B*e*trFS: A Right-Optimized Write-Optimized File System

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

*Stony Brook University, *Tokutek Inc., †Rutgers University, and ‡Massachusetts Institute of Technology*

Abstract

The ***B$^\varepsilon$-tree File System***, or B*e*trFS, (pronounced "better eff ess") is the first in-kernel file system to use a write-optimized index. Write optimized indexes (WOIs) are promising building blocks for storage systems because of their potential to implement both microwrites and large scans efficiently.

Previous work on WOI-based file systems has shown promise but has also been hampered by several open problems, which this paper addresses. For example, FUSE issues many queries into the file system, superimposing read-intensive workloads on top of write-intensive ones, thereby reducing the effectiveness of WOIs. Moving to an in-kernel implementation can address this problem by providing finer control of reads. This paper also contributes several implementation techniques to leverage kernel infrastructure without throttling write performance.

Our results show that B*e*trFS provides good performance for both arbitrary microdata operations, which include creating small files, updating metadata, and small writes into large or small files, and for large sequential I/O. On one microdata benchmark, B*e*trFS provides more than 4× the performance of `ext4` or `XFS`. B*e*trFS is an ongoing prototype effort, and requires additional data-structure tuning to match current general-purpose file systems on some operations such as deletes, directory renames, and large sequential writes. Nonetheless, many applications realize significant performance improvements. For instance, an in-place `rsync` of the Linux kernel source realizes roughly 1.6–22× speedup over other commodity file systems.

1 Introduction

Today's applications exhibit widely varying I/O patterns, making performance tuning of a general-purpose file system a frustrating balancing act. Some software, such as virus scans and backups, demand large, sequential scans of data. Other software requires many small writes (microwrites). Examples include email delivery, creating lock files for an editing application, making small updates to a large file, or updating a file's `atime`. The underlying problem is that many standard data structures in the file-system designer's toolbox optimize for one case at the expense of another.

Recent advances in write-optimized indexes (WOI) [4, 8–10, 23, 27, 28] are exciting because they have the potential to implement both efficient microwrites and large scans. The key strength of the best WOIs is that they can ingest data up to two orders of magnitude faster than B-trees while matching or improving on the B-tree's point- and range-query performance [4, 9].

WOIs have been successful in commercial key-value stores and databases [2, 3, 12, 17, 20, 33, 34], and previous research on WOIs in file systems has shown promise [15, 25, 31]. However, progress towards a production-quality write-optimized file system has been hampered by several open challenges, which we address in this paper:

- **Code complexity.** A production-quality WOI can easily be 50,000 lines of complex code, which is difficult to shoehorn into an OS kernel. Previous WOI file systems have been implemented in user space.
- **FUSE squanders microwrite performance.** FUSE [16] issues a query to the underlying file system before almost every update, superimposing search-intensive workloads on top of write-intensive workloads. Although WOIs are no worse for point queries than any other sensible data structure, writes are much faster than reads, and injecting needless point queries can nullify the advantages of write optimization.
- **Mapping file system abstractions onto a WOI.** We cannot realize the full potential performance benefits of write-optimization by simply dropping in a WOI as a replacement for a B-tree. The schema and use of kernel infrastructure must exploit the performance advantages of the new data structure.

This paper describes the ***B$^\varepsilon$-tree File System***, or B*e*trFS, the first in-kernel file system designed to take

full advantage of write optimization. Specifically, B*e*trFS is built using the mature, well-engineered B^ε-tree implementation from Tokutek's Fractal Tree index, called TokuDB [33]. Our design is tailored to the performance characteristics of Fractal Tree indexes, but otherwise uses them as a black-box key-value store, so many of our design decisions may be applicable to other write-optimized file systems.

Experiments show that B*e*trFS can give up to an order-of-magnitude improvement to the performance of file creation, random writes to large files, recursive directory traversals (such as occur in `find`, recursive `grep`s, backups, and virus scans), and meta-data updates (such as updating file `atime` each time a file is read).

