Writes Wrought Right, and Other Adventures in File System Optimization

JUN YUAN, Farmingdale State College‡
YANG ZHAN, University of North Carolina at Chapel Hill‡
WILLIAM JANNEN and PRASHANT PANDEY, Stony Brook University
AMOGH AKSHINTALA, University of North Carolina at Chapel Hill‡
KANCHAN CHANDNANI, Apple Inc‡
POOJA DEO, Arista Networks‡
ZARDOSHT KASHEFF, Facebook
LEIF WALSH, Two Sigma†
MICHAEL A. BENDER, Stony Brook University
MARTIN FARACH-COLTON, Rutgers University
ROB JOHNSON, Stony Brook University
BRADLEY C. KUSZMAUL, Oracle§
DONALD E. PORTER, University of North Carolina at Chapel Hill‡

File systems that employ write-optimized dictionaries (WODs) can perform random-writes, metadata updates, and recursive directory traversals orders of magnitude faster than conventional file systems. However, previous WOD-based file systems have not obtained all of these performance gains without sacrificing performance on other operations, such as file deletion, file or directory renaming, or sequential writes.

Using three techniques, *late-binding journaling*, *zoning*, and *range deletion*, we show that there is no fundamental trade-off in write-optimization. These dramatic improvements can be retained while matching conventional file systems on *all* other operations.

BetrFS 0.2 delivers order-of-magnitude better performance than conventional file systems on directory scans and small random writes and matches the performance of conventional file systems on rename, delete, and sequential I/O. For example, BetrFS 0.2 performs directory scans 2.2× faster, and small random writes

This research was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, CNS-1149229, CNS-1161541, CNS-1228839, IIS-1247750, CCF-1314547, CNS-1526707, and VMware. †Work done while at Tokutek. ‡Work done while at Stony Brook University. §Work done while at Massachusetts Institute of Technology.

Authors' addresses: J. Yuan, Computer Systems Department, Farmingdale State College, NY 11735; email: yuanj@farmingdale.edu; Computer Science Department, Stony Brook University, NY 11794-2424; email: junyuan@cs.stonybrook.edu; Y. Zhan, Department of Computer Science, Sitterson Hall, 201 S Columbia St, Chapel Hill, NC 27516; email: yzhan@cs.unc.edu; W. Jannen, Williams College; email: jannen@cs.williams.edu; A. Akshintala, Department of Computer Science, Sitterson Hall, 201 S Columbia St, Chapel Hill, NC 27516; email: aakshintala@cs.unc.edu; K. Chandnani and P. Deo, 355 North Wolfe Road, Apt 521, Sunnyvale, CA 9408; emails: ckanchan92@gmail.com, pdeo@cs.stonybrook.edu; Z. Kasheff, 770 Broadway, 15th floor, New York, NY 10003; email: zardosht@gmail.com; L. Walsh, 102 Pierrepont Street, Apt 3F, Brooklyn, NY 11201; email: leif.walsh@gmail.com; M. A. Bender, Computer Science Department, Stony Brook University, Stony Brook, NY 11794-2424; email: bender@cs.stonybrook.edu; M. Farach-Colton, Department of Computer Science, Rutgers University, Piscataway, NJ 08854; email: martin@farach-colton.edu; R. Johnson, Computer Science Department, Stony Brook University, Stony Brook, NY 11794-2424; email: rob@cs.stonybrook.edu; B. C. Kuszmaul, 37 Vaille Ave Lexington, MA 02421; email: Kuszmaul@gmail.com; D. E. Porter, Department of Computer Science, University of North Carolina at Chapel Hill, Campus Box 3175, Brooks Computer Science Building, Chapel Hill, NC 27599-3175; email: porter@cs.unc.edu.

DOI: http://dx.doi.org/10.1145/3032969

over two orders of magnitude faster, than the fastest conventional file system. But unlike BetrFS 0.1, it renames and deletes files commensurate with conventional file systems and performs large sequential I/O at nearly disk bandwidth. The performance benefits of these techniques extend to applications as well. BetrFS 0.2 continues to outperform conventional file systems on many applications, such as as `rsync`, `git-diff`, and `tar`, but improves `git-clone` performance by 35% over BetrFS 0.1, yielding performance comparable to other file systems.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; D.4.3 [**Operating Systems**]: File Systems Management; D.4.8 [**Operating Systems**]: Performance; E.5 [**Files**]: Organization/ Structure; Sorting/Searching

General Terms: Design, Performance, Theory

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

ACM Reference Format:
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, Donald E. Porter. 2017. Writes wrought right, and other adventures in file system optimization. ACM Trans. Storage 13, 1, Article 3 (March 2017), 26 pages.
DOI: http://dx.doi.org/10.1145/3032969

1. INTRODUCTION

Write-Optimized Dictionaries (WODs),[1] such as Log-Structured Merge Trees (LSM-trees) [O'Neil et al. 1996] and B$^{\varepsilon}$-trees [Brodal and Fagerberg 2003], are promising building blocks for managing on-disk data in a file system. Compared to conventional file systems, previous WOD-based file systems have improved the performance of random writes [Esmet et al. 2012; Jannen et al. 2015a; Shetty et al. 2013], metadata updates [Esmet et al. 2012; Jannen et al. 2015a; Shetty et al. 2013; Ren and Gibson 2013], and recursive directory traversals [Esmet et al. 2012; Jannen et al. 2015a] by orders of magnitude.

However, previous WOD-based file systems have not obtained all three of these performance gains without sacrificing performance on some other operations. For example, TokuFS [Esmet et al. 2012] and BetrFS [Jannen et al. 2015a] have slow file deletions, renames, and sequential file writes. Directory traversals in KVFS [Shetty et al. 2013] and TableFS [Ren and Gibson 2013] are essentially no faster than conventional file systems. TableFS stores large files in the underlying ext4 file system, and hence offers no performance gain for random file writes.

These performance improvements from WODs should not come at the cost of other file system operations; in the case of a B$^{\varepsilon}$-tree, asymptotic analysis indicates that B$^{\varepsilon}$-trees should match or outperform a B-tree, or other common on-disk data structures. In other words, there should not be any downside to replacing a B-tree with a B$^{\varepsilon}$-tree in a file system.

This article shows that a WOD-based file system can retain performance improvements to outperform conventional file systems on metadata updates, small random writes, and recursive directory traversals—sometimes by orders of magnitude—while matching conventional file systems on other operations.

We identify three techniques to address fundamental performance issues for WOD-based file systems and implement them in BetrFS [Jannen et al. 2015a, 2015b]. We call the resulting system BetrFS 0.2 and the baseline BetrFS 0.1. Although we implement these ideas in BetrFS, we expect they will improve any WOD-based file system and possibly have more general applications.

[1] The terms Write-Optimized Index (WOI), Write-Optimized Dictionary (WOD), and Write-Optimized Data Structure (WODS) can be used interchangeably.

First, we use a *late-binding journal* to perform large sequential writes at disk bandwidth while maintaining the strong recovery semantics of full-data journaling. BetrFS 0.1 provides full-data journaling but halves system throughput for large writes because all data is written at least twice. Our late-binding journal adapts an indirection approach used by no-overwrite file systems, such as zfs [Bonwick and Moore 2005] and btrfs [Rodeh et al. 2013], which writes data into free space only once. A particular challenge in adapting this technique to a B^{ε}-tree is balancing the crash consistency of data against sufficient I/O scheduling flexibility to avoid reintroducing large, duplicate writes in B^{ε}-tree message flushing. B^{ε}-tree small-write performance hinges on I/O scheduling, including batched and deferred writes. TokuDB uses full-data journaling to guarantee the crash consistency without losing the benefit of batching small writes. However, full-data journaling doubles the IO of large writes.

Second, BetrFS 0.2 introduces a tunable directory tree partitioning technique, called *zoning*, that balances the tension between fast recursive directory traversals and fast file and directory renames. There is a tension between fast renames and fast directory traversals: Fast traversals require colocating related items on disk, but to maintain this locality, renames must physically move data. Fast renames can be implemented by updating a few metadata pointers, but this can scatter a directory's contents across the disk, requiring performance-killing seeks during a recursive traversal. In practical terms, most file systems use inodes, resulting in fast renames but potentially slow traversals; BetrFS sorts file system data/metadata by full-path names, enabling fast traversals at the expense of renames. Zoning yields most of the benefits of both designs. BetrFS 0.2 traverses directories at near disk bandwidth and renames at speeds comparable to inode-based systems.

Finally, BetrFS 0.2 contributes a new *range delete* WOD operation that accelerates unlinks, sequential writes, renames, and zoning. Although we originally developed the range delete operation to reduce the latency of file deletions, it has proven to be a generally useful tool; we use range delete during renames, file overwrites, and zoning. BetrFS 0.2 uses range deletes to tell the WOD when large swaths of data are no longer needed. Range deletes enable further optimizations, such as avoiding the read-and-merge of stale data, that would otherwise be difficult or impossible.

The key to WOD performance is batching writes; unless deletions from the WOD are also batched, performance is throttled by reading and merging keys that are ultimately obviated. Although random writes can be batched without a special interface, deletion is more complex, as this requires a priori knowledge of the keyspace. Range delete is powerful in that a file system can communicate more information about what it is doing in a simple way, which the WOD can then use to optimize its behavior in ways not previously possible.

With these enhancements, BetrFS 0.2 can roughly match other local file systems on Linux. In some cases, it is much faster than other file systems or provides stronger guarantees at a comparable cost. In a few cases, it is slower, but within a reasonable margin.

The contributions of this article are:

—A **late-binding journal** for large writes to a message-oriented WOD, which provides full-data journaling recovery at the cost of metadata-only journaling. This technique leverages copy-on-write techniques to avoid writing large values twice, while retaining all of the guarantees of traditional logging and all the optimizations of the WOD. BetrFS 0.2 writes large files at 96MB/s, compared to 28MB/s in BetrFS 0.1.

—A **zone-tree schema** and analytical framework for reasoning about tradeoffs between locality in directory traversals and indirection for fast file and directory renames. We identify a point that preserves most of the scan performance of the original BetrFS and supports renames competitive with conventional file systems for most file and directory sizes. The highest rename overhead is bound at 3.8× slower than the ext4.

