to perform 256K (262,144) 4-Byte random writes on a 10GiB file (1 MiB total IO, lower is better).

2000 seconds on the other file systems, which is more than 350 times slower. After the sequential write process, different versions of BetrFS have different $B^\varepsilon$-trees, resulting in different numbers of node flushes during the benchmark. Therefore, different versions of BetrFS have different performances in the benchmark.

Table 6.1b shows the results of running the benchmark on aged file systems, which are similar to those on empty file systems.

**Directory operations.** Next, we measure three common directory operations, grep, find, and delete. The grep benchmark measures the time to grep keyword cpu_to_be64 on the Linux-3.11.10 source directory. The find benchmark measures the time to find file wait.c on the same directory. And the delete benchmark measures the time to recursively delete the directory with "rm -rf".

Table 6.2a shows the results of running the benchmark on empty file systems. Because full-path indexing ensures locality in BetrFS, find and grep on BetrFS 0.4 and BetrFS 0.5 are more than two times faster than the other file systems. BetrFS 0.3 is slower than BetrFS 0.4 and BetrFS 0.5, but still faster than the other file systems in the find and grep benchmarks because of relative-path indexing. In BetrFS 0.3 and BetrFS 0.4, file deletes in the recursive delete benchmark are implemented with range-delete messages, while in BetrFS 0.5, file deletes are implemented with

| File system | grep (sec) | | find (sec) | | delete (sec) | |
|---|---|---|---|---|---|---|
| ext4 | 37.795 ± | 1.145 | 2.224 ± | 0.070 | 2.989 ± | 0.372 |
| Btrfs | 9.265 ± | 0.139 | 1.121 ± | 0.041 | 3.029 ± | 0.127 |
| XFS | 48.130 ± | 0.206 | 5.259 ± | 0.209 | 15.423 ± | 0.722 |
| ZFS | 463.184 ± | 33.878 | 9.101 ± | 0.041 | 9.026 ± | 0.396 |
| NILFS2 | 8.318 ± | 0.107 | 6.515 ± | 0.023 | 9.726 ± | 0.364 |
| BetrFS 0.3 | 5.070 ± | 0.078 | 0.217 ± | 0.012 | 2.713 ± | 0.097 |
| BetrFS 0.4 | 3.979 ± | 0.160 | 0.201 ± | 0.008 | 3.305 ± | 0.299 |
| BetrFS 0.5 | 3.991 ± | 0.067 | 0.198 ± | 0.004 | 2.668 ± | 0.152 |

(a) Benchmarking on empty file systems.

| File system | grep (sec) | | find (sec) | | delete (sec) | |
|---|---|---|---|---|---|---|
| ext4 | 68.521 ± | 0.199 | 6.384 ± | 0.048 | 11.683 ± | 0.858 |
| Btrfs | 138.787 ± | 0.683 | 2.675 ± | 0.127 | 5.700 ± | 0.397 |
| XFS | 141.669 ± | 0.842 | 14.992 ± | 0.022 | 27.021 ± | 0.864 |
| ZFS | 452.775 ± | 15.318 | 10.219 ± | 0.718 | 10.381 ± | 0.645 |
| NILFS2 | 9.124 ± | 0.091 | 7.328 ± | 0.054 | 10.927 ± | 0.926 |
| BetrFS 0.3 | 7.127 ± | 0.076 | 0.390 ± | 0.006 | 14.737 ± | 0.249 |
| BetrFS 0.4 | 5.031 ± | 0.052 | 0.315 ± | 0.007 | 35.660 ± | 0.989 |
| BetrFS 0.5 | 5.031 ± | 0.112 | 0.307 ± | 0.010 | 3.642 ± | 0.297 |

(b) Benchmarking on aged file systems.

Table 6.2: Time to perform recursive grep, find and delete of the Linux source directory (lower is better).

`GOTO` messages. All versions of BetrFS show performances comparable to the other file systems in the recursive delete benchmark.

Table 6.2b shows the results of running the benchmark on aged file systems. In the aged setting, ext4, Btrfs, XFS and ZFS show worse performance in all benchmarks, because the file systems are not able to allocate spaces that are close to each other for files under the same directory. Btrfs and ZFS show less regression in the find and recursive delete benchmarks, because in Btrfs and ZFS, metadata are kept in trees. Since the find and grep benchmarks are read-only, and the recursive delete benchmark doesn't write new data, these benchmarks don't trigger garbage collection in NILFS2. Therefore, NILFS2 doesn't have much regression in the aged setting. All versions of BetrFS have slightly worse performance in the find and grep benchmarks because the $B^{\varepsilon}$-trees are taller in the aged setting. However, BetrFS 0.3 and BetrFS 0.4 have much worse per-

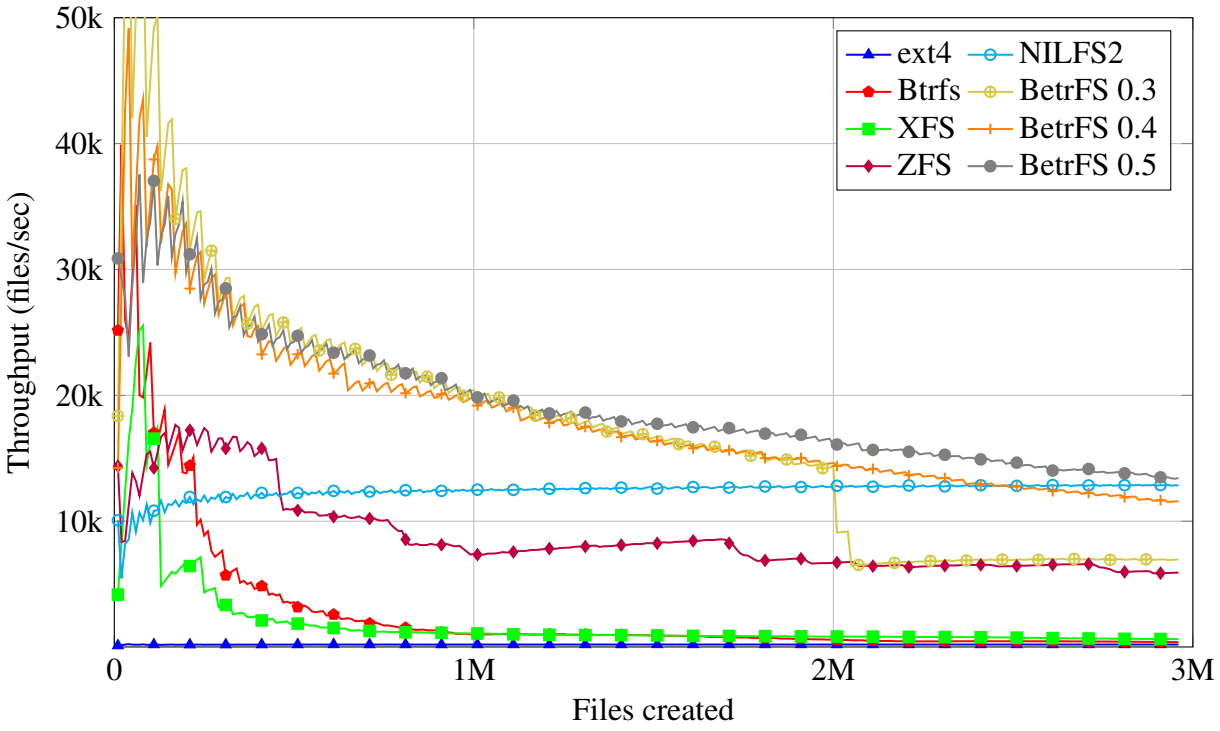



Figure 6.3: Cumulative file creation throughput during the Tokubench benchmark on empty file systems (higher is better).

formance in the recursive delete benchmark, because, during the benchmark, more range-delete messages are flushed to the leaf nodes, resulting in more node merges.

**TokuBench.** TokuBench [13] is a small-write-intensive benchmark that creates 3 million 200-Byte files in a balanced tree directory. The benchmark first creates the balanced tree directory with a fanout of 128, i.e., each directory has at most 128 directories or 128 files. Then, the benchmark creates 4 threads. Each thread iterates over the leaf directories, creating one file at a time. The benchmark reports the cumulative throughput of the file creation each time when 10,000 files are created.