The contributions of this paper are:

- The design and implementation of an in-kernel, write-optimized file system.
- A schema, which ensures both locality and fast writes, for mapping VFS operations to a write-optimized index.
- A design that uses unmodified OS kernel infrastructure, designed for traditional file systems, yet minimizes the impact on write optimization. For instance, B*e*trFS uses the OS kernel cache to accelerate reads without throttling writes smaller than a disk sector.
- A thorough evaluation of the performance of the B*e*trFS prototype. For instance, our results show almost two orders of magnitude improvement on a small-write microbenchmark and a 1.5× speedup on applications such as `rsync`

Our results suggest that a well-designed B^ε-tree-based file system can match or outperform traditional file systems on almost every operation, some by an order of magnitude. Comparisons with state-of-the-art file systems, such as `ext4`, `XFS`, `zfs`, and `btrfs` support this claim. We believe that the few slower operations are not fundamental to WOIs, but can be addressed with a combination of algorithmic advances and engineering effort in future work.

2 Motivation and Background

This section summarizes background on the increasing importance of microwrites, explains how WOIs work, and summarizes previous WOI-based file systems.

2.1 The microwrite problem

A microwrite is a write operation where the setup time (i.e. seek time on a conventional disk) exceeds the data-transfer time. Conventional file-system data structures force file-system designers to choose between optimizing for efficient microwrites and efficient scans.

Update-in-place file systems [11, 32] optimize for scans by keeping related items, such as entries in a directory or consecutive blocks in a file, near each other. However, since items are updated in place, update performance is often limited by the random-write latency of the underlying disk.

B-tree-based file systems store related items logically adjacent in the B-tree, but B-trees do not guarantee that logically-adjacent items will be physically adjacent. As a B-tree *ages*, leaves become scattered across the disk due to node splits from insertions and node merges from deletions. In an aged B-tree, there is little correlation between the logical and physical order of the leaves, and the cost of reading a new leaf involves both the data-transfer cost and the seek cost. If leaves are too small to amortize the seek costs, then range queries can be slow. The seek costs can be amortized by using larger leaves, but this further throttles update performance.

At the other extreme, logging file systems [5, 7, 26, 29, 30] optimize for writes. Logging ensures that files are created and updated rapidly, but the resulting data and metadata can be spread throughout the log, leading to poor performance when reading data or metadata from disk. These performance problems are particularly noticeable in large scans (recursive directory traversals and backups) that cannot be accelerated by caching.

The microwrite bottleneck creates problems for a range of applications. HPC checkpointing systems generate so many microwrites that a custom file system, PLFS, was designed to efficiently handle them [5] by exploiting the specifics of the checkpointing workload. Email servers often struggle to manage large sets of small messages and metadata about those messages, such as the `read` flag. Desktop environments store preferences and active state in a key-value store (i.e., a registry) so that accessing and updating keys will not require file-system-level microdata operations. Unix and Linux system administrators commonly report 10–20% performance improvements by disabling the `atime` option [14]; maintaining the correct `atime` behavior induces a heavy microwrite load, but some applications require accurate `atime` values.

Microwrites cause performance problems even when the storage system uses SSDs. In a B-tree-based file system, small writes trigger larger writes of entire B-tree nodes, which can further be amplified to an entire erase block on the SSD. In a log-structured file system, microwrites can induce heavy cleaning activity, especially when the disk is nearly full. In either case, the extra write activity reduces the lifespan of SSDs and can limit performance by wasting bandwidth.

2.2 Write-Optimized Indexes

In this subsection, we review write-optimized indexes and their impact on performance. Specifically, we describe the B^ε-tree [9] and why we have selected this data structure for BεtrFS. The best WOI can dominate B-trees in performance rather than representing a different tradeoff choice between reads and writes.

B^ε-trees. A B^ε-tree is a B-tree, augmented with per-node buffers. New items are inserted in the buffer of the root node of a B^ε-tree. When a node's buffer becomes full, messages are moved from that node's buffer to one of its children's buffers. The leaves of the B^ε-tree store key-value pairs, as in a B-tree. Point and range queries behave similarly to a B-tree, except that each buffer on the path from root to leaf must also be checked for items that affect the query.