—A **range delete** primitive, which enables WOD -internal optimizations for file deletion, and also avoids costly reads and merges of dead tree nodes. With range delete, BetrFS 0.2 can unlink a 1GB file in 11ms, compared to over a minute on BetrFS 0.1 and 110ms on ext4.

—A thorough evaluation of these optimizations and their impact on real-world applications.

Thus, BetrFS 0.2 demonstrates that a WOD can improve file system performance on random writes, metadata updates, and directory traversals by orders of magnitude without sacrificing performance on other file system operations.

2. BACKGROUND

This section gives the background necessary to understand and analyze the performance of WOD-based file systems, with a focus on B$^\varepsilon$-trees and BetrFS. See Bender et al. [2015] for a more comprehensive tutorial.

2.1. Write-Optimized Dictionaries

WODs include Log-Structured Merge Trees (LSM-trees) [O'Neil et al. 1996] and their variants [Sears and Ramakrishnan 2012; Shetty et al. 2013; Wu et al. 2015], B$^\varepsilon$-trees [Brodal and Fagerberg 2003], xDicts [Brodal et al. 2010], and cache-oblivious lookahead arrays (COLAs) [Bender et al. 2007; Santry and Voruganti 2014]. WODs provide a key-value interface supporting insert, query, delete, and range-query operations.

The WOD interface is similar to that of a B-tree, but the performance profile is different:

—WODs can perform inserts of random keys orders of magnitude faster than B-trees. On a rotating disk, a B-tree can perform only a couple hundred inserts per second in the worst case, whereas a WOD can perform many tens of thousands.

—In WODs, a delete is implemented by inserting a tombstone message, which is extremely fast.

—Some WODs, such as B$^\varepsilon$-trees, can perform point queries as fast as a B-tree. B$^\varepsilon$-trees (but not LSM-trees) offer a provably optimal combination of query and insert performance.

—WODs perform range queries at nearly disk bandwidth. Because a WOD can use nodes over a megabyte in size, a scan requires less than one disk seek per megabyte of data and hence is bandwidth bound.

The key idea behind write optimization is deferring and batching small, random writes. A B$^\varepsilon$-tree logs insertions or deletions as *messages* at the root of the tree and only flushes messages down a level in the tree when enough messages have accrued to offset the cost of accessing the child. As a result, a single message may be written to disk multiple times. Since each message is always written as part of a larger batch, the amortized cost for each insert is typically much less than one I/O. In comparison, writing a random element to a large B-tree requires a minimum of one I/O.

B$^\varepsilon$-tree queries have logarithmic I/O cost, comparable to a B-tree. A B$^\varepsilon$-tree stores keys and values at the leaves, and all messages that mutate a key-value pair are guaranteed to be in buffers on the path from the root of the tree to the appropriate leaf. Thus, a single root-to-leaf traversal brings all the relevant messages into memory, where the relevant pending messages can be applied to the old value to compute the current value before returning it as the result of the query.

Finally, B$^{\varepsilon}$-trees can realize faster range queries because leaves in the tree are typically larger than a B-tree (megabytes vs. tens to hundreds of kilobytes). In both data structures, large nodes mean fewer seeks within a given range. For a B-tree, large nodes harm insertion performance, as more data must be rewritten around an inserted item. In a B$^{\varepsilon}$-tree, however, data is moved in batches, so the cost of rewriting a large leaf can be amortized over all the messages in a batch.

In theory, WODS can perform large sequential inserts at speeds proportional to disk bandwidth. However, most production-quality WODs are engineered for use in databases, not in file systems, and are therefore designed with different performance requirements. For example, the open-source WOD implementation underlying BetrFS is a port of TokuDB[2] into the Linux kernel [Tokutek, Inc. 2013]. TokuDB logs all inserted keys and values to support transactions, limiting the write bandwidth to at most half of disk bandwidth. As a result, BetrFS 0.1 provides full-data journaling, albeit at a cost to large sequential writes.

Caching and recovery. We now summarize the relevant logging and cache management features of TokuDB necessary to understand the technical contributions of this article.

TokuDB updates B$^{\varepsilon}$-tree nodes using redirect on write [Garimella 2006]. In other words, each time a dirty node is written to disk, the node is placed at a new location. Recovery is based on periodic, stable, sharp checkpoints of the tree. Between checkpoints, a write-ahead, logical log tracks all tree updates and can be replayed against the last stable checkpoint for recovery. This log is buffered in memory and is made durable at least once every second.

It is also possible to sync the log upon each commit. Flushing the write-ahead log out is one way that the B$^{\varepsilon}$-tree guarantees the durability of the messages in the tree. The `fsync` operation of BetrFS 0.1 and BetrFS 0.2 is simply based on the log flushing. The other way to guarantee the durability of the messages, which is unprevailing in TokuDB, is to write out the dirty nodes of these messages, but the significant write amplification it might incur hurts the small-write performance.

This scheme of checkpoint and write-ahead log allows the B$^{\varepsilon}$-tree to cache dirty nodes in memory and write them back in any order, as long as a consistent version of the tree is written to disk at checkpoint time. After each checkpoint, old checkpoints, logs, and unreachable nodes are garbage collected.

Caching dirty nodes improves insertion performance because TokuDB can often avoid writing internal tree nodes to disk. When a new message is inserted into the tree, it can immediately be moved down the tree as far as possible without dirtying any new nodes. If the message is part of a long stream of sequential inserts, then the *entire* root-to-leaf path is likely to be dirty, and the message can go straight to its leaf. This caching, combined with write-ahead logging, explains why large sequential writes in BetrFS 0.1 realize at most half[3] of the disk's bandwidth: most messages are written once to the log and only once to a leaf. Section 3 describes a late-binding journal, which lets BetrFS 0.2 write large data values only once, without sacrificing the crash consistency of data.

Message propagation. As the buffer in an internal B$^{\varepsilon}$-tree node fills up, the B$^{\varepsilon}$-tree estimates which child or children would receive enough messages to amortize the cost of flushing these messages down one level. Messages are kept logically consistent within a node buffer, stored in commit order. Even if messages are physically applied to leaves

[2] TokuDB implements Fractal Tree indexes [Bender et al. 2007], a B$^{\varepsilon}$-tree variant.

[3] TokuDB had a performance bug that further reduced BetrFS 0.1's sequential write performance to at most one-third of disk bandwidth. See Section 6 for details.

at different times, any read applies all matching buffered messages between the root and leaf in commit order. The only difference is whether saving the result of applying these messages is worth the cost of a copy-on-write replacement of the leaf and some ancestor nodes (for many updates), or whether it makes more sense to reapply pending messages on read (for a few updates).

Pending messages are estimated by *pivot* keys, which delimit the range of keys stored in each child subtree and track the number of pending messages within each child range. Both the parent and child are rewritten to disk, in copy-on-write fashion. The use of COW nodes also permits several important optimizations. For instance, "hot" messages can move down the tree in memory, without frequently rewriting each level of the tree. If a large set of messages destined for a small range of leaves are added to the root of the tree in quick succession, the tree can elide writing interior nodes to the disk until this region of the tree becomes less active or the next checkpoint.

Section 5 introduces a "rangecast" message type, which can propagate to multiple children.

2.2. BetrFS

BetrFS stores all file system data—both metadata and file contents—in B$^\varepsilon$-trees [Jannen et al. 2015a]. BetrFS uses two B$^\varepsilon$-trees: a metadata index and a data index. The metadata index maps full paths to the corresponding `struct stat` information. The data index maps (path, block-number) pairs to the contents of the specified file block.

Indirection. A traditional file system uses indirection (e.g., inode numbers) to implement renames efficiently with a single pointer swap. This indirection can hurt directory traversals because, in the degenerate case, there could be one seek per file.

The BetrFS 0.1 full-path-based schema instead optimizes directory traversals at the expense of renaming large files and directories. A recursive directory traversal maps directly to a range query in the underlying B$^\varepsilon$-tree, which can run at nearly disk bandwidth. On the other hand, renames in BetrFS 0.1 must move all data from the old keys to new keys, which can become expensive for large files and directories. Section 4 presents schema changes that enable BetrFS 0.2 to perform recursive directory traversals at nearly disk bandwidth and renames at speeds comparable to inode-based file systems.

Indexing data and metadata by full path also harms deletion performance, as each block of a large file must be individually removed. The sheer volume of these delete messages in BetrFS 0.1 leads to orders-of-magnitude worse unlink times for large files. Section 5 describes our new "rangecast delete" primitive for implementing efficient file deletion in BetrFS 0.2.

Consistency. In BetrFS, file writes and metadata changes are first recorded in the kernel's generic VFS data structures. The VFS may cache dirty data and metadata for up to 5 seconds before writing it back to the underlying file system, which BetrFS converts to B$^\varepsilon$-tree operations. During a VFS write-back, BetrFS records changes in an in-memory log buffer, which is made durable at least once every second. Thus, BetrFS can lose at most 6 seconds of data during a crash—5 seconds from the VFS layer and 1 second from the B$^\varepsilon$-tree log buffer. `fsync` in BetrFS first writes all dirty data and metadata associated with the inode, then writes the entire log buffer to disk.

3. AVOIDING DUPLICATE WRITES

This section discusses *late-binding journaling*, a technique for delivering the sequential-write performance of metadata-only journaling while guaranteeing full-data journaling semantics.

BetrFS 0.1 is unable to match the sequential write performance of conventional file systems because it writes all data at least twice: once to a write-ahead log and at least once to the B$^\varepsilon$-tree. As our experiments in Section 7 show, BetrFS 0.1 on a commodity disk performs large sequential writes at 28MB/s, whereas other local file systems perform large sequential writes at 78 to 106MB/s—utilizing nearly all of the hard drive's 125MB/s of bandwidth. The extra write for logging does not significantly affect the performance of small random writes, since they are likely to be written to disk several times as they move down the B$^\varepsilon$-tree in batches. However, large sequential writes are likely to go directly to tree leaves, as explained in Section 2.1. Since they would otherwise be written only once in the B$^\varepsilon$-tree, logging halves BetrFS 0.1 sequential write bandwidth. Similar overheads are well known for update-in-place file systems, such as ext4, which defaults to metadata-only journaling as a result.