Figure 6.3 shows the results of running Tokubench on empty file systems. Because Tokubench exercises small, random writes, BetrFS 0.4 and BetrFS 0.5 are significantly faster than ext4, Btrfs and XFS. Also, BetrFS 0.4 and BetrFS 0.5 don't have the sudden performance drop that occurs in BetrFS 0.3 (Section 3.3). The performance of BetrFS 0.5 is better than that of BetrFS 0.4 because

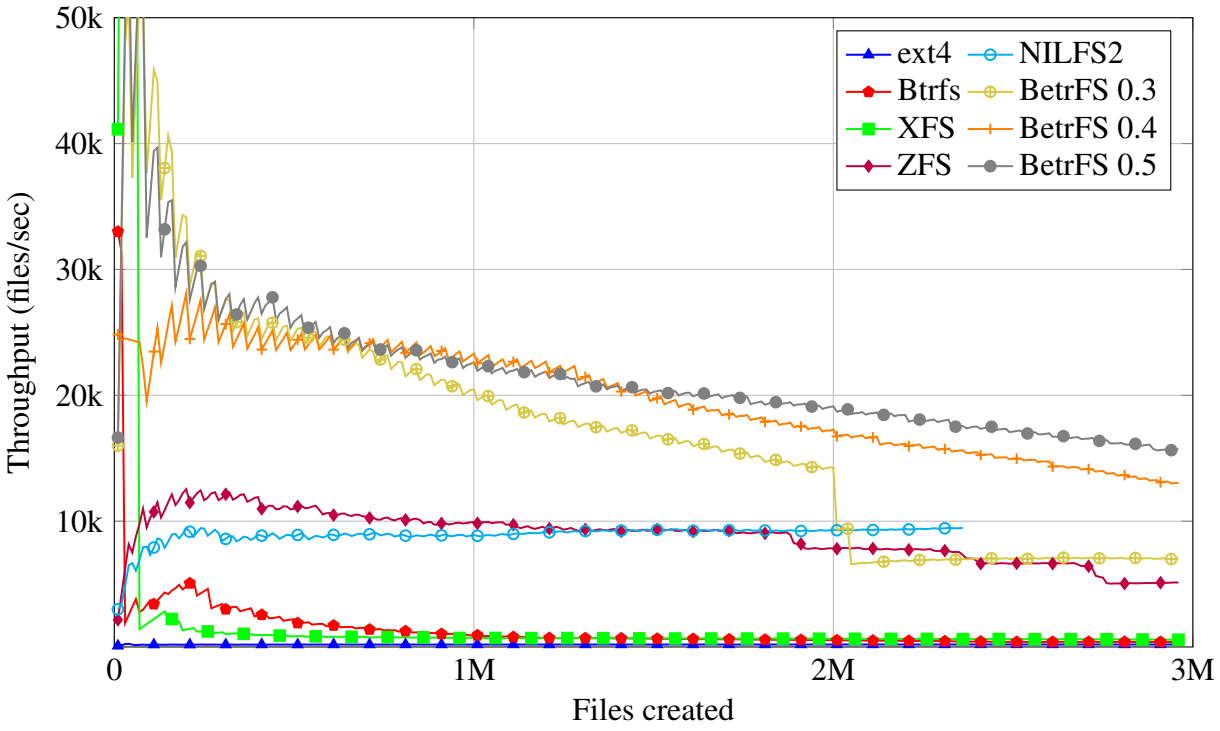



Figure 6.4: Cumulative file creation throughput during the Tokubench benchmark on aged file systems (higher is better).

preferential splitting avoids further re-lifting in the benchmark. NILFS2 translates all operations in the benchmark to log appends, therefore, the performance of NILFS2 is high and stable.

Figure 6.4 shows the results of running Tokubench on aged file systems. Because of the cost introduced by garbage collection, NILFS2 has much worse performance in the aged setting. Moreover, NILFS2 returns `ENOSPC` before completing the benchmark, despite that the total amount of data in the file system is far from filling up the disk. The performances of the other file systems are similar to those on empty file systems.

### 6.1.2 Namespace operations

**File renames.** The file rename benchmark measures the throughput of renaming files with different sizes. The benchmark first creates a file filled with random data. Then, the benchmark renames the file 100 times, each followed by an `fsync` of the parent directory, and reports the throughput, that is, renames per second.

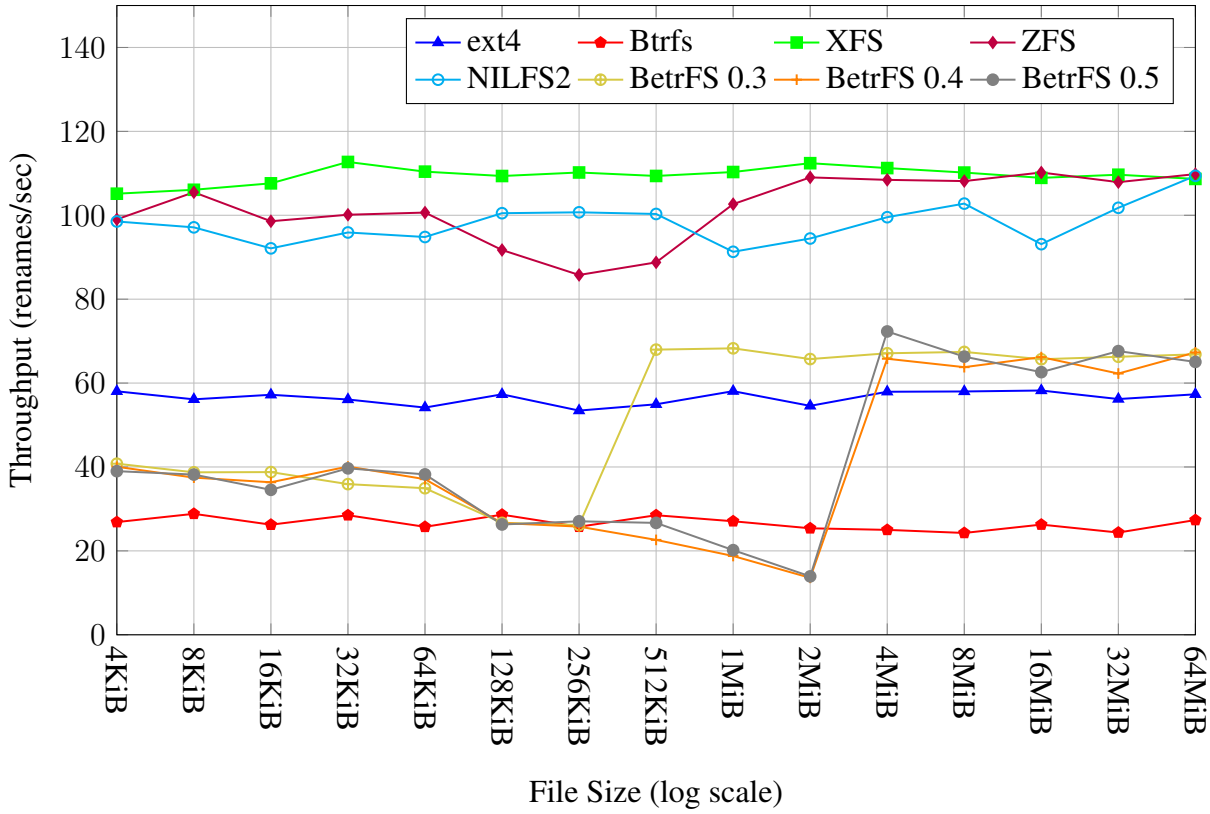



Figure 6.5: Throughput of renaming a file of different sizes on empty file systems (higher is better).

Figure 6.5 shows the results of running the benchmark on empty file systems. For all file systems except BetrFS, the rename throughput is stable regardless of the file being renames because a rename is just a pointer swing in those file systems. BetrFS uses the simple rename implementation that updates all related key/value pairs when the file size is small. Therefore, the rename throughput gradually decreases when the file grows larger. However, BetrFS 0.3 moves the file into a zone when the file is larger than or equal to 512KiB, making the rename a pointer swing. To rename a file larger than or equal to 4MiB, BetrFS 0.4 and BetrFS 0.5 use range-rename and range-clone, respectively. As shown in the figure, the throughputs of all versions of BetrFS are comparable to the other file systems after reaching the thresholds.

Figure 6.6 shows the results of running the benchmark on aged file systems. The performances of ext4, XFS, ZFS and NILFS2 are similar to those on empty file systems. Btrfs becomes

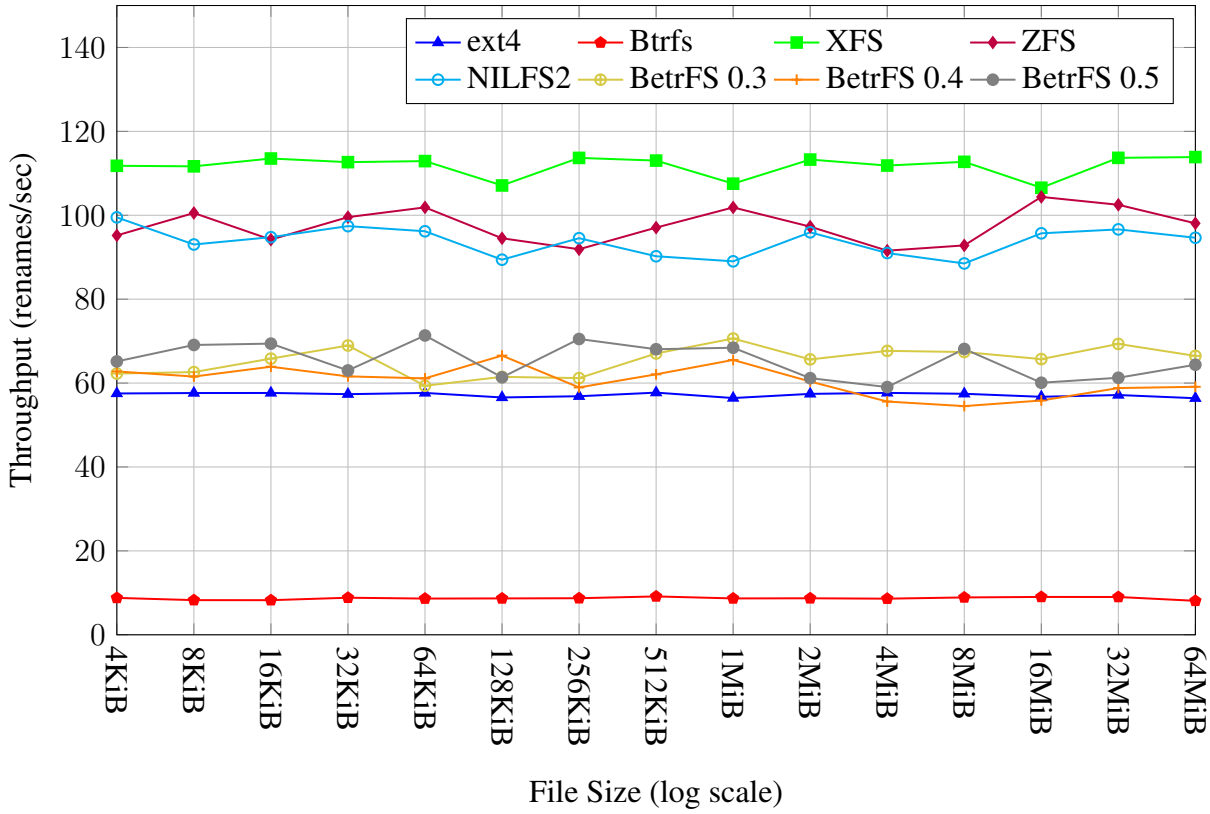



Figure 6.6: Throughput of renaming a file of different sizes on aged file systems (higher is better).

much slower. All versions of BetrFS have higher rename throughput for small files, because the rename doesn't write key/value pairs directly to leaf nodes, resulting in node splits and merges.