B^ε-trees are asymptotically faster than B-trees, as summarized in Table 1. To see why, consider a B-tree with N items and in which each node can hold B keys. (For simplicity, assume keys have constant size and that the data associated with each key has negligiable size.) The tree will have fanout B, so its height will be $O(\log_B N)$. Inserts and lookups will therefore require $O(\log_B N)$ I/Os. A range query that returns k items will require $O(\log_B N + k/B)$ I/Os.

For comparison, a B^ε-tree with nodes of size B will have B^ε children, where $0 < \varepsilon \leq 1$. Each node will store one "pivot key" for each child, consuming B^ε space per node. The remaining $B - B^\varepsilon$ space in each node will be used to buffer newly inserted items. Since the fanout of the tree is B^ε, its height is $O(\log_{B^\varepsilon} N) = O(\frac{1}{\varepsilon}\log_B N)$. Consequently, searches will be slower by a factor of $\frac{1}{\varepsilon}$. However, each time a node flushes items to one of its children, it will move at least $(B - B^\varepsilon)/B^\varepsilon \approx B^{1-\varepsilon}$ elements. Since each element must be flushed $O(\frac{1}{\varepsilon}\log_B N)$ times to reach a leaf, the amortized cost of inserting N items is $O(\frac{1}{\varepsilon B^{1-\varepsilon}}\log_B N)$. Range queries returning k items require $O(\frac{1}{\varepsilon}\log_B N + k/B)$ I/Os. If we pick, for example, $\varepsilon = 1/2$, the point and range query costs of a B^ε-tree become $O(\log_B N)$ and $O(\log_B N + k/B)$, which are the same as a B-tree, but the insert cost becomes $O(\log_B N/\sqrt{B})$, which is faster by a factor of $\sqrt{B}$.

In practice, however, B^ε-trees use much larger nodes than B-trees. For example, a typical B-tree might use 4KB or 64KB nodes, compared to 4MB nodes in Tokutek's Fractal Tree indexes. B-trees must use small nodes because a node must be completely re-written each time a new item is added to the database, unlike in B^ε-trees, where writes are batched. Large nodes mean that, in practice, the height of a B^ε-tree is not much larger than the height of a B-tree on the same data. Thus, the performance of point queries in a B^ε-tree implementation can be comparable to point query performance in a B-tree. Large nodes also speed up range queries, since the data is spread over fewer nodes, requiring fewer disk seeks to read in all the data.

To get a feeling for what this speedup looks like, consider the following example. Suppose a key-value store holds 1TB of data, with 128-byte keys and records (key+value) of size 1KB. Suppose that data is logged for durability, and periodically all updates in the log are applied to the main tree in batch.

In the case of a B-tree with 4KB nodes, the fanout of the tree will be 4KB/128B$=32$. Thus the non-leaf nodes can comfortably fit into the memory of a typical server with 64GB of RAM, but only a negligible fraction of the 1TB of leaves will be cached at any given time. During a random insertion workload, most updates in a batch will require exactly 2 I/Os: 1 I/O to read in the target leaf and 1 I/O to write it back to disk after updating its contents.

For comparison, suppose a B^ε-tree has branching factor of 10 and nodes of size 1MB. Once again, all but the last level fit in memory. When a batch of logged updates is applied to the tree, they are simply stored in the tree's root buffer. Since the root is cached, this requires a single I/O. When an internal node becomes full and flushes its buffer to a non-leaf child, this causes two writes: an update of the parent and an update of the child. There are no reads required since both nodes are cached. When an internal node flushes its buffer to a leaf node, this requires one additional read to load the leaf into memory.

There are 1TB/1MB=2^{20} leaves, so since the tree has fanout 10, its height is $1 + \log_{10} 2^{20} \approx 7$. Each item is therefore involved in 14 I/Os: it is written and read once at each level.