Popular no-overwrite file systems address journal write amplification with indirection. For small values, zfs embeds data directly in a log entry. For large values, it writes data to disk redirect-on-write and stores a pointer in the log [McKusick et al. 2014]. This gives zfs fast durability for small writes by flushing the log, avoids the overhead of writing large values twice, and retains the recovery semantics of data journaling. On the other hand, btrfs [Rodeh et al. 2013] uses indirection for all writes, regardless of size. It writes data to newly allocated blocks and records those writes with pointers in its journal.

In the rest of this section, we explain how we integrate indirection for large writes into the BetrFS recovery mechanism, and we discuss the challenges posed by the message-oriented design of the B$^\varepsilon$-tree.

We present the late-binding journal in the context of a simplified version of the logging and copy-on-checkpointing system used by BetrFS.

BetrFS on-disk structures. B$^\varepsilon$-tree maintains three on-disk structures:

1) A B$^\varepsilon$-tree
2) A blockmap, which contains all the information on free and allocated disk blocks, including blocks used to hold the log
3) A write-ahead logical log

After checkpoint i completes, there is a single B$^\varepsilon$-tree, T_i, and a block map, Blockmap$_i$, indicating the disk blocks used by T_i. Blockmap$_i$ also reserves some space for Log$_{i+1}$, which will log all operations between checkpoint i and $i+1$. If the system crashes at any time between the completion of checkpoint i and checkpoint $i+1$, it will resume from tree T_i and Blockmap$_i$ and replay Log$_{i+1}$.

Between checkpoints, the system maintains two in-memory blockmaps: an immutable copy of Blockmap$_i$ and an active Blockmap$_{i+1}$. New allocations are recorded in Blockmap$_{i+1}$, but Blockmap$_{i+1}$ is not written to disk until the next checkpoint. Thus, if the system crashes, all disk allocations performed since checkpoint i will automatically be freed. A block is considered allocated if it is allocated in either Blockmap$_i$ or Blockmap$_{i+1}$. Thus, the system will never overwrite a node of tree T_i before checkpoint $i+1$ completes.

The system also may cache clean and dirty tree nodes in memory, although only the dirty nodes matter for recovery. Dirty nodes may be written back to disk at any time between two checkpoints. Thus, between two checkpoints, there may be two trees on disk, T_i and T_{i+1}, which may share nodes in a copy-on-write fashion.

Finally, the system maintains an in-memory buffer of recent log entries. These entries may be written out at any time (e.g., periodically), in response to a transaction commit, or because the buffer becomes full.

The system performs checkpoints as follows:

—All other activity is suspended.
—All entries in the log buffer are written to Log_{i+1}.
—All dirty nodes of T_{i+1} are written to disk, with allocation information recorded in Blockmap_{i+1}.
—Log_{i+1} is freed and space for Log_{i+2} is allocated in Blockmap_{i+1}.
—Blockmap_{i+1} is written to disk (without overwriting overwriting Blockmap_i). The checkpoint is now complete on disk.
—The in-memory copy of Blockmap_i is replaced with a copy of Blockmap_{i+1}.

Writing out Blockmap_{i+1} atomically moves the system from checkpoint i to $i+1$. If the system crashes before the write completes, then it will recover from checkpoint i. After the write completes, it will recover from checkpoint $i+1$. By writing out the new blockmap, all the nodes used by T_i but not T_{i+1} get freed, and all the nodes allocated by T_{i+1} have their allocation recorded to disk, the space used by Log_{i+1} is freed, and the space that will be used by Log_{i+2} is allocated.

B^ε-trees store full keys and values in the log by default. As a consequence, BetrFS implemented full data journaling. Since file blocks were stored as values keyed by their path and block number, every application-level write caused the file block to first be written to the log and then later written to the tree. This limits large sequential writes to at most half the disk bandwidth.

BetrFS 0.2 achieves the semantics of full data journaling, but without the overhead of writing data twice for large sequential writes. When an application performs a large sequential write, BetrFS 0.2 inserts the new key-value pairs into T_{i+1} as normal, which will cause some nodes to be marked dirty. For each key-value pair inserted into T_{i+1}, it appends a special "late-binding" log message to the in-memory buffer. A late-binding message specifies the key of the block written but does not contain the value. Later, when the system writes a dirty node containing an unbound key-value pair back to disk, either due to memory pressure or as part of a checkpoint, it appends a binding message that points to the original message and to the on-disk node containing that data. It also increments the reference count of that node's blocks in Blockmap_{i+1} so that the blocks will not be overwritten before the next checkpoint completes.

During checkpoint $i+1$, the system decrements the reference counts in Blockmap_{i+1} of all the blocks referenced by binding messages in Log_{i+1} immediately before writing out Blockmap_{i+1}. Our implementation keeps a simple in-memory list of all the blocks referenced by binding messages in Log_{i+1} for this purpose. Thus, writing out Blockmap_{i+1} simultaneously frees the space used by Log_{i+1} and decrements the reference counts of all blocks referenced by Log_{i+1}. Hence, no blocks are leaked or accidentally overwritten between checkpoints.

This approach does necessitate a second pass over the log during recovery. If the system crashes between checkpoint i and $i+1$, then there may be blocks referenced by the log but that are not marked as allocated in Blockmap_i. Thus, BetrFS 0.2 first scans the log and, for each binding message, adds the referenced block to its in-memory copy of Blockmap_i. It also constructs an in-memory list of all the late-binding messages for which there is no binding message. These unmatched late-binding messages will be ignored during recovery, since they must be part of uncommitted transactions. It then performs log replay as normal.

Late-binding journal. BetrFS 0.2 handles large messages, or large runs of consecutive messages, as follows and illustrated in Figure 1:

—A special *unbound log entry* is appended to the in-memory log buffer ①. An unbound log entry specifies an operation and a key, but not a value. These messages record the insert's logical order.

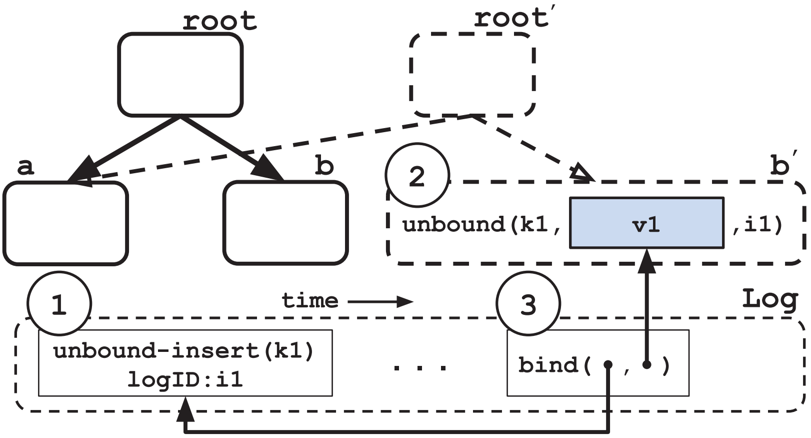


Fig. 1. Late-binding journaling in BetrFS. In step (1), keys are recorded in unbound entries and written to the log in memory. Next (2), values are inserted to the tree's copy-on-write nodes. To make these messages durable, all nodes containing unbound messages are written to disk, and a binding entry is written to the log (3). A binding entry records the physical location of an unbound message and a reference to the original unbound entry.

—A special *unbound message* is inserted into the B^{ε}-tree ②. An unbound message contains the key, value, and log entry ID of its corresponding unbound log entry. Unbound messages move down the tree like any other message.

—To make the log durable, all nodes containing unbound messages are first written to disk. As part of writing the node to disk, each unbound message is converted to a normal insert message (nonleaf node) or a normal key-value pair (leaf node). After an unbound message in a node is written to disk, a *binding log entry* is appended to the in-memory log buffer ③. Each binding log entry contains the log entry ID from the unbound message and the physical disk address of the node. Once all inserts in the in-memory log buffer are bound, the in-memory log buffer is written to disk.

—Node write-backs are handled similarly: when a node containing an unbound message is written to disk as part of a cache eviction or checkpoint, or for any other reason, binding entries are appended to the in-memory log buffer for all the unbound messages in the node, and the messages in the node are marked as bound.

The system can make logged operations durable at any time by writing out all the tree nodes that contain unbound messages and then flushing the log to disk. It is an invariant that all unbound inserts in the on-disk log will have matching binding log entries. Thus, recovery can always proceed to the end of the log.

The on-disk format does not change for an unbound insert: unbound messages exist only in memory.

The late-binding journal accelerates large messages. A negligible amount of data is written to the log, but a tree node is forced to be written to disk. If the amount of data to be written to a given tree node is equivalent to the size of the node, this reduces the bandwidth cost by half.

In the case where one or more inserts only account for a small fraction of the node, logging the values is preferable to unbound inserts. The issue is that an unbound insert can prematurely force the node to disk (at a log flush, rather than the next checkpoint), losing opportunities to batch more small modifications. Writing a node that is mostly unchanged wastes bandwidth. Thus, BetrFS 0.2 uses unbound inserts only when writing at least 1MB of consecutive pages to disk.

Crash recovery. Late-binding requires two passes over the log during recovery: one to identify nodes containing unbound inserts and a second to replay the log.

The core issue is that each checkpoint only records the on-disk nodes in use for that checkpoint. In BetrFS 0.2, nodes referenced by a binding log entry are not marked as allocated in the checkpoint's allocation table. Thus, the first pass is needed to update the allocation table to include all nodes referenced by binding log messages. The second pass replays the logical entries in the log. After the next checkpoint, the log is discarded, and the reference counts on all nodes referenced by the log are decremented. Any nodes whose reference count hits zero (i.e., because they are no longer referenced by other nodes in the tree) are garbage collected at that time.

Implementation. BetrFS 0.2 guarantees consistent recovery up until the last log flush or checkpoint. By default, a log flush is triggered on a `sync` operation, every second, or when the 32MB log buffer fills up. Flushing a log buffer with unbound log entries also requires searching the in-memory tree nodes for nodes containing unbound messages, in order to first write these nodes to disk. Thus, BetrFS 0.2 also reserves enough space at the end of the log buffer for the binding log messages. In practice, the log-flushing interval is long enough that most unbound inserts are written to disk before the log flush, minimizing the delay for a log write.