**Directory renames.** The directory rename benchmark measures the throughput of renaming a directory. On empty file systems, the benchmark first clones the Linux source repository to the file system and measures the throughput by renaming the directory that contains the repository 100 times, each followed by an `fsync` of the parent directory. When measuring the performance on aged file systems, the benchmark renames the directory is aged by the aging script 100 times, each followed by an `fsync` of the parent directory.

Figure 6.7 shows the results of running the benchmark on empty file systems. The throughputs of different versions of BetrFS are comparable to other file system.

Figure 6.8 shows the results of running the benchmark on aged file systems. The performances of ext4, Btrfs, XFS, and BetrFS 0.3 are similar to those on empty file systems. BetrFS

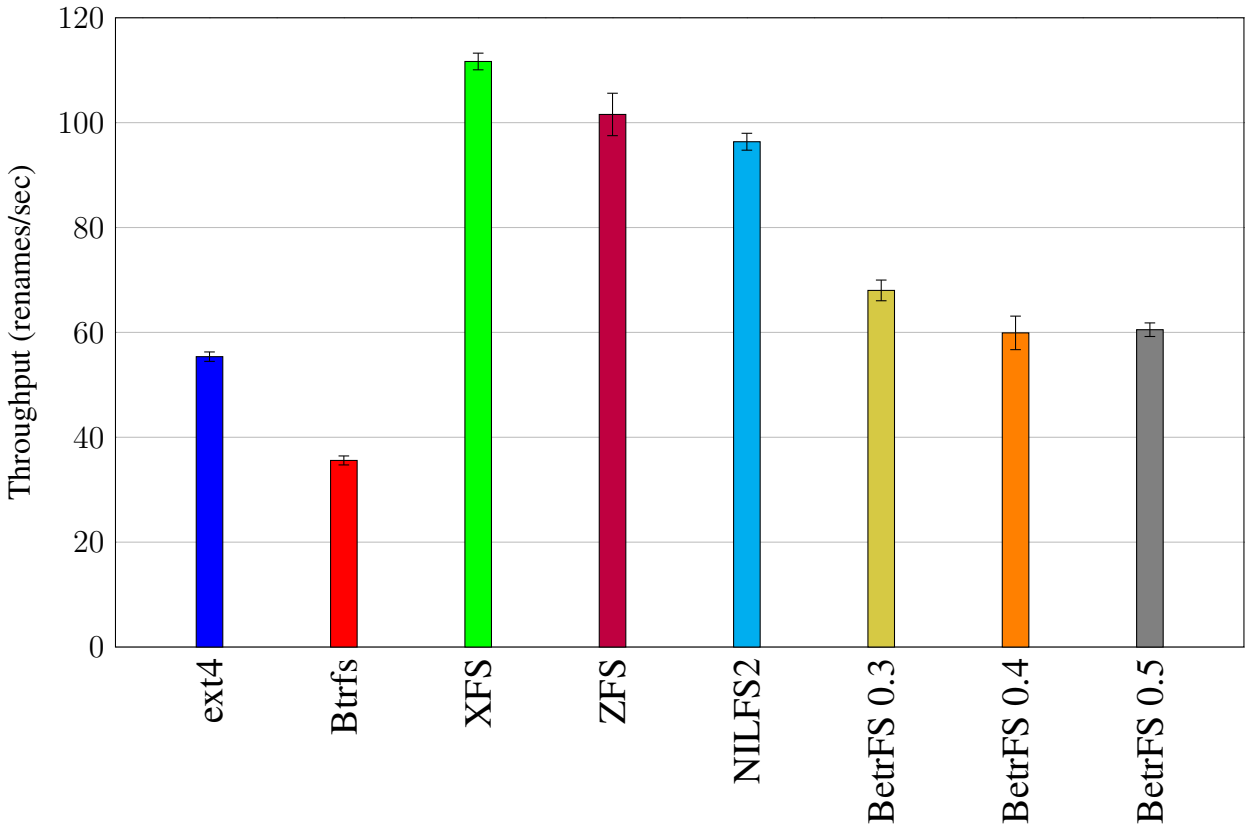



Figure 6.7: Throughput of renaming a directory on empty file systems (higher is better).

0.5 and BetrFS 0.4 become slower become slicing in taller trees. The performance of NILFS2 degrades and becomes unstable because of garbage collection. The performance of ZFS also degrades.

**Directory clones.** To evaluate the performance of cloning and similar copy-on-write optimizations in the other file systems, we wrote a simple microbenchmark that creates a directory hierarchy with 8 directories, each of which has 8 files that are 4 MiB each. At each step, we measure the copy time (Figure 6.9a), we measure the impact of copy-on-write by writing 16 bytes to each copied file and syncing the data (Figure 6.9d). We then clear the file system caches and measure the impact on read time by grepping the copied directory (Figure 6.9c). Finally, we record the change in space utilization for the file system at each step (Figure 6.9b).

We compare directory-level clone in BetrFS to 3 Linux file systems that either support volume snapshots or clones (Btrfs and ZFS) or reflink copies of files (Btrfs and XFS). We com-

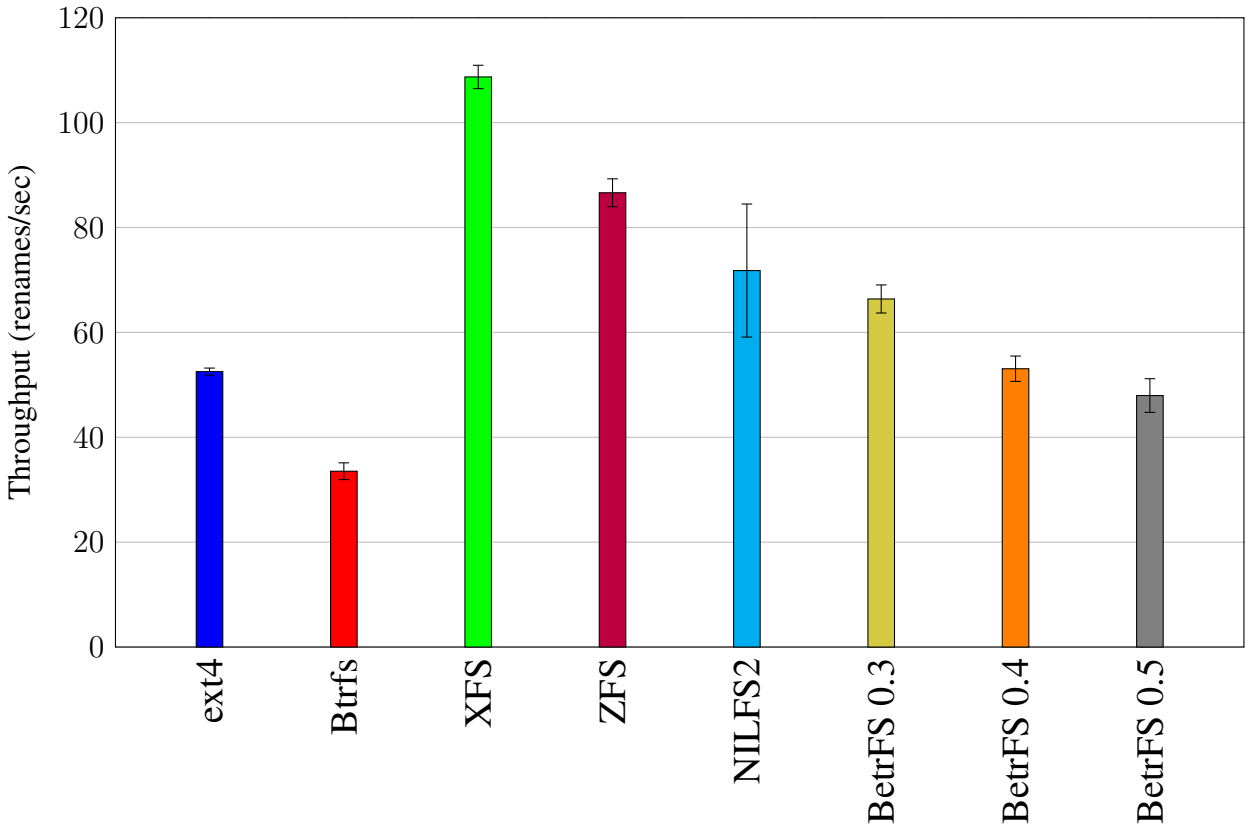
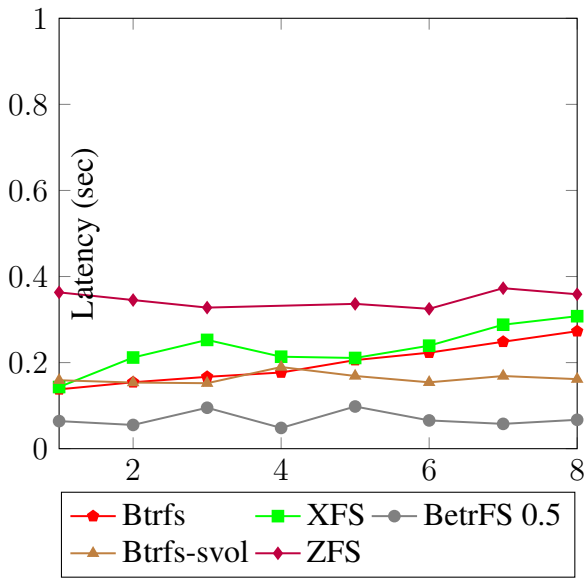


