Optimizing Every Operation in a Write-optimized File System

Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, and Pooja Deo, *Stony Brook University;* 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, *Massachusetts Institute of Technology;* Donald E. Porter, *Stony Brook University*

https://www.usenix.org/conference/fast16/technical-sessions/presentation/yuan

This paper is included in the Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16).

February 22–25, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-28-7

Open access to the Proceedings of the 14th USENIX Conference on File and Storage Technologies is sponsored by USENIX

Optimizing Every Operation in a Write-Optimized File System

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

*Stony Brook University, *Facebook, **Two Sigma, †Rutgers University,*
and ‡Massachusetts Institute of Technology

Abstract

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 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.

1 Introduction

Write-Optimized Dictionaries (WODs)[1], such as Log-Structured Merge Trees (LSM-trees) [24] and B^ε-trees [6], 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 [7, 12, 30], metadata updates [7, 12, 25, 30], and recursive directory traversals [7, 12] 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 [7] and BetrFS [12] have slow file deletions, renames, and sequential file writes. Directory traversals in KVFS [30] and TableFS [25] 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.

This paper shows that a WOD-based file system can retain performance improvements to 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 [11, 12]. 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 application.

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 approach used by no-overwrite file systems, such as zfs [4] and btrfs [26], which writes data into free space only once. A particular challenge in adapting this technique to a B^ε-tree is balancing crash consistency of data against sufficient I/O scheduling flexibility to avoid reintroducing large, duplicate writes in B^ε-tree message flushing.

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. Fast traversals require

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

co-locating 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. 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. 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.

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 paper are:

- A **late-binding journal** for large writes to a message-oriented 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 trade-offs 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 1 GB 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^ε-trees and BetrFS. See Bender et al. [3] for a more comprehensive tutorial.

2.1 Write-Optimized Dictionaries

WODs include Log-Structured Merge Trees (LSM-trees) [24] and their variants [29, 30, 37], B^ε-trees [6], xDicts [5], and cache-oblivious lookahead arrays (COLAs) [2, 28]. 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 of 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^ε-trees, can perform point queries as fast as a B-tree. B^ε-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 MB of data and hence is bandwidth bound.

The key idea behind write optimization is deferring and batching small, random writes. A B^ε-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.

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 [32]. 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 relevant logging and cache-management features of TokuDB.

TokuDB updates B^ε-tree nodes using redirect on write [8]. 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 checkpoints of the tree. Between checkpoints, a write-ahead, logical log tracks

[2] TokuDB implements Fractal Tree indexes [2], a B^ε-tree variant.

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.

This scheme of checkpoint and write-ahead log allows the B^ε-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^ε-tree node fills up, the B^ε-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 at different times, any read applies all matching buffered messages between the root and leaf in commit order. 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^ε-trees [12]. BetrFS uses two B^ε-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^ε-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^ε-tree operations. 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^ε-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^ε-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 78–106MB/s—utilizing nearly all of the hard drive's 125 MB/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^ε-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^ε-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,

[3] TokuDB had a performance bug that further reduced BetrFS 0.1's sequential write performance to at most 1/3rd of disk bandwidth. See Section 6 for details.

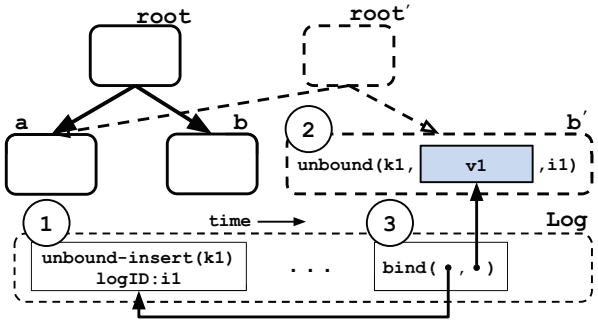


Figure 1: Late-binding journaling in a B$^\varepsilon$-tree.

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 [21]. 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 [26] 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.