Additional optimizations. Section 5 explains some optimizations where logically obviated operations can be discarded as part of flushing messages down one level of the tree. One example is when a key is inserted and then deleted; if the insert and delete are in the same message buffer, the insert can be dropped, rather than flushed to the next level. In the case of unbound inserts, we allow a delete to remove an unbound insert before the value is written to disk under the following conditions: (1) all transactions involving the unbound key-value pair have committed, (2) the delete transaction has committed, and (3) the log has not yet been flushed. If these conditions are met, the file system can be consistently recovered without this unbound value. In this situation, BetrFS 0.2 binds obviated inserts to a special NULL node and drops the insert message from the B^{ε}-tree.

Level-to-level promotion. For large sequential writes, the TokuDB caching and flushing policies prevent most data from being rewritten at multiple levels in the tree. A portion of the B^{ε}-tree's nodes are cached in memory and evicted using an LRU policy. Data is inserted sequentially into the B^{ε}-tree root until it fills, at which point the newly written data must be flushed to a child node. As more data is inserted, flushes cascade, and the entire root-to-leaf path is cached and dirtied. Since interior nodes continue to be dirtied by node flushes, it is primarily the leaf nodes that are evicted and written to disk. Thus, with the exception of residual data at the tail of large I/Os, data is only written at the lowest level, in the leaf nodes.

4. BALANCING SEARCH AND RENAME

In this section, we argue that there is a design tradeoff between the performance of renames and recursive directory scans. We present an algorithmic framework for picking a point along this tradeoff curve.

Conventional file systems support fast renames at the expense of slow recursive directory traversals. Each file and directory is assigned its own inode, and names in a directory are commonly mapped to inodes with pointers. Renaming a file or directory can be very efficient, requiring only creation and deletion of a pointer to an inode, and a constant number of I/Os. However, searching files or subdirectories within a directory requires traversing all these pointers. When the inodes under a directory are not stored together on disk, for instance, because of renames, then each pointer traversal can require a disk seek, severely limiting the speed of the traversal.

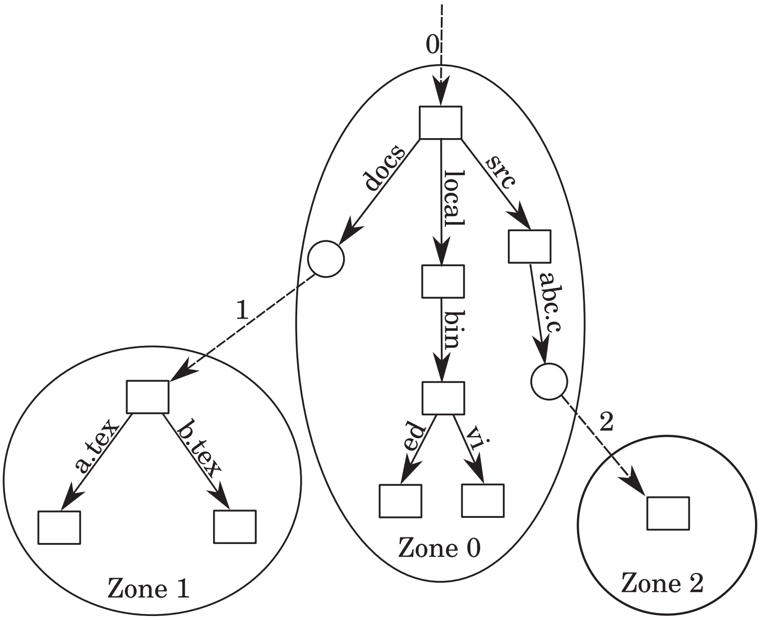


Fig. 2. An example zone tree in BetrFS 0.2.

BetrFS 0.1 and TokuFS are at the other extreme. They index every directory, file, and file block by its full path in the file system. The sort order on paths guarantees that all the entries beneath a directory are stored contiguously in logical order within nodes of the B$^{\varepsilon}$-tree, enabling fast scans over entire subtrees of the directory hierarchy. Renaming a file or directory, however, requires physically moving every file, directory, and block to a new location.

This tradeoff is common in file system design. Some file systems try to get intermediate points between these extremes. TableFS stores files that are smaller than 4KB with metadata in LSM-tree to reduce disk access in "readdir+read" workloads, at the expense of moving data in small file renames. Btrfs also stores small files inside the metadata tree. Fast directory traversals require on-disk locality, and renames need to do more work to maintain locality.

BetrFS 0.2's schema makes this tradeoff parameterizable and tunable by partitioning the directory hierarchy into connected regions, which we call *zones*. Figure 2 shows how files and directories within subtrees are collected into zones in BetrFS 0.2. Each zone has a unique zone-ID, which is analogous to an inode number in a traditional file system. Each zone contains either a single file or a single root directory, which we call the root of the zone. Files and directories are identified by their zone-ID and their relative path within the zone.

Directories and files within a zone are stored together, enabling fast scans within that zone. Crossing a zone boundary potentially requires a seek to a different part of the tree. Renaming a file or a directory under a zone root moves the data and metadata in the zone, whereas renaming a large file or directory (a zone root) requires only changing a pointer.

Zoning supports a spectrum of tradeoff points between the two extremes described previously. When zones are restricted to size 1, the BetrFS 0.2 schema is equivalent to an inode-based schema as each file and directory is stored in its own zone. If we set the zone size bound to infinity (∞), then BetrFS 0.2's schema is equivalent to BetrFS 0.1's

Metadata Index

(0, ”/”)	→	metadata for “/”
(0, ”/docs”)	→	zone 1
(0, ”/local”)	→	metadata for “/local”
(0, ”/src”)	→	metadata for “/src”
(0, ”/local/bin”)	→	metadata for “/local/bin”
(0, ”/local/bin/ed”)	→	metadata for “/local/bin/ed”
(0, ”/local/bin/vi”)	→	metadata for “/local/bin/vi”
(0, ”/src/abc.c”)	→	zone 2
(1, ”/”)	→	metadata for “/docs”
(1, ”/a.tex”)	→	metadata for “/docs/a.tex”
(1, ”/b.tex”)	→	metadata for “/docs/b.tex”
(2, ”/”)	→	metadata for “/src/abc.c”

Data Index

(0, “/local/bin/ed”, i)	→	block i of “/local/bin/ed”
(0, “/local/bin/vi”, i)	→	block i of “/local/bin/vi”
(1, “/a.tex”, i)	→	block i of “/docs/a.tex”
(1, “/b.tex”, i)	→	block i of “/docs/b.tex”
(2, “/”, i)	→	block i of “/src/abc.c”

Fig. 3. Example metadata and data indices in BetrFS 0.2.

schema since only one zone exists. At an intermediate setting, BetrFS 0.2 can balance the performance of directory scans and renames.

The default zone size in BetrFS 0.2 is 512 KiB. Intuitively, moving a very small file is sufficiently inexpensive that indirection would save little, especially in a WOD. On the other extreme, once a file system is reading several megabytes between each seek, the dominant cost is transfer time, not seeking. Thus, one would expect the best zone size to be between tens of kilobytes and a few megabytes. We also note that this tradeoff is somewhat implementation dependent: the more efficiently a file system can move a set of keys and values, the larger a zone can be without harming rename performance. Section 7 empirically evaluates these tradeoffs.

As an effect of zoning, BetrFS 0.2 supports hard links. Because BetrFS 0.1 stores data blocks for each file and has no effective way to propagate updates to all files sharing the link, hard links are not supported in BetrFS 0.1. BetrFS 0.2 moves a file with more than one link into its own zone and places one pointer for each path. Therefore, updates can be done in the zone.

Metadata and data indexes. The BetrFS 0.2 metadata index maps (zone-ID, relative-path) keys to metadata about a file or directory, as shown in Figure 3. For a file or directory in the same zone, the metadata includes the typical contents of a `stat` structure, such as owner, modification time, and permissions. For instance, in zone 0, path “/local” maps onto the stat info for this directory. If this key (i.e., relative path within the zone) maps onto a different zone, then the metadata index maps onto the ID of that zone. For instance, in zone 0, path “/docs” maps onto zone-ID 1, which is the root of that zone.

Besides the `stat` structure, the metadata also contains bookkeeping information for managing zones. Each file and directory stores `nr_meta`, which is the number of metadata entries that reside in its subtree and share the same zone-ID. Likewise, each file and directory have `nr_data` for the number of data entries. In the example shown in Figure 3, the `nr_meta` of (0, “/src”) is 2, counting itself and (0, “/src/abc.c”), while the `nr_data` of (0, “/src”) is 0 because no data block in the subtree has zone-ID 0. BetrFS 0.2 uses this bookkeeping information to determine whether to split or merge zones.

The data index maps (zone-ID, relative-path, block-number) to the content of the specified file block.

Path sorting order. BetrFS 0.2 sorts keys by zone-ID first, and then by their relative path. Since all the items in a zone will be stored consecutively in this sort order, recursive directory scans can visit all the entries within a zone efficiently. Within a zone, entries are sorted by path in a "depth-first-with-children" order, as illustrated in Figure 3. The sort order can be viewed as, for a directory, we first place all direct children of the directory and then recursively do the same procedure for each subdirectory. Compared to BetrFS 0.1 that sorts entries first by the number of slashes and then by alphabet, this sort order ensures that all the entries beneath a directory are stored logically contiguously in the underlying key-value store, followed by recursive listings of the subdirectories of that directory. Thus, an application that performs `readdir` on a directory and then recursively scans its subdirectories in the order returned by `readdir` will effectively perform range queries on that zone and each of the zones beneath it.

ALGORITHM 1: Algorithm for Comparing Two Paths in BetrFS 0.2

```
function PATH-COMPARE($a$, $b$)
    if $a = b$ then
        return 0
    else
        let $a_0, \ldots, a_{k-1}$ be the path components of $a$
        let $b_0, \ldots, b_{\ell-1}$ be the path components of $b$
        let $a_k = \perp$, $b_\ell = \perp$
        let $i$ be the smallest integer such that $a_i \neq b_i$
        if $i \geq k - 1$ and $k < \ell$ then
            return 1                                  ▷ $b$ is in a subdir of $a$'s parent
        else if $i \geq \ell - 1$ and $\ell < k$ then
            return $-1$                               ▷ $a$ is in a subdir of $b$'s parent
        else                                          ▷ $a$ and $b$ are either entries in a common
                                                      ▷ parent, or in subdirs of a common ancestor
            return STRING-COMPARE($a_i$, $b_i$)
        end if
    end if
end function
```

Rename. Renaming a file or directory that is the root of its zone requires simply inserting a reference to its zone at its new location and deleting the old reference. So, for example, renaming "/src/abc.c" to "/docs/def.c" in Figure 3 requires deleting key (0, "/src/abc.c") from the metadata index and inserting the mapping (1, "/def.c") → Zone 2.