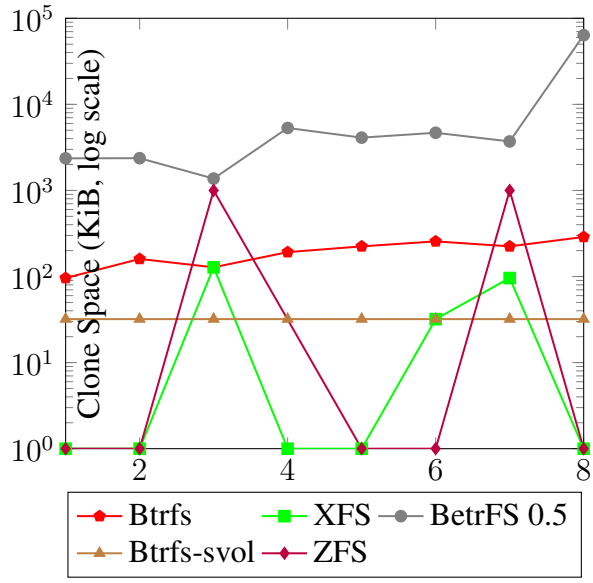

Figure 6.8: Throughput of renaming a directory on aged file systems (higher is better).

pare in both modes; the label Btrfs-svol is in volume snapshot mode. Neither model supports directory-level clone natively, but Btrfs does allow a user to turn a directory into a subvolume, which can then be snapshotted. We cover this behavior by measuring the volume case. we compare to both methods, either creating a volume with these contents and cloning it, or doing a reflink copy of each file.

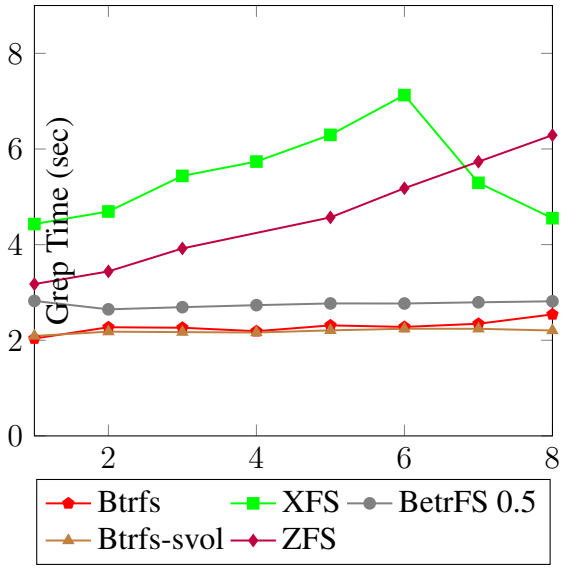
In terms of time to do a clone, both Btrfs and XFS file-level cloning degrade as a function of the number of prior clones; after 8 iterations, the latency to do the clones is roughly doubled. In contrast, the time to clone a volume in Btrfs and ZFS is relatively flat. The cloning time in BetrFS 0.5 varies somewhat, but oscillates around 60 ms, or 33% faster than the closest data point from another file system (the first clone on XFS), 58% faster than a volume clone on Btrfs, and an order of magnitude faster than the worst case for the competition.



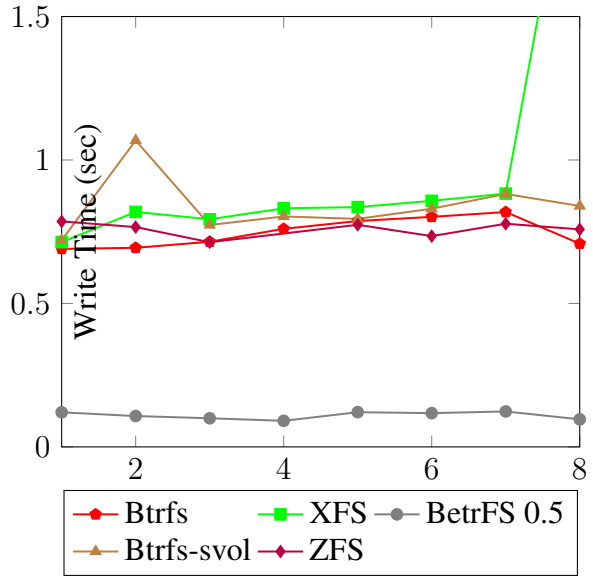

(a) Time to clone a directory.




(b) Change in space usage.




(c) Grep Time.




(d) Small write latency.

Figure 6.9: Latency to clone, read, and write, as well as space usage, as a function of the number of times a directory tree has been cloned (lower is better for all metrics).

In terms of write costs, the cost to write a cloned file or volume is flat, although BetrFS 0.5 can ingest writes 8–10× faster. In our experiments, XFS hits an occasional stutter.

Other than cloning time, the other primary degradation comes in read time. XFS and ZFS degrade severely—after 6 clones the grep time is nearly doubled. For XFS, there appears to be some work that temporarily improves locality; the degradation trend resumes after more iterations. In comparison, read time for BetrFS 0.5 is roughly constant—40% slower than the initial speed of Btrfs, and almost 2× as fast as XFS.

Finally, Figure 6.9b shows the change in space usage after each clone. For every file system except ZFS and BetrFS 0.5, the change is small an consistent. ZFS appears to allocate space for new clones in larger bursts, every few clones. The benchmark triggers a worst case space usage in BetrFS 0.5. A clone shares the LCA in the lifted $B^\varepsilon$-DAG, then, the next close flushes messages (small writes between clones) to the LCA, which breaks the sharing by cloning the LCA (a node can be as large as 4MiB in our implementation).

In total, these results indicate that the state of the art in logical copy optimization keeps write costs and space usage relatively flat, at the cost of degrading subsequent file copies and subsequent reads. We note that degrading subsequent reads is a persistent tax on future applications. BetrFS 0.5 strikes a new point: flat clone, read, and write times, most of which are much faster than the state of the art.

## 6.2 Macrobenchmarks

We then measure the performance of file systems on canonical applications.

**Git.** The git benchmark first measures the latency of finishing "git clone" that clones a local copy of the linux source repository, then measures the latency of finishing "git diff" that generates a patch between two tags in the repository, the "v4.7" and "v4.14" tags. The "git clone" process first copies files in the ".git" directory, including a large object file (2.52 GiB in our case). Then, the "git clone" process creates files in the repository by copying the content from the object file.