BetrFS on-disk structures. The BetrFS B$^\varepsilon$-tree implementation writes B$^\varepsilon$-tree nodes to disk using redirect-on-write and maintains a logical write-ahead redo log. Each insert or delete message is first recorded in the log and then inserted into the tree's in-memory nodes. Each entry in the log specifies the operation (insert or delete) and the relevant keys and values.

Crash consistency is implemented by periodically checkpointing the B$^\varepsilon$-tree and by logging operations between checkpoints. An operation is durable once its log entry is on disk. At each checkpoint, all dirty nodes are written to ensure that a complete and consistent B$^\varepsilon$-tree snapshot is on disk, and the log is discarded. For instance, after checkpoint i completes, there is a single B$^\varepsilon$-tree, T_i, and an empty log. Any blocks that are not reachable in T_i can be garbage collected and reallocated.

Between checkpoints i and $i+1$, all operations are logged in Log_{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 replay Log_{i+1}.

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.
- A special *unbound message* is inserted into the B$^\varepsilon$-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 (non-leaf 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, 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 32 MB 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.

4 Balancing Search and Rename

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

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^ε-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 trade-off is common in file system design. Intermediate points between these extremes are possible, such as embedding inodes in directories but not moving data blocks of renamed files. Fast directory traversals require on-disk locality, whereas renames must issue only a small number of I/Os to be fast.

BetrFS 0.2's schema makes this trade-off parameterizable and tunable by partitioning the directory hierarchy into connected regions, which we call *zones*. Figure 2a 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 has 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.

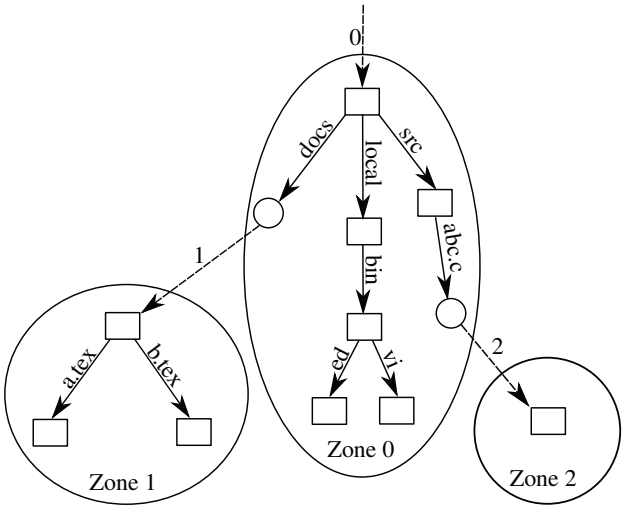
Directories and files within a zone are stored together, enabling fast scans within that zone. Crossing a zone boundary potentially requires a seek to a different part of the tree. Renaming a file under a zone root moves the data, whereas renaming a large file or directory (a zone root) requires only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

Zoning supports a spectrum of trade-off points between the two extremes described above. When zones are restricted to size 1, the BetrFS 0.2 schema is equivalent to an inode-based schema. If we set the zone size bound to infinity (∞), then BetrFS 0.2's schema is equivalent to BetrFS 0.1's schema. 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 MB between each seek, the dominant cost is transfer time, not seeking. Thus, one would expect the best zone size to be between tens of KB and a few MB. We also note that this trade-off 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 trade-offs.

As an effect of zoning, BetrFS 0.2 supports hard links by placing a file with more than 1 link into its own zone.

Metadata and data indexes. The BetrFS 0.2 meta-



(a) An example zone tree in BetrFS 0.2.

Metadata Index		
(0,"/")	→	stat info for "/"
(0,"/docs")	→	zone 1
(0,"/local")	→	stat info for "/local"
(0,"/src")	→	stat info for "/src"
(0,"/local/bin")	→	stat info for "/local/bin"
(0,"/local/bin/ed")	→	stat info for "/local/bin/ed"
(0,"/local/bin/vi")	→	stat info for "/local/bin/vi"
(0,"/src/abc.c")	→	zone 2
(1,"/")	→	stat info for "/docs"
(1,"/a.tex")	→	stat info for "/docs/a.tex"
(1,"/b.tex")	→	stat info for "/docs/b.tex"
(2,"/")	→	stat info 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"

(b) Example metadata and data indices in BetrFS 0.2.