Renaming a file or directory that is not the root of its zone requires copying the contents of that file or directory to its new location. So, for example, renaming "/local/bin" to "/docs/tools" requires (1) deleting all the keys of the form (0, "/local/bin/p") in the metadata index, (2) reinserting them as keys of the form (1, "/tools/p"), (3) deleting all keys of the form (0, "/local/bin/p", i) from the data index, and (4) reinserting them as keys of the form (1, "/tools/p", i). Note that renaming a directory never requires recursively moving into a child zone. Thus, by bounding the size of the directory subtree within a single zone, we also bound the amount of work required to perform a rename.

Splitting and merging. To maintain a consistent rename and scan performance tradeoff throughout system lifetime, zones must be split and merged so that the following two invariants are upheld:

ZoneMin: Each zone has size at least C_0.
ZoneMax: Each directory that is not the root of its zone has size at most C_1.

The ZoneMin invariant ensures that recursive directory traversals will be able to scan through at least C_0 consecutive bytes in the key-value store before initiating a scan of another zone, which may require a disk seek. The ZoneMax invariant ensures that no directory rename will require moving more than C_1 bytes.

The BetrFS 0.2 design upholds these invariants as follows. Each inode maintains two counters `nr_meta` and `nr_data` as described previously. Whenever a data or metadata entry is added or removed, BetrFS 0.2 recursively updates counters from the corresponding file or directory up to its zone root. If either of a file or directory's counters exceed C_1, BetrFS 0.2 creates a new zone for the entries in that file or directory. When a zone size falls below C_0, that zone is merged with its parent. BetrFS 0.2 avoids cascading splits and merges by merging a zone with its parent only when doing so would not cause the parent to split. To avoid unnecessary merges during a large directory deletion, BetrFS 0.2 defers merging until writing back dirty inodes.

We can tune the tradeoff between rename and directory traversal performance by adjusting C_0 and C_1. Larger C_0 will improve recursive directory traversals. However, increasing C_0 beyond the block size of the underlying data structure will have diminishing returns, since the system will have to seek from block to block during the scan of a single zone. Smaller C_1 will improve rename performance. All objects larger than C_1 can be renamed in a constant number of I/Os, and the worst-case rename requires only C_1 bytes be moved. In the current implementation, $C_0 = C_1$, and by default, they are 512KiB.

The zone schema enables BetrFS 0.2 to support a spectrum of tradeoffs between rename performance and directory traversal performance. We explore these tradeoffs empirically in Section 7.

5. EFFICIENT RANGE DELETION

This section explains how BetrFS 0.2 obtains nearly flat deletion times by introducing a new *rangecast* message type to the B$^\varepsilon$-tree and implementing several B$^\varepsilon$-tree-internal optimizations using this new message type.

BetrFS 0.1 file and directory deletion performance is linear in the amount of data being deleted. Although this is true to some extent in any file system, as the freed disk space will be linear in the file size, the slope for BetrFS 0.1 is alarming. For instance, unlinking a 4GB file takes 5 minutes on BetrFS 0.1!

Two underlying issues are the sheer volume of delete messages that must be inserted into the B$^\varepsilon$-tree and missed optimizations in the B$^\varepsilon$-tree implementation. Because the B$^\varepsilon$-tree implementation does not bake in any semantics about the schema, the B$^\varepsilon$-tree cannot infer that two keys are adjacent in the keyspace. Without hints from the file system, a B$^\varepsilon$-tree cannot optimize for the common case of deleting large, contiguous key ranges.

5.1. Rangecast Messages

In order to support deletion of a key range in a single message, we added a *rangecast* message type to the B$^\varepsilon$-tree implementation. In the baseline B$^\varepsilon$-tree implementation, updates of various forms (e.g., insert and delete) are encoded as messages addressed to a single key, which, as explained in Section 2, are flushed down the path from root to leaf. A rangecast message can be addressed to a contiguous range of keys, specified by the beginning and ending keys, inclusive. These beginning and ending keys need not exist, and the range can be sparse; the message will be applied to any keys in the range that do exist. We have currently added rangecast delete messages, but we can envision range insert and upsert [Jannen et al. 2015a] being useful.

Rangecast message propagation. When single-key messages are propagated from a parent to a child, they are simply inserted into the child's buffer space in logical order (or in key order when applied to a leaf). Rangecast message propagation is similar to regular message propagation, with two differences.

First, rangecast messages may be applied to multiple children at different times. When a rangecast message is flushed to a child, the propagation function must check whether the range spans multiple children. If so, the rangecast message is transparently split and copied for each child, with appropriate subsets of the original range. If a rangecast message covers multiple children of a node, the rangecast message can be split and applied to each child at different points in time—most commonly, deferring until there are enough messages for that child to amortize the flushing cost. As messages propagate down the tree, they are stored and applied to leaves in the same commit order. Thus, any updates to a key or reinsertions of a deleted key maintain a global serial order, even if a rangecast spans multiple nodes.

Second, when a rangecast delete is flushed to a leaf, it may remove multiple key/value pairs, or even an entire leaf. Because `unlink` uses rangecast delete, all of the data blocks for a file are freed atomically with respect to a crash.

Query. A B$^{\varepsilon}$-tree query must apply all pending modifications in node buffers to the relevant key(s). Applying these modifications is efficient because all relevant messages will be in a node's buffer on the root-to-leaf search path. Rangecast messages maintain this invariant.

Each B$^{\varepsilon}$-tree node maintains a FIFO queue of pending messages and, for single-key messages, a balanced binary tree sorted by the messages' keys. For rangecast messages, our current prototype checks a simple list of rangecast messages and interleaves the messages with single-key messages based on commit order. This search cost is linear in the number of rangecast messages. A faster implementation would store the rangecast messages in each node using an interval tree, enabling it to find all the rangecast messages relevant to a query in $O(k + \log n)$ time, where n is the number of rangecast messages in the node and k is the number of those messages relevant to the current query.

Rangecast messages can potentially speed up range query. TokuDB developed a heuristic strategy to determine if the messages of a node, even if its message buffer is not yet full, are worth flushing down to speed up the query. The strategy is based on calculating how much data in the leaves have been impacted by applying the messages in this node. This way a rangecast message that impacts a much wider range or a message that has been applied to the leaves over and over again has a higher chance to be flushed down soon.

Rangecast `unlink` and `truncate`. In the BetrFS 0.2 schema, 4KB data blocks are keyed by a concatenated tuple of zone ID, relative path, and block number. Unlinking a file involves one delete message to remove the file from the metadata index and, in the same B$^{\varepsilon}$-tree-level transaction, a rangecast delete to remove all of the blocks. Deleting all data blocks in a file is simply encoded by using the same prefix, but from blocks 0 to infinity. Truncating a file works the same way but can start with a block number other than zero and does not remove the metadata key.

5.2. B$^{\varepsilon}$-Tree-Internal Optimizations

The ability to group a large range of deletion messages not only reduces the number of total delete messages required to remove a file but also creates new opportunities for B$^{\varepsilon}$-tree-internal optimizations.

Leaf pruning. When a B$^{\varepsilon}$-tree flushes data from one level to the next, it must first read the child, merge the incoming data, and rewrite the child. In the case of a large, sequential write, a large range of obviated data may be read from disk, only to be overwritten. In the case of BetrFS 0.1, unnecessary reads make overwriting a 10GB file 30–63MB/s slower than the first write of the file.

The leaf pruning optimization identifies when an entire leaf is obviated by a range delete and elides reading the leaf from disk. When a large range of consecutive keys and values are inserted, such as overwriting a large file region, BetrFS 0.2 includes a range delete for the key range in the same transaction. This range delete message is necessary, as the B^{ε}-tree cannot infer that the range of the inserted keys are contiguous; the range delete communicates information about the keyspace. On flushing messages to a child, the B^{ε}-tree can detect when a range delete encompasses the child's keyspace. BetrFS 0.2 uses transactions inside the B^{ε}-tree implementation to ensure that the removal and overwrite are atomic: at no point can a crash lose both the old and new contents of the modified blocks. Stale leaf nodes are reclaimed as part of normal B^{ε}-tree garbage collection.

Thus, this leaf pruning optimization avoids expensive reads when a large file is being overwritten. This optimization is both essential to sequential I/O performance and possible only with rangecast delete.

Pac-Man. A rangecast delete can also obviate a significant number of buffered messages. For instance, if a user creates a large file and immediately deletes the file, the B^{ε}-tree may include many obviated insert messages that are no longer profitable to propagate to the leaves.

BetrFS 0.2 adds an optimization to message flushing, where a rangecast delete message can devour obviated messages ahead of it in the commit sequence. We call this optimization "Pac-Man," in homage to the arcade game character known for devouring ghosts. This optimization further reduces background work in the tree, eliminating "dead" messages before they reach a leaf.

6. OPTIMIZED STACKING

BetrFS has a stacked file system design [Jannen et al. 2015a]; B^{ε}-tree nodes and the journal are stored as files on an ext4 file system. BetrFS 0.2 corrects two points where BetrFS 0.1 was using the underlying ext4 file system suboptimally.

First, in order to ensure that nodes are physically placed together, TokuDB writes zeros into the node files to force space allocation in larger extents. For sequential writes to a new FS, BetrFS 0.1 zeros these nodes and then immediately overwrites the nodes with file contents, wasting up to a third of the disk's bandwidth. We replaced this with the newer `fallocate` API, which can physically allocate space but logically zero the contents.

Second, the I/O to flush the BetrFS journal file was being amplified by the ext4 journal. Each BetrFS log flush appended to a file on ext4, which required updating the file size and allocation. BetrFS 0.2 reduces this overhead by preallocating space for the journal file and using `fdatasync`.