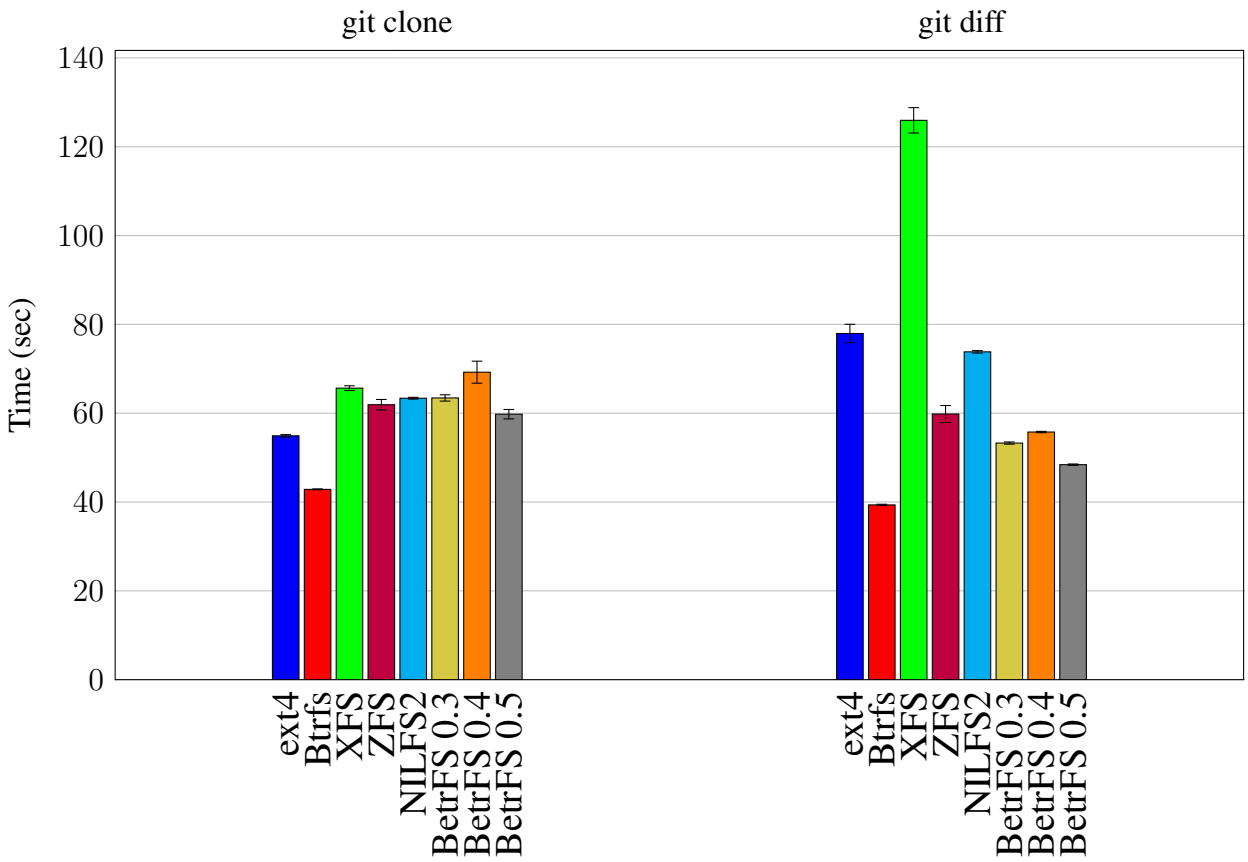



Figure 6.10: Latency of "git clone" and "git diff" on empty file systems (lower is better).

The "git diff" process opens all differing files and computes the difference by reading the object file.

Figure 6.10 shows the results of running the benchmark on empty file systems. For the "git clone" benchmark, because of the slower sequential writes, BetrFS 0.4 is slower than BetrFS 0.3 and BetrFS 0.5, which are slower than ext4 and Btrfs. The "git diff" benchmark opens differing files in depth-first-search order, therefore, though not all files are read, all versions of BetrFS are faster than the other file systems, except Btrfs.

Figure 6.11 shows the results of running the benchmark on aged file systems. All versions of BetrFS become slightly slower, because the $B^{\varepsilon}$-trees are taller. However, BetrFS 0.5 is faster than any other file systems, because, in the aged setting, the other file systems cannot allocate spaces that are close to each for files in the same directory.

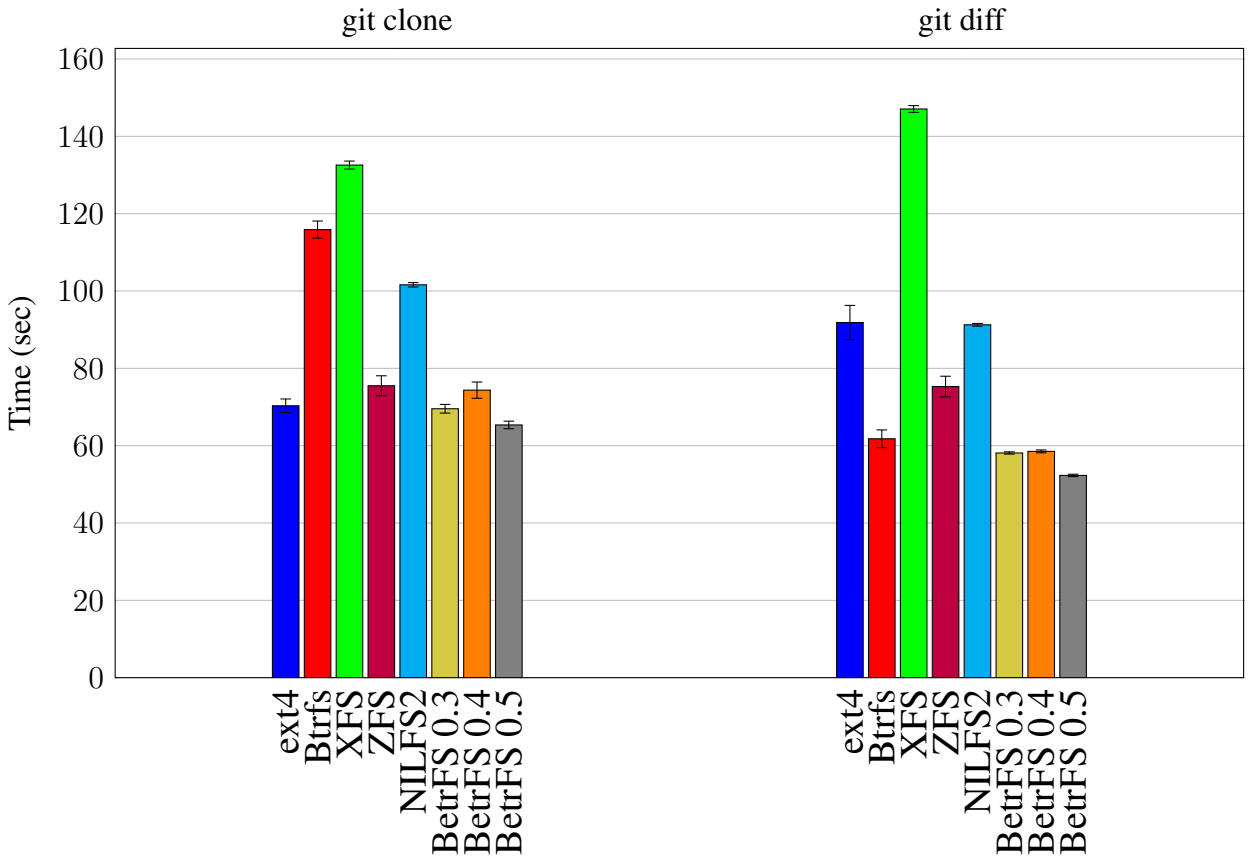



Figure 6.11: Latency of "git clone" and "git diff" on aged file systems (lower is better).

**Tar.** The tar benchmark measures the latency of untar and tar. The benchmark first copies a Linux-3.11.10 source tar ball to the file systems. Then, it measures the time to untar the tar ball, which writes a Linux-3.11.10 source directory. Next, it measures the time to generate a tar ball from the newly created Linux-3.11.10 source directory.

The untar process creates a child process that reads and uncompresses the tar ball. At the same time, the parent process creates files with the information sent from the child process. The tar process traverses the directory and sends the compressed content to a child process, which dumps the compressed content into a tar ball.

Figure 6.12 shows the results of running the benchmark on empty file systems. Because all versions of BetrFS have slower sequential reads, which are further decelerated by the messages injected by file creates, the untar benchmark is slower on versions of BetrFS than ext4 and Btrfs.

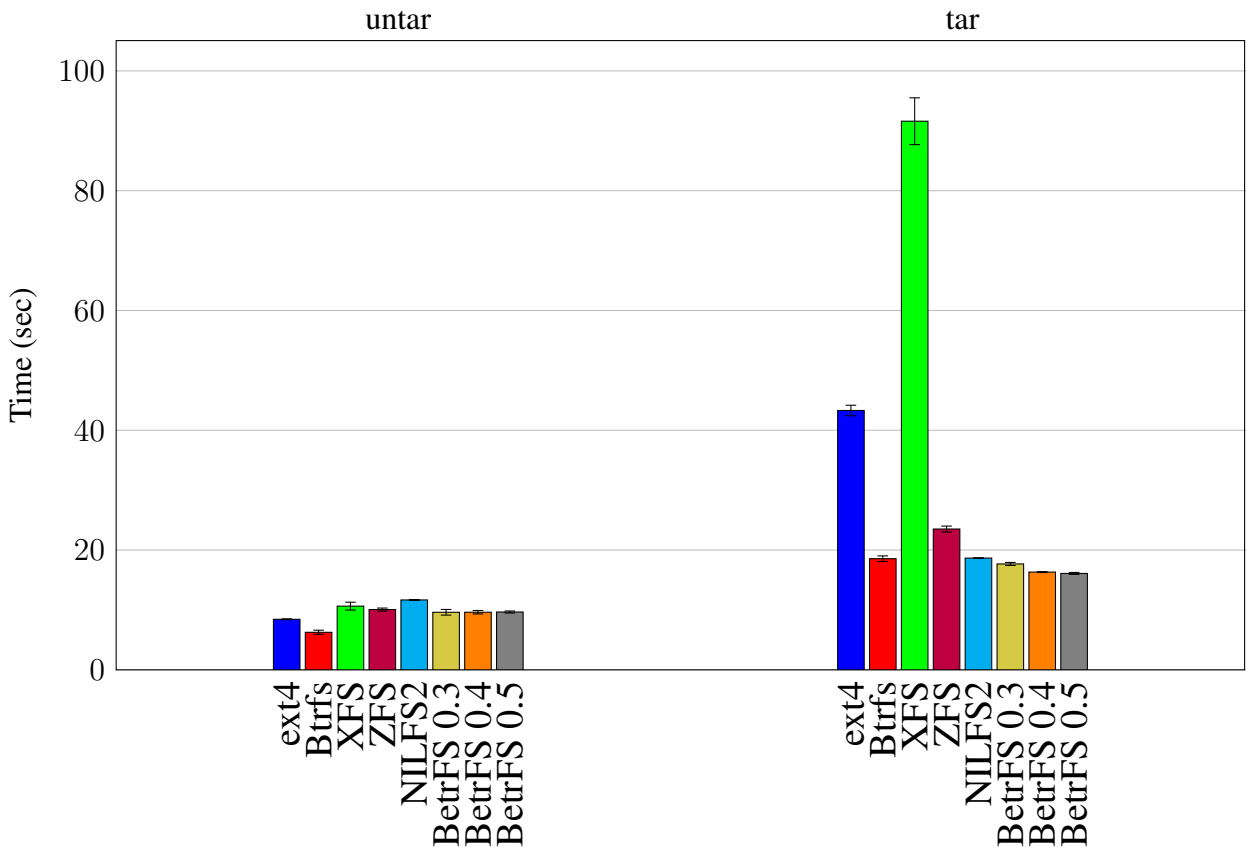



Figure 6.12: Latency to untar and tar the Liinux-3.11.10 source directory on empty file systems (lower is better).