Figure 2: Pictorial and schema illustrations of zone trees in BetrFS 0.2.

data index maps (zone-ID, relative-path) keys to metadata about a file or directory, as shown in Figure 2b. 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.

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 2b. 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 sub-directories in the order returned by `readdir` will effectively perform range queries on that zone and each of the zones beneath it.

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 2 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 trade-off 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 to record the number of data and metadata entries in its subtree.

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 trade-off 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, $C_0 = C_1 = 512$ KiB.

The zone schema enables BetrFS 0.2 to support a spectrum of trade-offs 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 §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 [12] 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 costs 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 number of rangecast messages in the node and k is the number of those messages relevant to the current query.

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^ε-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^ε-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 it also creates new opportunities for B^ε-tree-internal optimizations.

Leaf Pruning. When a B^ε-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 10 GB file 30–63 MB/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 [12]; 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.

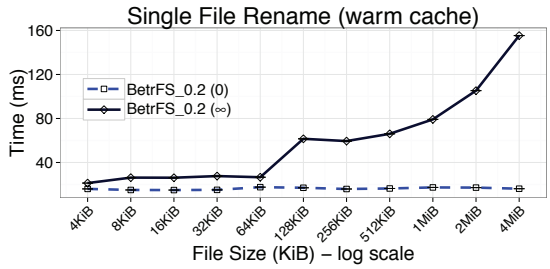
Second, the I/O to flush the BetrFS journal file was being amplified by the ext4 journal. Each BetrFS log flush appended to a file on ext4, which required updating the file size and allocation. BetrFS 0.2 reduces this overhead by pre-allocating space for the journal file and using `fdatasync`.

7 Evaluation

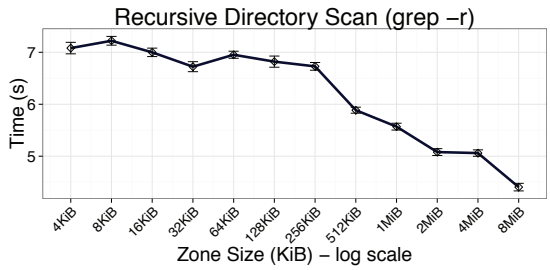
Our evaluation targets the following questions:

- 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 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 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, and a 500 GB, 7200 RPM ATA disk, with a 4096-byte block size. Each file system's block size is 4096 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 [12], btrfs [26], ext4 [20], XFS [31], and zfs [4]. We use the versions of XFS, btrfs, 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 2 partitions roughly 240 GB 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



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



(b) Recursive scans of the Linux 3.11.10 source for "cpu_to_be64" with different BetrFS 0.2 zone sizes.

Figure 3: The impact of zone size on rename and scan performance. Lower is better.

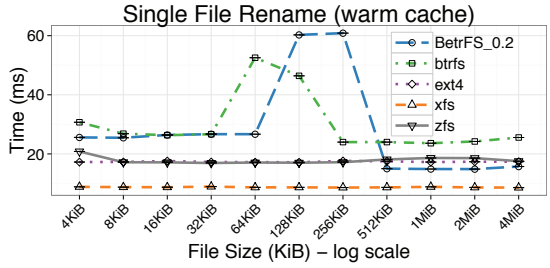


Figure 4: Time to rename single files. Lower is better.

experiment was run a minimum of 4 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 3a 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 3b, which shows `grep` performance versus zone size. As predicted in Section 4 directory-traversal performance improves as the zone size increases.

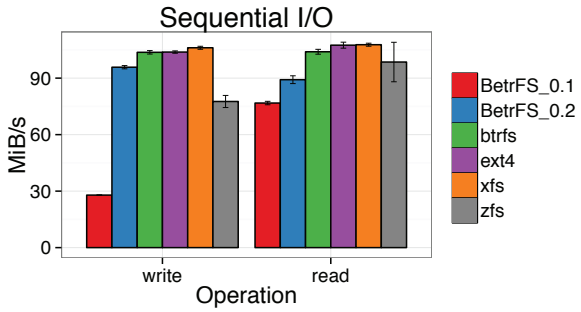


Figure 5: Large file I/O performance. We sequentially read and write a 10GiB file. Higher is better.