7. EVALUATION

Our evaluation targets the following questions:

- —How does one choose the zone size?
- —Does BetrFS 0.2 perform comparably to other file systems on the worst cases for BetrFS 0.1?
- —Does BetrFS 0.2 perform comparably to BetrFS 0.1 on the best cases for BetrFS 0.1?
- —How do BetrFS 0.2 optimizations impact application performance? Is this performance comparable to other file systems and as good as or better than BetrFS 0.1?
- —What are the costs of background work in BetrFS 0.2?

All experimental results were collected on a Dell Optiplex 790 with a four-core 3.40GHz Intel Core i7 CPU, 4GB RAM, and a 500GB, 7200 RPM ATA disk, with a 4,096-byte block size. Each file system's block size is 4,096 bytes. The system ran Ubuntu 13.10, 64-bit, with Linux kernel version 3.11.10. Each experiment is compared with several file systems, including BetrFS 0.1 [Jannen et al. 2015a], btrfs [Rodeh et al. 2013], ext4 [Mathur et al. 2007], XFS [Sweeney et al. 1996], and zfs [Bonwick and Moore 2005]. We use the versions of XFS, btrfs, and ext4 that are part of the 3.11.10 kernel, and zfs 0.6.3, downloaded from www.zfsonlinux.org. The disk was divided into two partitions roughly 240GB each: one for the root FS and the other for experiments. We use default recommended file system settings unless otherwise noted. Lazy inode table and journal initialization were turned off on ext4. Each experiment was run a minimum of four times. Error bars and ± ranges denote 95% confidence intervals. Unless noted, all benchmarks are cold-cache tests.

7.1. Choosing a Zone Size

This subsection quantifies the impact of zone size on rename and scan performance.

A good zone size limits the worst-case costs of `rename` but maintains data locality for fast directory scans. Figure 4 shows the average cost to `rename` a file and fsync the parent directory, over 100 iterations, plotted as a function of size. We show BetrFS 0.2 with an infinite zone size (no zones are created—`rename` moves all file contents) and 0 (every file is in its own zone—`rename` is a pointer swap). Once a file is in its own zone, the performance is comparable to most other file systems (16ms on BetrFS 0.2 compared to 17ms on ext4). This is balanced against Figure 5, which shows `grep` performance versus zone size. As predicted in Section 4, directory-traversal performance improves as the zone size increases.

We select a default zone size of 512 KiB, which enforces a reasonable bound on worst-case `rename` (compared to an unbounded BetrFS 0.1 worst case) and keeps search performance within 25% of the asymptote. Figure 7 compares BetrFS 0.2 rename time to other file systems. Specifically, worst-case rename performance at this zone size is 66ms, 3.7× slower than the median file system's rename cost of 18ms. However, renames of files 512KiB or larger are comparable to other file systems, and search performance is 2.2× the best baseline file system and 8× the median. We use this zone size for the rest of the evaluation.

7.2. Improving the Worst Cases

This subsection measures BetrFS 0.1's three worst cases and shows that, for typical workloads, BetrFS 0.2 is either faster or within roughly 10% of other file systems.

Sequential writes. Figure 8 shows the throughput to sequentially read and write a 10GiB file (more than twice the size of the machine's RAM). The optimizations described in Section 3 improve the sequential write throughput of BetrFS 0.2 to 96MiB/s, up from 28MiB/s in BetrFS 0.1. Except for zfs, the other file systems realize roughly 10% higher throughput. We also note that these file systems offer different crash consistency properties: ext4 and XFS only guarantee metadata recovery, whereas zfs, btrfs, and BetrFS guarantee data recovery.

The sequential read throughput of BetrFS 0.2 is improved over BetrFS 0.1 by roughly 12MiB/s, which is attributable to streamlining the code. This places BetrFS 0.2 within striking distance of other file systems.

Rename. Table I shows the execution time of several common directory operations on the Linux 3.11.10 source tree. The `rename` test renames the entire source tree. BetrFS 0.1 directory rename is two orders of magnitude slower than any other file system, whereas BetrFS 0.2 is faster than every other file system except XFS. By partitioning the directory hierarchy into zones, BetrFS 0.2 ensures that the cost of a rename is comparable to other file systems.

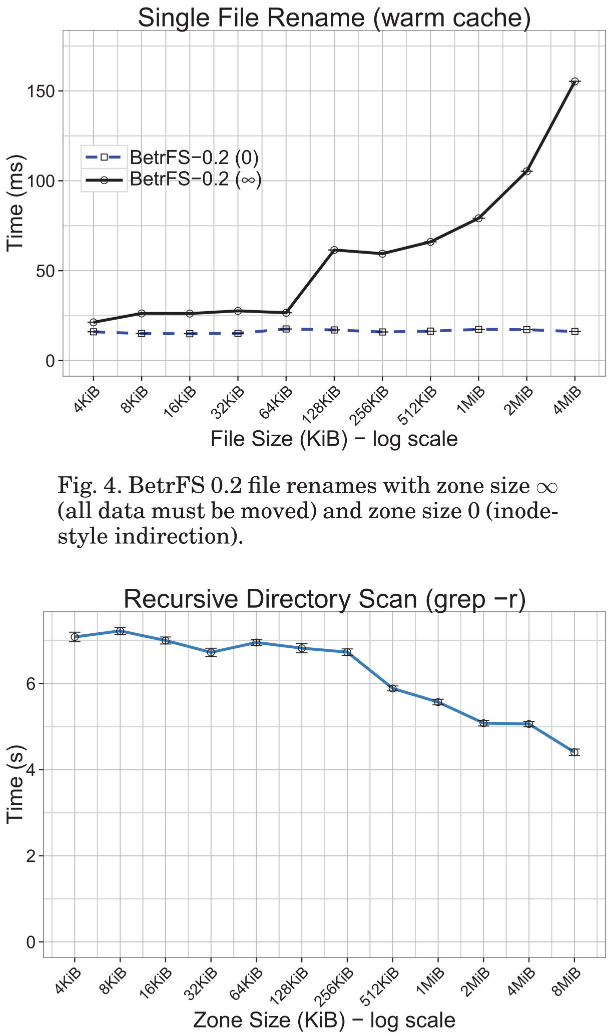


Fig. 4. BetrFS 0.2 file renames with zone size ∞ (all data must be moved) and zone size 0 (inode-style indirection).

Fig. 5. Recursive scans of the Linux 3.11.10 source for “cpu_to_be64” with different BetrFS 0.2 zone sizes.

Fig. 6. The impact of zone size on rename and scan performance. Lower is better.

Unlink. Table I also includes the time to recursively delete the Linux source tree. Again, whereas BetrFS 0.1 is an order of magnitude slower than any other file system, BetrFS 0.2 is *faster*. We attribute this improvement to BetrFS 0.2’s fast directory traversals and to the effectiveness of range deletion.

We also measured the latency of unlinking files of increasing size. Due to scale, we contrast BetrFS 0.1 with BetrFS 0.2 in Figure 9(a), and we compare BetrFS 0.2 with other file systems in Figure 9(b). In BetrFS 0.1, the cost to delete a file scales linearly with the file size. Figure 9(b) shows that BetrFS 0.2 delete latency is not sensitive to file size. Measurements show that zfs performance is considerably slower and noisier; we suspect that this variance is attributable to unlinking incurring amortized housekeeping work.

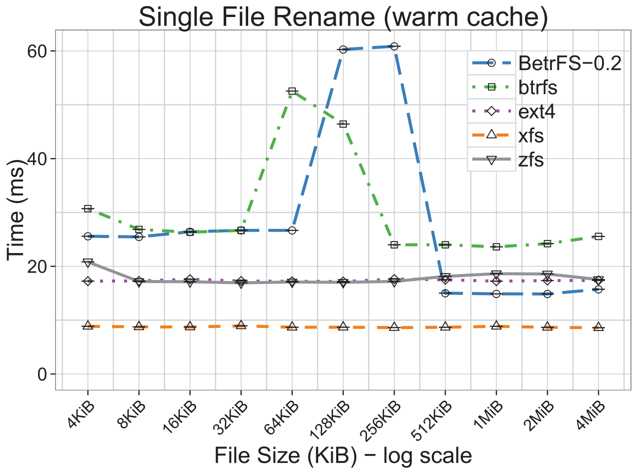


Fig. 7. Time to rename single files. Lower is better.

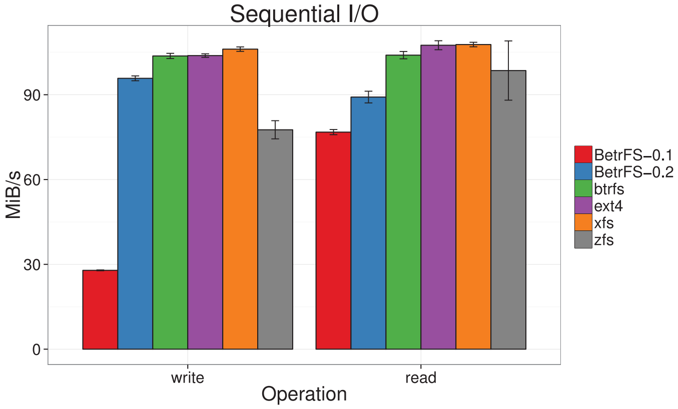


Fig. 8. Large file I/O performance. We sequentially read and write a 10GiB file. Higher is better.