The tar benchmark runs faster on versions of BetrFS because of the locality from full-path or relative-path indexing.

Figure 6.13 shows the results of running the benchmark on aged file systems. All versions of BetrFS have similar performances to those on empty file systems for the untar benchmark, while the tar benchmark runs slightly slower because of the taller $B^{\varepsilon}$-trees. However, versions of BetrFS are faster than any other file systems on both benchmarks in the aged setting.

**Rsync.** The rsync benchmark first copies the Linux-3.11.10 source directory to the file system, then uses rsync to create a copy of the directory. The benchmark runs twice on each file system, one with the "in-place" flags and the other without the flag. The rsync process divides the total amount of data being copies by the time elapsed and reports the throughput. However, the origi-

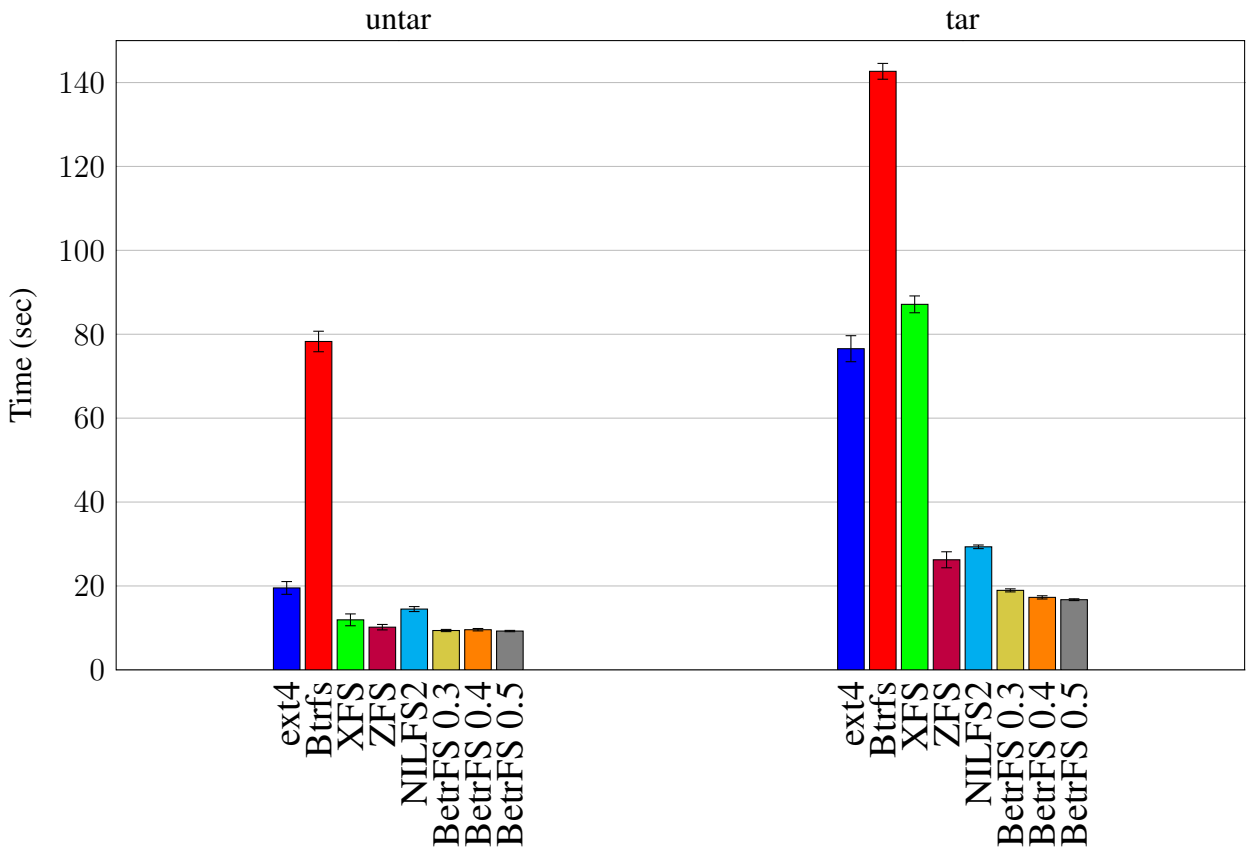



Figure 6.13: Latency to untar and tar the Liinux-3.11.10 source directory on aged file systems (lower is better).

nal rsync process measures time in seconds, resulting in a coarse granularity. In order to get more accurate results, we modify the rsync process to measure time in microseconds.

The rsync process traverses the source directory and sends the content to a child process that creates files in the destination directory. If the "in-place" is not set, for each file in a directory, the child process will first create a temporary file in the directory and then rename the temporary file to the target file. If the "in-place" is set, the child process will create files in place.

Figure 6.14 shows the results of running the benchmark on empty file systems. Because the benchmark traverses the source directory, versions of BetrFS are much faster than the other file systems. BetrFS 0.4 and BetrFS 0.5 are faster than BetrFS 0.3 because full-path indexing has better locality than relative-path indexing.

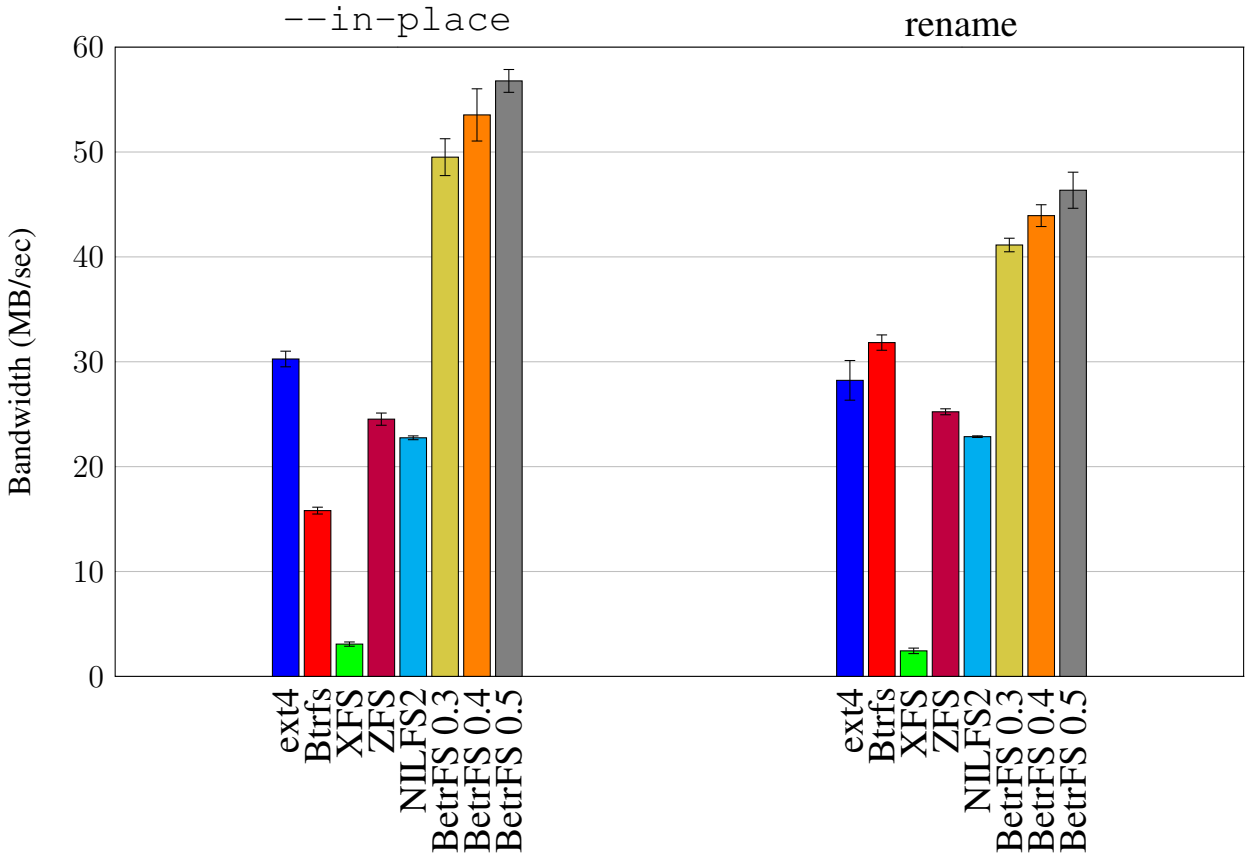



Figure 6.14: Throughput of cloning the Linux-3.11.10 source directory with rsync on empty file systems (higher is better).

Figure 6.15 shows the results of running the benchmark on aged file systems. In the aged setting, ext4, Btrfs and NILFS2 cannot make the same allocation, resulting in orse performances. Versions of BetrFS are only slightly slower because of the taller $B^{\varepsilon}$-trees.