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 4 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 512 KiB 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 5 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 §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 1 shows the execution time of several common directory operations on the Linux 3.11.10

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

Table 1: 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	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

Table 2: Time to perform 10,000 4-byte overwrites on a 10 GiB file. Lower is better.

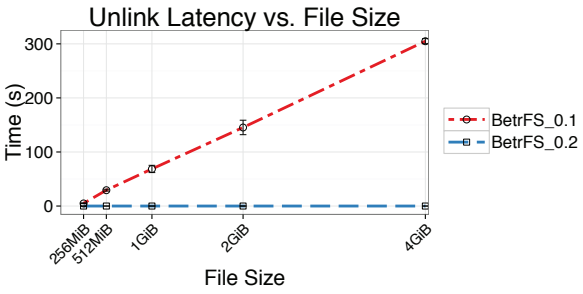


Figure 6: Unlink latency by file size. Lower is better.

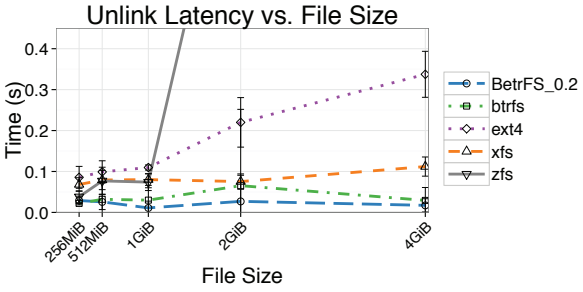


Figure 7: Unlink latency by file size. Lower is better.

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.

Unlink. Table 1 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 6, and we compare BetrFS 0.2 with other file systems in Figure 7. In BetrFS 0.1, the cost to delete a file scales linearly with the file size. Figure 7 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 unlink incurring amortized housekeeping work.

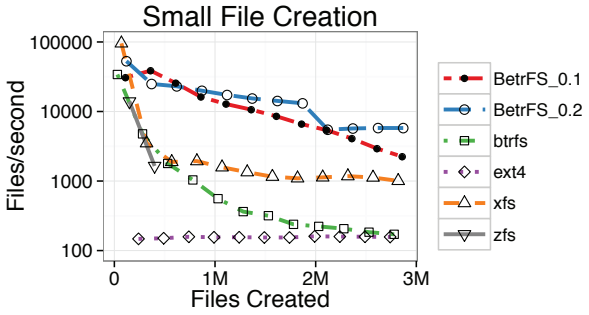


Figure 8: Sustained file creation for 3 million 200-byte files, using 4 threads. Higher is better, y-axis is log scale.

7.3 Maintaining the Best Cases

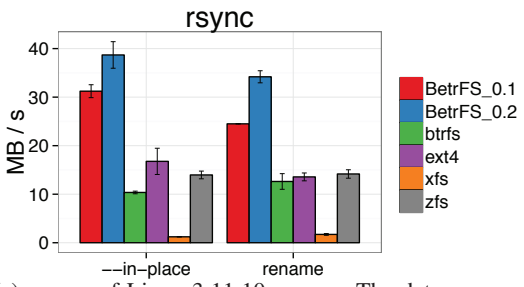
This subsection evaluates the best cases for a write-optimized file system, including small random writes, file creation, and searches. We confirm that our optimizations have not eroded the benefits of write-optimization. In most cases, there is no loss.

Small, random writes. Table 2 shows the execution time of a microbenchmark that issues 10,000 4-byte overwrites at random offsets within a 10GiB file, followed by an `fsync`. BetrFS 0.2 not only retains a two orders-of-magnitude improvement over the other file systems, but improves the latency over BetrFS 0.1 by 34%.