However, each flush moves 1MB/10 = 100kB of data, in other words, 100 items. Thus, the average per-item cost of flushing an item to a leaf is $14/100$. Since a B-tree would require 2 I/Os for each item, the B^ε-tree is able to insert data $2/(14/100) = 14.3$ times faster than a B-tree. As key-value pairs get smaller, say for metadata updates, this speedup factor grows.

In both cases, a point query requires a single I/O to read the corresponding leaf for the queried key. Range queries can be much faster, as the B^ε-tree seeks only once every 1MB vs once every 4KB in the B-tree.

Because buffered messages are variable length in our implementation, even with fixed-size nodes, B is not constant. Rather than fix ε, our implementation bounds the range of pivots per node (B^ε) between 4 and 16.

Upserts. B^ε-trees support "upserts," an efficient method for updating a key-value pair in the tree. When an application wants to update the value associated with key k in the B^ε-tree, it inserts a message $(k, (f, \Delta))$ into the tree, where f specifies a call-back function that can be used to apply the change specified by Δ to the old value

associated with key k. This message is inserted into the tree like any other piece of data. However, every time the message is flushed from one node to a child C, the B^ε-tree checks whether C's buffer contains the old value v associated with key k. If it does, then it replaces v with $f(v,\Delta)$ and discards the upsert message from the tree. If an application queries for k before the callback function is applied, then the B^ε-tree computes $f(v,\Delta)$ on the fly while answering the query. This query is efficient because an upsert for key k always lies on the path from the root of the B^ε-tree to the leaf containing k. Thus, the upsert mechanism can speed up updates by one to two orders of magnitude without slowing down queries.

Log-structured merge trees. Log-structured merge trees (LSM trees) [23] are a WOI with many variants [28]. They typically consist of a logarithmic number of indexes (e.g., B-trees) of exponentially increasing size. Once an index at one level fills up, it is emptied by merging it into the index at the next larger level.

LSM trees can be tuned to have the same insertion complexity as a B^ε-tree, but queries in a naïvely implemented LSM tree can be slow, as shown in Table 1. Implementers have developed methods to improve the query performance, most notably using Bloom filters [6] for each B-tree. A point query for an element in the data structure is typically reported as improving to $O(\log_B N)$, thus matching a B-tree. Bloom filters are used in most LSM tree implementations (e.g., [3, 12, 17, 20]).

Bloom filters do not help in range queries, since the successor of any key may be in any level. In addition, the utility of Bloom filters degrades with the use of upserts, and upserts are key to the performance of B*e*trFS. To see why, note that in order to compute the result of a query, all relevant upserts must be applied to the key-value pair. If there are many upserts "in flight" at different levels of the LSM tree, then searches will need to be performed on each such level. Bloom filters can be helpful to direct us to the levels of interest, but that does not obviate the need for many searches, and since the leaves of the different LSM tree B-trees might require fetching, the search performance can degrade.

B*e*trFS uses B^ε-trees, as implemented in Tokutek's Fractal Tree indexes, because Fractal Tree indexes are the only WOI implementation that matches the query performance of B-trees for all workloads, including the upsert-intensive workloads generated by B*e*trFS. In short, LSMs match B-tree query times in special cases, and B^ε-trees match B-tree query times in general.

Data Struct.	Insert	Point Query no Upserts	Point Query w/ Upserts	Range Query
B-tree	$\log_B N$	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$
LSM	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{\log_B^2 N}{\varepsilon}$	$\frac{\log_B^2 N}{\varepsilon}$	$\frac{\log_B^2 N}{\varepsilon} + \frac{k}{B}$
LSM+BF	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\log_B N$	$\frac{\log_B^2 N}{\varepsilon}$	$\frac{\log_B^2 N}{\varepsilon} + \frac{k}{B}$
B^ε-tree	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{\log_B N}{\varepsilon}$	$\frac{\log_B N}{\varepsilon}$	$\frac{\log_B N}{\varepsilon} + \frac{k}{B}$

Table 1: Asymptotic I/O costs of important operations in B-trees and several different WOIs. Fractal Tree indexes simultaneously support efficient inserts, point queries (even in the presence of upserts), and range queries.