**Mailserver.** The mailserver benchmarks measures the throughput of the dovecot mailserver on different file systems. The mailserver is configured with the Maildir option, therefore, each mail is a file. Initially, the mailserver has 10 mail boxes, each with 1000 mails. Then, the benchmark creates four threads that interact with the mailserver and measure the throughput. Each threads performs 50% reads and 50%write. Writes are randomly chosen from creating a new mail, changing the flag of an existing mail and deleting an existing mail with equal probabilities. For a read operation, the mailserver reads the index file and the file that stores the content of the mail. To create a new mail, the mailrserver first creates a file in the "tmp" directory to store the content.

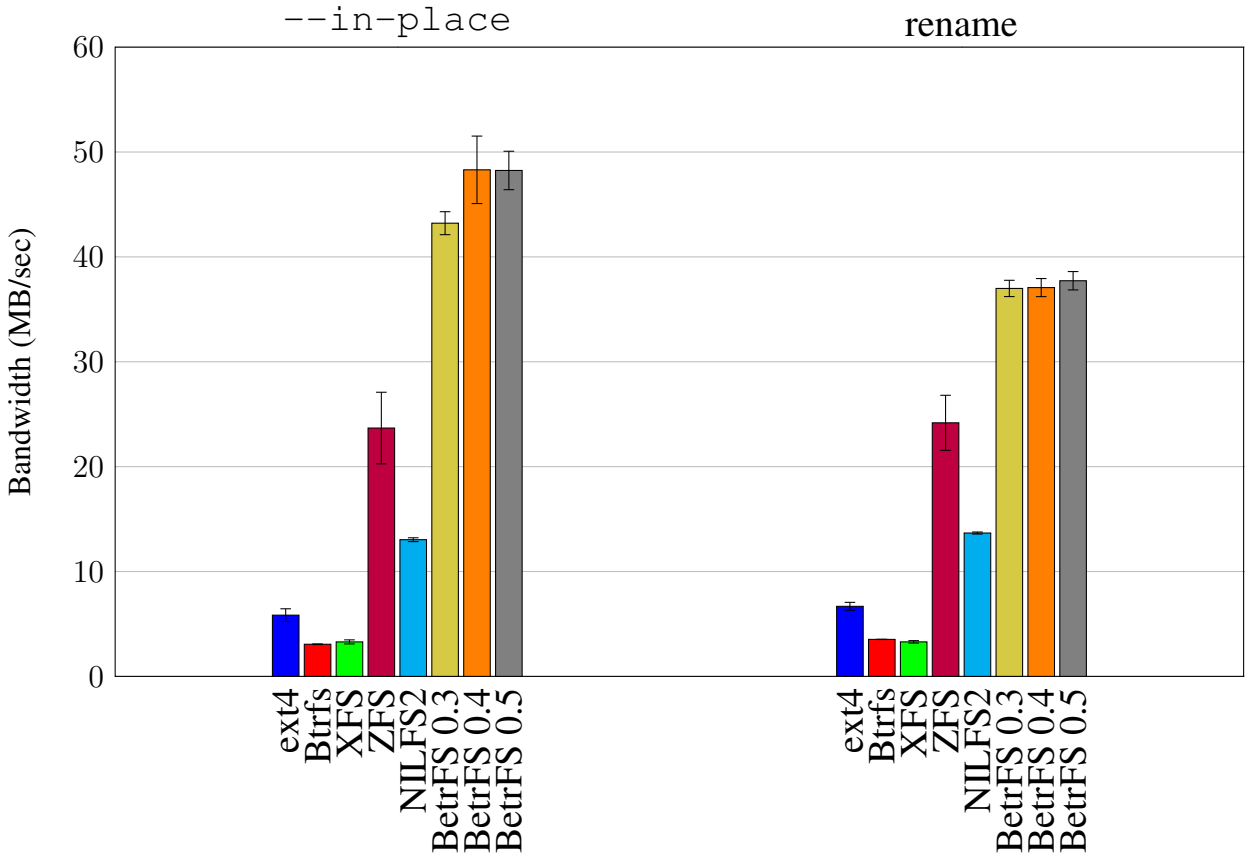



Figure 6.15: Throughput of cloning the Linux-3.11.10 source directory with rsync on aged file systems (higher is better).

Subsequently, the file is moved into the "new" directory and then into the "cur" directory. Because the mailserver stores flags in the file name, to change the flag of a mail, the mailserver renames the file that represents the mail. To delete a mail, the mailserver first renames the file so that the file name contains a "deleted" flag. Then, the mailserver deletes the file. In addition to modifying the file that represents the mail, all write operations create a temporary index file and renames the temporary file to overwrite the old index file.

Figure 6.16 shows the results of running the benchmark on empty file systems. Versions of BetrFS are faster than the other file systems except NILFS2, because versions of BetrFS have faster file creates.

Figure 6.17 shows the results of running the benchmark on aged file systems. NILFS2 becomes much slower because of garbage collection. Since the $B^{\varepsilon}$-trees are taller in the aged set-

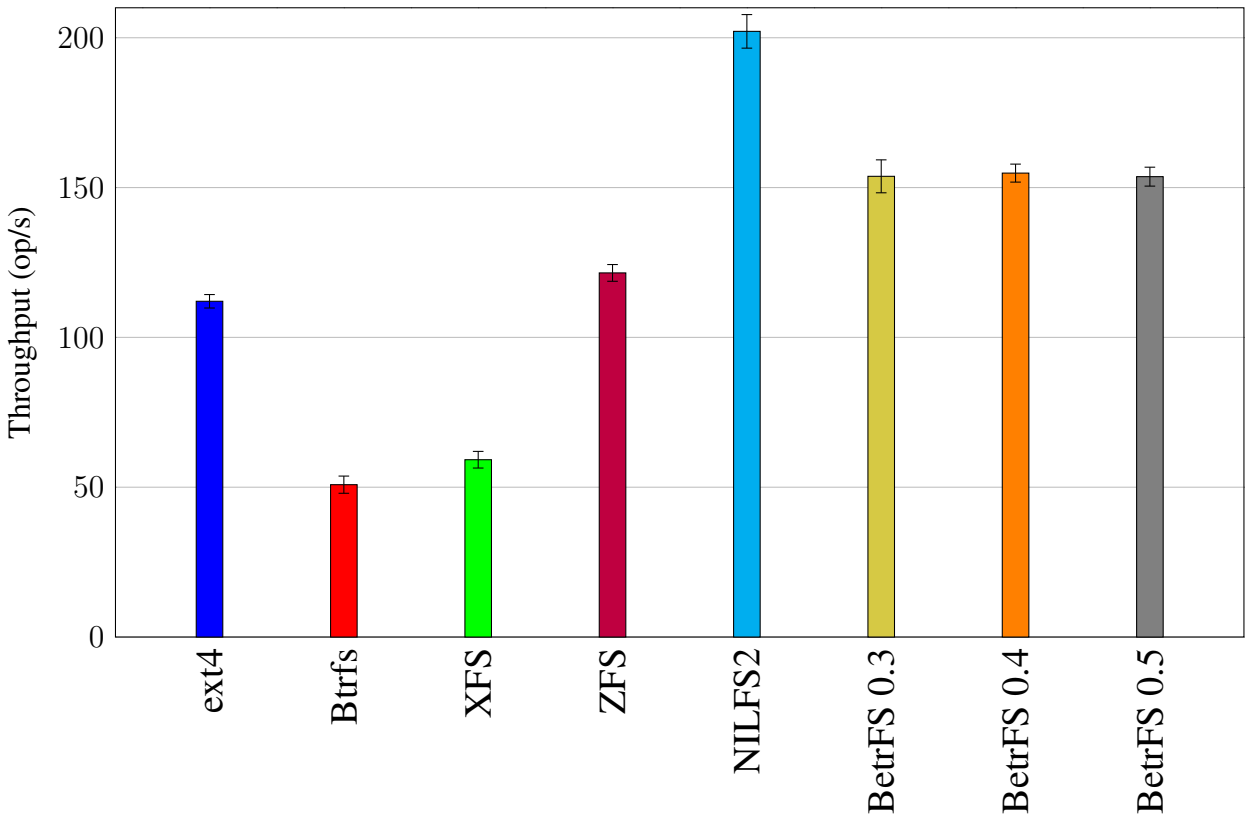



Figure 6.16: Throughput of the dovecot mailserver on empty file systems (higher is better).

ting, versions of BetrFS become slower. However, versions of BetrFS are faster than the other file systems, because the other file systems cannot make the same allocation for files in the aged setting.

## 6.3 Conclusion

BetrFS 0.4 and BetrFS 0.5 show good performance on workloads that have intensive directory scans or small random writes. The namespace operations on BetrFS 0.4 and BetrFS 0.5 have comparable performance to those on the other file systems. And in the aged setting, versions of BetrFS show much less regressions than the other file systems.