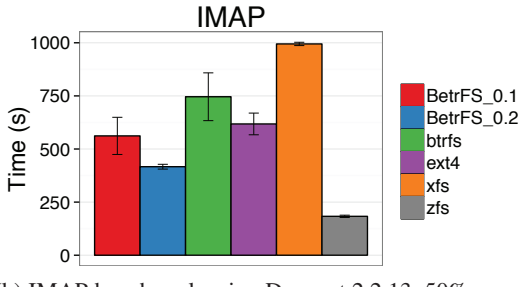
Small file creation. To evaluate file creation, we used the TokuBench benchmark [7] to create three million 200-byte files in a balanced directory tree with a fanout of 128. We used 4 threads, one per core of the machine.

Figure 8 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.

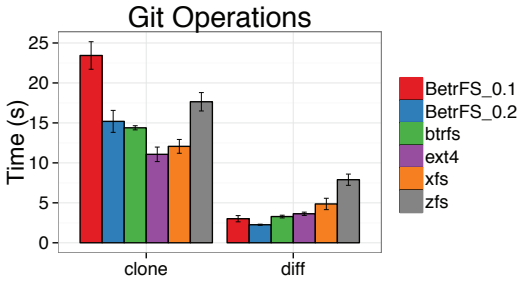
The line for BetrFS 0.2 is mostly higher than the line



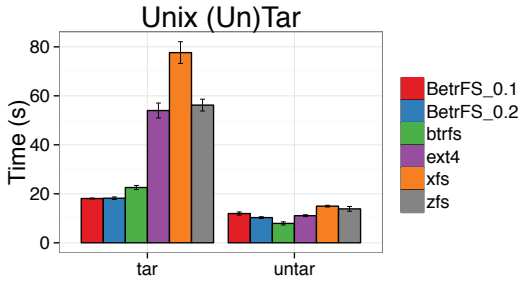
(a) `rsync` of Linux 3.11.10 source. The data source and destination are within the same partition and file system. Throughput in MB/s, higher is better.



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



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



(d) 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.

Figure 9: Application benchmarks

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 1 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 trade-offs to add zoning.

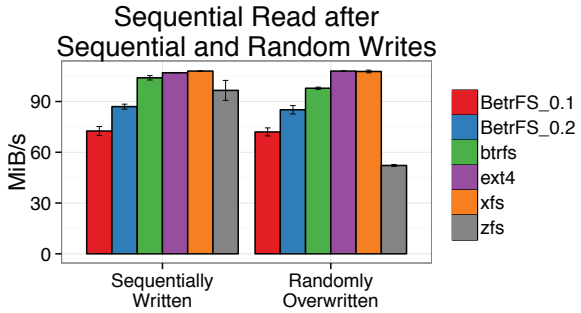


Figure 10: Sequential read throughput after sequentially writing a 10GiB file (left) and after partially overwriting 10,000 random blocks in the file (right). Higher is better.

7.4 Application Performance

This subsection evaluates the impact of the BetrFS 0.2 optimizations on the performance of several applications, shown in Figure 9. Figure 9a 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.

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 `fsyncs`, 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 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 10.

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 [35] is a technique designed for large-scale distributed systems, like Ceph [36], 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 [15] introduces a partitioning technique for multi-dimensional metadata indices based on KD-trees. Partitioning techniques have also been used to determine which data goes on slower versus faster media [23], to efficiently maintain inverted document indices [14], or to plug in different storage data structures to optimize for read- or write-intensive workloads [19]. Chunkfs [10] partitions the ext2 file system to improve recovery time. A number of systems also divide disk bandwidth and cache space for performance isolation [33, 34]; 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 trade-offs different from zoning's balance of directory searches and renames.

IceFS [18] 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.

Late-binding log entries. KVFS [30] 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 [1, 13, 16, 27] 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 [9] stores before- and after-images of individual database pages, which may be expensive for large updates or small updates to large objects. Logical logging [17] 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 [21]. We adapt this technique to implement late-binding journaling of large messages (or large groups of related small messages) in BetrFS 0.2.

Previous systems have implemented variations of soft updates [22], 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 paper 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 offer 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. 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.

References

[1] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2009), pp. 1–14.

[2] BENDER, M. A., FARACH-COLTON, M., FINEMAN, J. T., FOGEL, Y. R., KUSZMAUL, B. C., AND NELSON, J. Cache-oblivious streaming B-trees. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)* (2007), pp. 81–92.