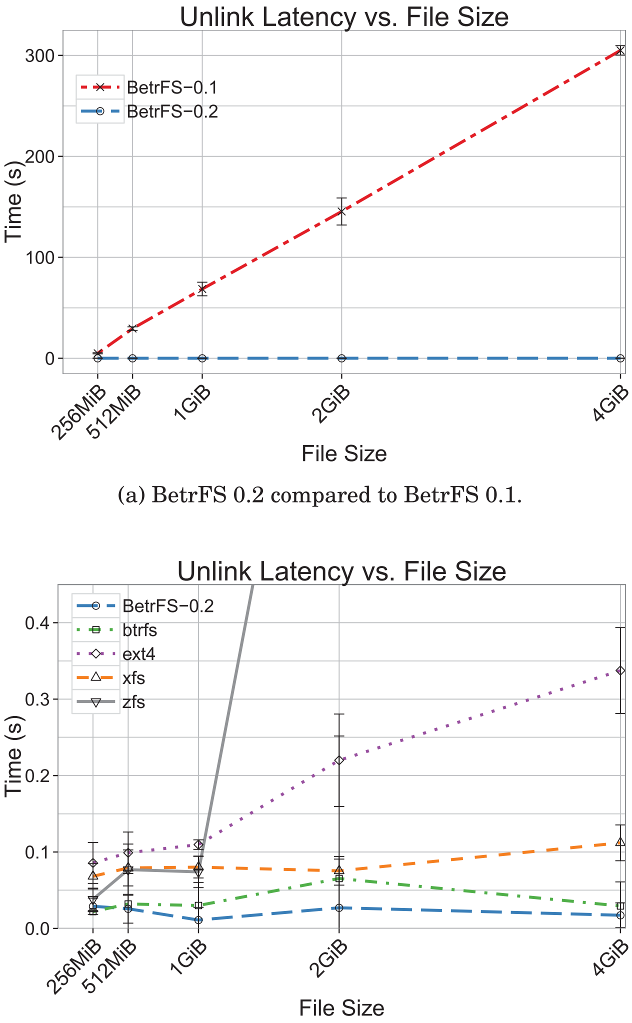
Table I. Time in Seconds to Complete Directory Operations on the Linux 3.11.10 Source: `find` of the File "wait.c", `grep` of the String "cpu_to_be64", `mv` of the Directory Root, and `rm -rf` (Lower Is Better)

File System	find	grep	mv	rm -rf
BetrFS 0.1	0.36 ± 0.0	3.95 ± 0.2	21.17 ± 0.7	46.14 ± 0.8
BetrFS 0.2	0.35 ± 0.0	5.78 ± 0.1	0.13 ± 0.0	2.37 ± 0.2
btrfs	4.84 ± 0.7	12.77 ± 2.0	0.15 ± 0.0	9.63 ± 1.4
ext4	3.51 ± 0.3	49.61 ± 1.8	0.18 ± 0.1	4.17 ± 1.3
xfs	9.01 ± 1.9	61.09 ± 4.7	0.08 ± 0.0	8.16 ± 3.1
zfs	13.71 ± 0.6	43.26 ± 1.1	0.14 ± 0.0	13.23 ± 0.7

7.3. Maintaining the Best Cases

This subsection evaluates the best cases for a write-optimized file system, including small random writes, file creation, and searches. We confirm that our optimizations have not eroded the benefits of write optimization. In most cases, there is no loss.

Small, random writes. Table II shows the execution time of a microbenchmark that issues 10,000 four-byte overwrites at random offsets within a 10GiB file, followed by



(a) BetrFS 0.2 compared to BetrFS 0.1.

(b) BetrFS 0.2 compared to commodity file systems.

Fig. 9. Unlink latency by file size. Lower is better.

Table II. Time to Perform 10,000 Four-Byte Overwrites on a 10GiB File (Lower Is Better)

File System	Time (s)
BetrFS 0.1	0.48 ± 0.1
BetrFS 0.2	0.32 ± 0.0
btrfs	104.18 ± 0.3
ext4	111.20 ± 0.4
xfs	111.03 ± 0.4
zfs	131.86 ± 12.6

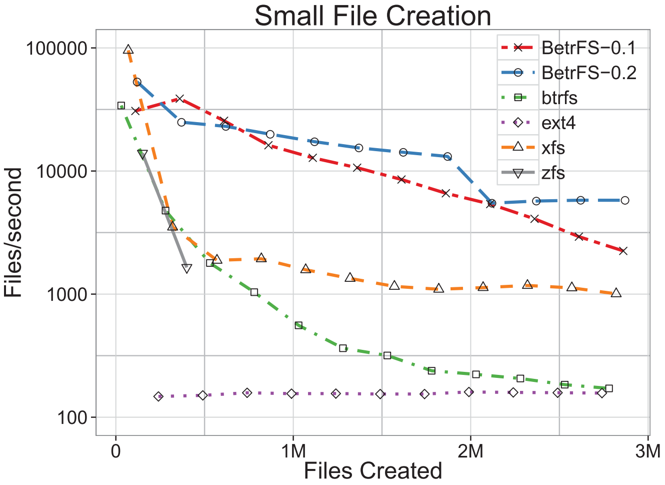


Fig. 10. Sustained file creation for 3 million 200-byte files, using four threads. Higher is better; y-axis is log scale.

an `fsync`. BetrFS 0.2 not only retains a two orders-of-magnitude improvement over the other file systems but also improves the latency over BetrFS 0.1 by 34%.

Small file creation. To evaluate file creation, we used the TokuBench benchmark [Esmet et al. 2012] to create 3 million 200-byte files in a balanced directory tree with a fanout of 128. We used four threads, one per core of the machine.

Figure 10 graphs files created per second as a function of the number of files created. In other words, the point at 1 million on the x-axis is the cumulative throughput at the time the millionth file is created. zfs exhausts system memory after creating around a half-million files.

The line for BetrFS 0.2 is mostly higher than the line for BetrFS 0.1 , and both sustain throughputs at least 3×, but often an order of magnitude, higher than any other file system (note the y-axis is log scale). Due to TokuBench's balanced directory hierarchy and write patterns, BetrFS 0.2 performs 16,384 zone splits in quick succession at around 2 million files. This leads to a sudden drop in performance and immediate recovery.

Searches. Table I shows the time to search for files named "wait.c" (`find`) and to search the file contents for the string "cpu_to_be64" (`grep`). These operations are comparable on both write-optimized file systems, although BetrFS 0.2 `grep` slows by 46%, which is attributable to the tradeoffs to add zoning.

7.4. Application Performance

This subsection evaluates the impact of the BetrFS 0.2 optimizations on the performance of several applications, shown in Figure 15. Figure 11 shows the throughput of an `rsync`, with and without the `--in-place` flag. In both cases, BetrFS 0.2 improves the throughput over BetrFS 0.1 and maintains a significant improvement over other file systems. Faster sequential I/O and, in the second case, faster rename contribute to these gains.

In the case of `git-clone`, sequential write improvements make BetrFS 0.2 performance comparable to other file systems, unlike BetrFS 0.1. Similarly, BetrFS 0.2 marginally improves the performance of `git-diff`, making it clearly faster than the other FSes.

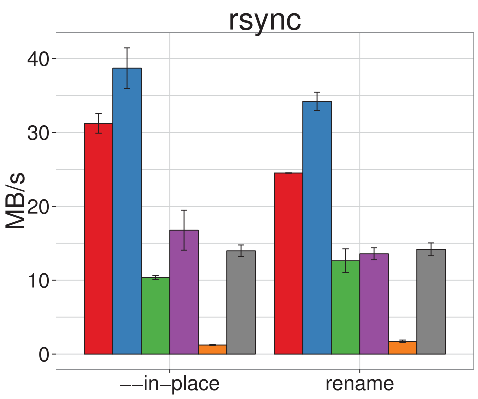


Fig. 11. `rsync` of Linux 3.11.10 source. The data source and destination are within the same partition and file system. Throughput is in MB/s; higher is better.

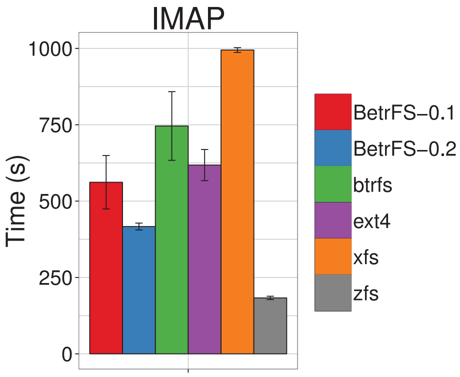


Fig. 12. IMAP benchmark using Dovecot 2.2.13, 50% message reads, 50% marking and moving messages among inboxes. Execution time is in seconds; lower is better.

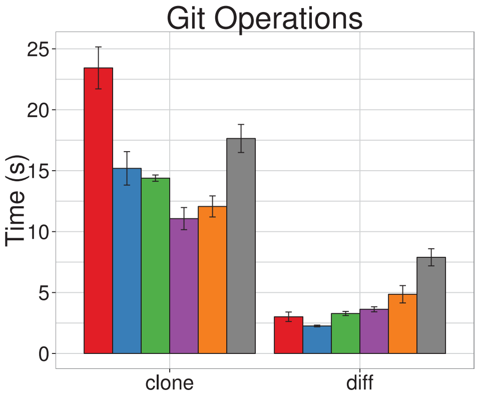


Fig. 13. Git operations. The BetrFS source repository was `git-cloned` locally, and a `git-diff` was taken between two milestone commits. Lower is better.

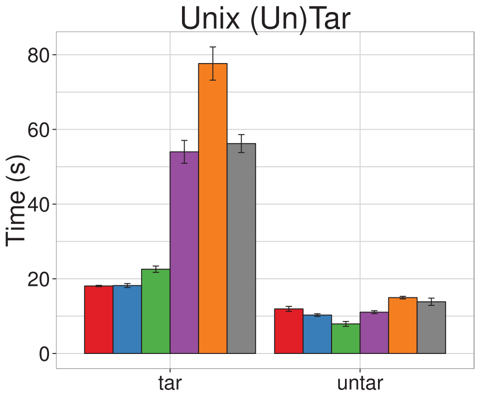


Fig. 14. Unix `tar` operations. The Linux version 3.11.10 source code was both `tar`ed and un-`tar`ed. Time is in seconds. Lower is better.

Fig. 15. Application benchmarks.

Both BetrFS 0.2 and zfs outperform other file systems on the Dovecot IMAP workload, although zfs is the fastest. This workload is characterized by frequent small writes and `fsync`s, and both file systems persist small updates quickly by flushing their logs.

On BetrFS 0.2, `tar` is 1% slower than BetrFS 0.1 due to the extra work of splitting zones.