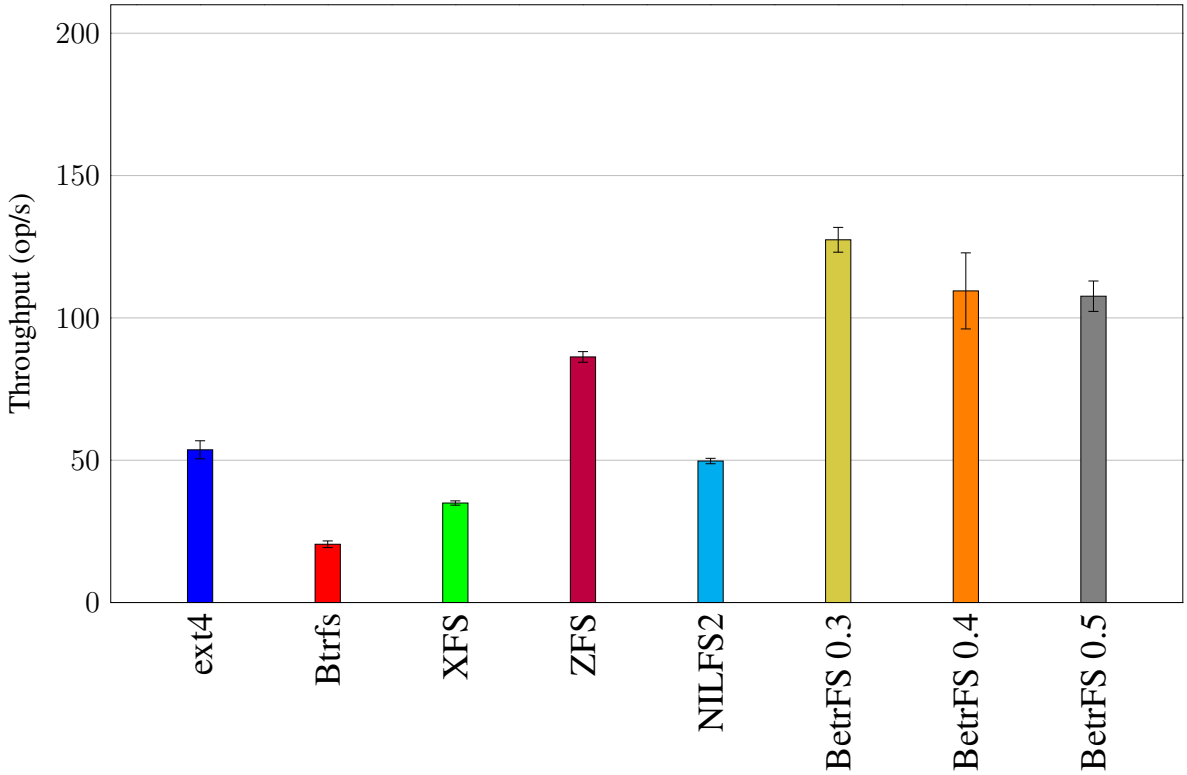



Figure 6.17: Throughput of the dovecot mailserver on aged file systems (higher is better).

# CHAPTER 7: CONCLUSION

File systems faces trade-offs between the performance of namespace operations and spatial locality. On one hand, traditional inode-based file systems have good performance for namespace operations because of the indirection between a directory and entries in the directory. However, this indirection stops these file systems from grouping metadata and data under one directory close to each other on disk, especially when the file system ages. On the other hand, full-path-indexed file systems ensure locality by indexing metadata and data by full-paths. However, existing full-path-indexed file systems either have unbounded I/O costs for namespace operations, or taxes other operations for efficient namespace operations.

This dissertation first presents a new operation on $B^\varepsilon$-trees, range-rename, that updates all keys with one prefix to have another prefix. The range-rename operation adopts **key lifting** to transform $B^\varepsilon$-trees into lifted $B^\varepsilon$-trees and accomplishes its task in a bounded number of I/Os through **tree surgery**. By invoking range-rename operations for file system renames, full-path-indexed BetrFS has bounded I/O costs for its renames.

The techniques in the range-rename operation can be applied to other full-path-indexed file systems, as long as the underlying data structures are tree-based. For example, Wang et al. [45] used a similar approach to build a full-path-indexed file system with B+-trees. Their results showed the generality of the range-rename techniques. Full-path indexing is no longer infeasible for general-purpose file systems.

Then, this dissertation introduces another new operation on $B^\varepsilon$-trees, range-clone, that clones all keys with one prefix to have another prefix. By transforming lifted $B^\varepsilon$-trees into lifted $B^\varepsilon$-DAGs, range-clone can utilize the techniques introduced by range-rename to complete its work on the critical path. Moreover, the range-clone operation can delay the tree surgery work with a

new type of messages, GOTO messages, fitting itself into the write-optimized framework of $B^{\varepsilon}$-DAGs. Full-path-indexed BetrFS thus has write-optimized file or directory renames and clones by implementing them with range-clone operations.

Similar to the range-rename operation, the range-clone operation on the critical path can be applied to other full-path-indexed file system with tree-based data structures. However, to write-optimize the range-clone operation with GOTO messages, the full-path-index file system must be built on WODs.

This dissertation shows that with the right optimizations, a full-path-indexed, write-optimized file system can have both efficient namespace operations and locality. There is no trade-off between them. Also, full-path indexing opens up new opportunities for namespace operations, such as directory clones.

More generally, this dissertation introduces techniques that update or clone a contiguous range of keys. File systems are just one example of hierarchical data. The same techniques can be applied to other hierarchical data, such as XML. Moving a subtree, which is similar to file system renames, can be normal in such hierarchical data [15]. Instead of using a secondary index, as people generally do [15], one can encode the full-paths in the keys and use a similar approach as ours.

## REFERENCES


[1] Apache. Hbase. `http://hbase.apache.org`.

[2] Michael A Bender, Martin Farach-Colton, Jeremy T Fineman, Yonatan R Fogel, Bradley C Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 81–92, 2007.

[3] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to $B^\varepsilon$-trees and write-optimization. *:login; Magazine*, 40(5):22–28, Oct 2015.

[4] Jeff Bonwick and Bill Moore. Zfs: The last word in file systems. 2007.

[5] Gerth Stølting Brodal, Erik D Demaine, Jeremy T Fineman, John Iacono, Stefan Langerman, and J Ian Munro. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1448–1456, 2010.

[6] Gerth Stolting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 546–554, 2003.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4, 2008.

[8] Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986.

[9] Sailesh Chutani, Owen T Anderson, Michael L Kazar, Bruce W Leverett, W Anthony Mason, Robert N Sidebotham, et al. The episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, 1992.

[10] 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. File systems fated for senescence? nonsense, says science! In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, pages 45–58, 2017.

[11] Chris Dragga and Douglas J. Santry. Gctrees: Garbage collecting snapshots. *ACM Transactions on Storage*, 12(1):4:1–4:32, 2016.

[12] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. Flexvol: Flexible, efficient file volume virtualization in wafl. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 129–142, 2008.

[13] John Esmet, Michael A Bender, Martin Farach-Colton, and Bradley C Kuszmaul. The tokufs streaming file system. In *Proceedings of the 4th USENIX Workshop on Hot Topics in storage and File Systems*, 2012.

[14] Facebook Inc. Rocksdb. `https://github.com/facebook/rocksdb`.

[15] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Faerber. Indexing highly dynamic hierarchical data. *Proceedings of the VLDB Endowment*, 8(10):986–997, 2015.

[16] FUSE. Fuse. `https://github.com/libfuse/libfuse`.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[18] Google Inc. Leveldb. `https://github.com/google/leveldb`.

[19] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 19–19, 1994.

[20] 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. Betrfs: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 301–315, 2015.

[21] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: Write-optimization in a kernel file system. *ACM Transactions on Storage*, 11(4):18:1–18:29, 2015.

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

[23] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

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

[25] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage*, 13(1):5, 2017.

[26] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. A fast file system for unix. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[27] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 1–17, 1999.

[28] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 115–128, 2004.

[29] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[30] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 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*, pages 537–550, 2016.

[31] Christopher Peery, Francisco Matias Cuenca-Acuna, Richard P Martin, and Thu D Nguyen. Wayfinder: Navigating and sharing information in a decentralized world. In *International Workshop on Databases, Information Systems, and Peer-to-Peer Computing*, pages 200–214, 2004.

[32] Zachary Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[33] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514, 2017.

[34] Kai Ren and Garth Gibson. Tablefs: Enhancing metadata efficiency in the local file system. In *Proceddings of the 2013 USENIX Annual Technical Conference*, pages 145–156, 2013.

[35] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage*, 3(4):2:1–2:27, 2008.

[36] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage*, 9(3):9:1–9:32, 2013.

[37] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[38] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, 1999.

[39] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. An implementation of a log-structured file system for unix. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 307–326, 1993.

[40] Pradeep J Shetty, Richard P Spillane, Ravikant R Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, pages 17–30, 2013.

[41] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, 2003.

[42] Alexander Thomson and Daniel J. Abadi. 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, pages 1–14, 2015.

[43] Tokutek Inc. ft-index. `https://github.com/Tokutek/ft-index`.

[44] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald E. Porter. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 441–456, 2015.

[45] Longhua Wang, Youyou Lu, Siyang Li, Fan Yang, and Jiwu Shu. Reducing rename overhead in full-path-indexed file system. In *Advanced Parallel Processing Technologies*, pages 43–54, 2019.

[46] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems*, page 16, 2014.

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

[48] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 4, 2004.

[49] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496, 2017.

[50] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Optimizing every operation in a write-optimized file system. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, pages 1–14, 2016.

[51] 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. Writes wrought right, and

other adventures in file system optimization. *ACM Transactions on Storage*, 13(1):3:1–3:26, 2017.

[52] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The full path to full-path indexing. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, pages 123–138, 2018.

[53] Yang Zhan, Yizheng Jiao, Donald E. Porter, Alex Conway, Eric Knorr, Martin Farach-Colton, Michael A. Bender, Jun Yuan, William Jannen, and Rob Johnson. Efficient directory mutations in a full-path-indexed file system. *ACM Transactions on Storage*, 14(3):22:1–22:27, 2018.