[3] BENDER, M. A., FARACH-COLTON, M., JANNEN, W., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., YUAN, J., AND ZHAN, Y. An introduction to B^e-trees and write-optimization. *:login; Magazine 40*, 5 (Oct 2015), 22–28.

[4] BONWICK, J., AND MOORE, B. ZFS: The Last Word in File Systems. `http://opensolaris.org/os/community/zfs/docs/zfslast.pdf`.

[5] BRODAL, G. S., DEMAINE, E. D., FINEMAN, J. T., IACONO, J., LANGERMAN, S., AND MUNRO, J. I. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2010), pp. 1448–1456.

[6] BRODAL, G. S., AND FAGERBERG, R. Lower bounds for external memory dictionaries. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2003), pp. 546–554.

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

[8] GARIMELLA, N. Understanding and exploiting snapshot technology for data protection. `http://www.ibm.com/developerworks/tivoli/library/t-snaptsm1/`, Apr. 2006.

[9] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[10] HENSON, V., VAN DE VEN, A., GUD, A., AND BROWN, Z. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the USENIX Conference on Hot Topics in System Dependability (HotDep)* (2006).

[11] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 301–315.

[12] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. BetrFS: Write-optimization in a kernel file system. *ACM Transactions on Storage (TOS) 11*, 4 (Oct. 2015), 18:1–18:29.

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

[14] LESTER, N., MOFFAT, A., AND ZOBEL, J. Fast on-line index construction by geometric partitioning. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)* (2005), pp. 776–783.

[15] LEUNG, A. W., SHAO, M., BISSON, T., PASUPATHY, S., AND MILLER, E. L. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2009), pp. 153–166.

[16] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2011), pp. 1–13.

[17] LOMET, D., AND TUTTLE, M. Logical logging to extend recovery to new domains. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (1999), pp. 73–84.

[18] LU, L., ZHANG, Y., DO, T., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Physical disentanglement in a container-based file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014), pp. 81–96.

[19] MAMMARELLA, M., HOVSEPIAN, S., AND KOHLER, E. Modular data storage with Anvil. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2009), pp. 147–160.

[20] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: Current status and future plans. In *Linux Symposium* (2007).

[21] MCKUSICK, M., NEVILLE-NEIL, G., AND WATSON, R. *The Design and Implementation of the FreeBSD Operating System*. Addison Wesley, 2014.

[22] MCKUSICK, M. K., AND GANGER, G. R. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the USENIX Annual Technical Conference* (1999), pp. 1–17.

[23] MITRA, S., WINSLETT, M., AND HSU, W. W. Query-based partitioning of documents and indexes for information lifecycle management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2008), pp. 623–636.

[24] O'NEIL, P., CHENG, E., GAWLIC, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[25] REN, K., AND GIBSON, G. A. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the USENIX Annual Technical Conference* (2013), pp. 145–156.

[26] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (Aug. 2013), 9:1–9:32.

[27] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (Feb. 1992), 26–52.

[28] SANTRY, D., AND VORUGANTI, K. Violet: A storage stack for IOPS/capacity bifurcated storage environments. In *Proceedings of the USENIX Annual Technical Conference* (2014), pp. 13–24.

[29] SEARS, R., AND RAMAKRISHNAN, R. bLSM: a general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2012), pp. 217–228.

[30] SHETTY, P., SPILLANE, R. P., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with VT-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2013), pp. 17–30.

[31] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference* (1996), pp. 1–14.

[32] TOKUTEK, INC. TokuDB: MySQL Performance, MariaDB Performance. `http://www.tokutek.com/products/tokudb-for-mysql/`, 2013.

[33] VERGHESE, B., GUPTA, A., AND ROSENBLUM, M. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1998), pp. 181–192.

[34] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance insulation for shared storage servers. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2007), pp. 5–5.

[35] WEIL, S., POLLACK, K., BRANDT, S. A., AND MILLER, E. L. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)* (Nov. 2004).

[36] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 307–320.

[37] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *Proceedings of the USENIX Annual Technical Conference* (2015), pp. 71–82.