7.5. Background Costs

This subsection evaluates the overheads of deferred work attributable to batching in a WOD. To measure the cost of deferred writes, we compare the time to read a

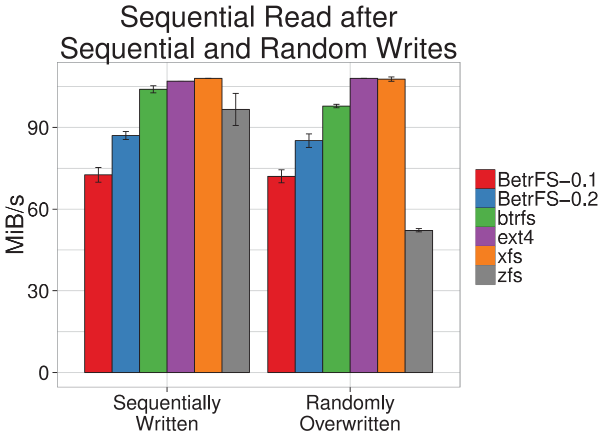


Fig. 16. Sequential read throughput after sequentially writing a 10GiB file (left) and after partially overwriting 10,000 random blocks in the file (right). Higher is better.

sequentially written 10GiB file to the time to read that same file after partially overwriting 10,000 random blocks. Both reads are cold cache and shown in Figure 16.

This experiment demonstrates that BetrFS 0.2's effective read throughput is nearly identical (87MiB/s vs. 85MiB/s), regardless of how the file was written.

8. RELATED WORK

Zoning. Dynamic subtree partitioning [Weil et al. 2004] is a technique designed for large-scale distributed systems, like Ceph [Weil et al. 2006], to reduce metadata contention and balance load. These systems distribute (1) the number of metadata objects and (2) the frequency of metadata accesses, across nodes. Zones instead partition objects according to their aggregate *size* to bound rename costs.

Spyglass [Leung et al. 2009] introduces a partitioning technique for multidimensional metadata indices based on KD-trees. Partitioning techniques have also been used to determine which data go on slower versus faster media [Mitra et al. 2008], to efficiently maintain inverted document indices [Lester et al. 2005], or to plug in different storage data structures to optimize for read- or write-intensive workloads [Mammarella et al. 2009]. Chunkfs [Henson et al. 2006] partitions the ext2 file system to improve recovery time. A number of systems also divide disk bandwidth and cache space for performance isolation [Verghese et al. 1998; Wachs et al. 2007]; speaking generally, these systems are primarily concerned with fairness across users or clients, rather than bounding worst-case execution time. These techniques strike domain-specific tradeoffs different from zoning's balance of directory searches and renames.

IceFS [Lu et al. 2014] uses cubes, a similar concept to zones, to isolate faults, remove physical dependencies in data structures and transactional mechanisms, and allow for finer granularity recovery and journal configuration. Cubes are explicitly defined by users to consist of an entire directory subtree and can grow arbitrarily large as users add more data. In contrast, zones are completely transparent to users, and dynamically split and merged.

SpanFS [Kang et al. 2015] distributes files and directories into different domains to accelerate file system journaling. Specifically, each domain maintains its own data structures for journaling, so the effect of *fsync*ing one file can be limited in its domain without causing contention on global data structures. On SpanFS, the number of domains is predefined during formatting, while in BetrFS 0.2, zones are dynamically created.

Late-binding log entries. KVFS [Shetty et al. 2013] avoids the journaling overhead of writing most data twice by creating a new VT-tree snapshot for each transaction. When a transaction commits, all in-memory data from the transaction's snapshot VT-tree is committed to disk, and that transaction's VT-tree is added *above* dependent VT-trees. Data is not written twice in this scenario, but the VT-tree may grow arbitrarily tall, making search performance difficult to reason about.

Log-structured file systems [Andersen et al. 2009; Lim et al. 2011; Rosenblum and Ousterhout 1992; Lee et al. 2015] avoid the problem of duplicate writes by only writing into a log. This improves write throughput in the best cases but does not enforce an optimal lower bound on query time.

Physical logging [Gray and Reuter 1992] stores before-and-after images of individual database pages, which may be expensive for large updates or small updates to large objects. Logical logging [Lomet and Tuttle 1999] may reduce log sizes when operations have succinct representations, but not for large data inserts.

The zfs intent log combines copy-on-write updates and indirection to avoid log write amplification for large records [McKusick et al. 2014]. We adapt this technique to implement late-binding journaling of large messages (or large groups of related small messages) in BetrFS 0.2.

The Okeanos wasteless journaling [Hatzieleftheriou and Anastasiadis 2011] transfers large requests to the final location without journaling of data but treats small requests according to wasteless journaling, which essentially merges subblock writes. The current prototype is only implemented on ext3.

Previous systems have implemented variations of soft updates [Mckusick and Ganger 1999], where data is written first, followed by metadata, from leaf to root. This approach orders writes so that on-disk structures are always a consistent checkpoint. Although soft updates may be possible in a B^{ε}-tree, this would be challenging. Like soft updates, the late-binding journal avoids the problem of doubling large writes but, unlike soft updates, is largely encapsulated in the block allocator. Late-binding imposes few additional requirements on the B^{ε}-tree itself and does not delay writes of any tree node to enforce ordering. Thus, a late-binding journal is particularly suitable for a WOD.

9. CONCLUSION

This article shows that write-optimized dictionaries can be practical not just to accelerate special cases, but as a building block for general-purpose file systems. BetrFS 0.2 improves the performance of certain operations by orders of magnitude and offers performance comparable to commodity file systems on all others. These improvements are the product of fundamental advances in the design of write-optimized dictionaries. We believe some of these techniques may be applicable to broader classes of file systems, which we leave for future work.

The source code for BetrFS 0.2 is available under GPLv2 at github.com/oscarlab/betrfs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Eddie Kohler for their insightful comments on earlier drafts of the work. Nafees Abdul, Amit Khandelwal, Nafisa Mandliwala, and Allison Ng contributed to the BetrFS 0.2 prototype.

REFERENCES

David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 1–14.

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

Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. And introduction to B$^{\varepsilon}$-trees and write-optimization. *Login; Magazine* 40, 5 (Oct. 2015).

Jeff Bonwick and B. Moore. 2005. ZFS: The Last Word in File Systems. Retrieved from http://opensolaris.org/os/community/zfs/docs/zfslast.pdf.

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

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

John Esmet, Michael A. Bender, Martin Farach-Colton, and B. C. Kuszmaul. 2012. The TokuFS streaming file system. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage (HotStorage'12)*.

Neeta Garimella. 2006. Understanding and Exploiting Snapshot Technology for Data Protection. Retrieved from http://www.ibm.com/developerworks/tivoli/library/t-snaptsm1/.

Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA.

Andromachi Hatzieleftheriou and Stergios V. Anastasiadis. 2011. Okeanos: Wasteless journaling for fast and reliable multistream storage. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'11)*. USENIX Association, Berkeley, CA, 19–19.

Val Henson, Amit Gud, Arjan van de Ven, and Zach Brown. 2006. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the 2nd Conference on Hot Topics in System Dependability (HotDep'06)*. USENIX Association, Berkeley, CA.

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. 2015a. BetrFS: Write-optimization in a kernel file system. *Transactions on Storage* 11, 4, Article 18 (Nov. 2015), 29 pages. DOI:http://dx.doi.org/10.1145/2798729

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. 2015b. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'15)*. 301–315.

Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. 2015. SpanFS: A scalable file system on fast storage devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'15)*. USENIX Association, Berkeley, CA, 249–261.

Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'15)*. 273–286.

Nicholas Lester, Alistair Moffat, and Justin Zobel. 2005. Fast on-line index construction by geometric partitioning. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM'05)*. ACM, New York, NY, 776–783. DOI:http://dx.doi.org/10.1145/1099554.1099739

Andrew W. Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L. Miller. 2009. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *Proceedings of the 7th Conference on File and Storage Technologies (FAST'09)*. USENIX Association, Berkeley, CA, 153–166.

Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 1–13.

David Lomet and Mark Tuttle. 1999. Logical logging to extend recovery to new domains. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*. ACM, New York, NY, 73–84.

Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Physical disentanglement in a container-based file system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 81–96.

Mike Mammarella, Shant Hovsepian, and Eddie Kohler. 2009. Modular data storage with anvil. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 147–160. DOI:http://dx.doi.org/10.1145/1629575.1629590

Avantika Mathur, MingMing Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*.

M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson. 2014. *The Design and Implementation of the FreeBSD Operating System*. Addison Wesley. 632–634 pages.

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

Soumyadeb Mitra, Marianne Winslett, and Windsor W. Hsu. 2008. Query-based partitioning of documents and indexes for information lifecycle management. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. ACM, New York, NY, 623–636. DOI:http://dx.doi.org/10.1145/1376616.1376680

Patrick O'Neil, Edward Cheng, Dieter Gawlic, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. DOI:http://dx.doi.org/10.1007/s002360050048

Kai Ren and Garth A Gibson. 2013. TABLEFS: Enhancing metadata efficiency in the local file system. In *USENIX Annual Technical Conference*. 145–156.

Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The linux B-tree filesystem. *Transactions on Storage* 9, 3, Article 9 (Aug. 2013), 32 pages. DOI:http://dx.doi.org/10.1145/2501620.2501623

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

Douglas Santry and Kaladhar Voruganti. 2014. Violet: A storage stack for IOPS/capacity bifurcated storage environments. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 13–24.

Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 217–228.

Pradeep Shetty, Richard P. Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'15)*. 17–30.

Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (ATEC'96)*. USENIX Association, Berkeley, CA, 1–14.

Tokutek. 2013. TokuDB: MySQL Performance, MariaDB Performance. http://www.tokutek.com/products/tokudb-for-mysql/. (2013).

Ben Verghese, Anoop Gupta, and Mendel Rosenblum. 1998. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII'98)*. ACM, New York, NY, 181–192. DOI:http://dx.doi.org/10.1145/291069.291044

Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. 2007. Argon: Performance insulation for shared storage servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*. USENIX Association, Berkeley, CA, 61–76. http://dl.acm.org/citation.cfm?id=1267903.1267908

Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, Berkeley, CA, 307–320. http://dl.acm.org/citation.cfm?id=1298455.1298485

Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. 2004. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*. IEEE Computer Society, Washington, DC, 4. DOI:http://dx.doi.org/10.1109/SC.2004.22

Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *Proceedings of the USENIX Annual Technical Conference*. 71–82.

Received October 2016; accepted December 2016