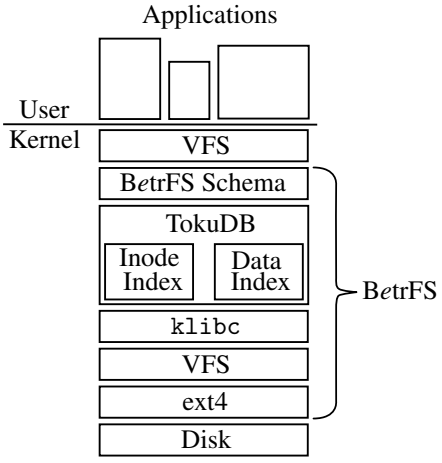


Figure 1: The B*e*trFS architecture.

3 B*e*trFS Design

B*e*trFS is an in-kernel file system designed to take full advantage of the performance strengths of B^ε-trees. The overall system architecture is illustrated in Figure 1.

The B*e*trFS VFS schema transforms file-system operations into efficient B^ε-tree operations whenever possible. The keys to obtaining good performance from B^ε-trees are (1) to use upsert operations to update file system state and (2) to organize data so that file-system scans can be implemented as range queries in the B^ε-trees. We describe how our schema achieves these goals in Section 4.

By implementing B*e*trFS as an in-kernel file system, we avoid the performance overheads of FUSE, which can be particularly deleterious for a write-optimized file system. We also expose opportunities for optimizing our file system's interaction with the kernel's page cache.

B*e*trFS's stacked file-system design cleanly separates the complex task of implementing write-optimized indexes from block allocation and free-space management. Our kernel port of TokuDB stores data on an underlying `ext4` file system, but any file system should suffice.

Porting a 45KLoC database into the kernel is a non-trivial task. We ported TokuDB into the kernel by writing a shim layer, which we call `klibc`, that translates the TokuDB external dependencies into kernel functions for locking, memory allocation, and file I/O. Section 6

```
/home
/home/alice
/home/bob
/home/alice/betrfs.pdf
/home/alice/betrfs.tex
/home/bob/betrfs.c
/home/bob/betrfs.ko
```

Figure 2: Example of the sort order used in B*e*trFS.

describes `klibc` and summarizes our experiences and lessons learned from the project.

4 The B*e*trFS File-System Schema

B^ε-trees implement a key-value store, so B*e*trFS must translate file-system operations into key-value operations. This section presents the B*e*trFS schema for performing this translation and explains how this schema takes advantage of the performance strengths of B^ε-trees.

4.1 B*e*trFS Data and Metadata Indexes

B*e*trFS stores file system data and metadata using two indexes in the underlying database: a metadata index and a data index. Since both keys and values may be variable-sized, B*e*trFS is able to pack many index entries into each B^ε-tree node.

The metadata index. The B*e*trFS prototype maintains an index mapping full pathnames (relative to the mount point) to file metadata (roughly equivalent to the contents of `struct stat`):

$$\text{path} \rightarrow (\text{size}, \text{owner}, \text{timestamps}, \text{etc}\ldots)$$

The metadata index is designed to support efficient file creations, deletions, lookups, and directory scans. The index sorts paths first by the number of slashes, then lexicographically. Thus, items within the same directory are stored consecutively as illustrated in Figure 2. With this ordering, scanning a directory, either recursively or not, can be implemented as a range query.

The data index. Though keys and values may be variable-sized, the B*e*trFS prototype breaks files into 4096-byte blocks for better integration with the page cache. Thus, the data index maps (file, offset) tuples to blocks:

$$(\text{path}, \text{block-number}) \rightarrow \text{data}[4096]$$

Keys in the data index are also sorted lexicographically, which guarantees that the contents of a file are logically adjacent and therefore, as explained in Subsection 2.2, almost always physically adjacent on disk. This enables file contents to be read sequentially at near disk bandwidth. B*e*trFS implements sparse files by simply omitting the sparse blocks from the data index.

FS Operation	B^ε-tree Operation
Mkdir	Upsert
Rmdir	Upsert
Create	Upsert
Unlink	Upsert + Delete data blocks
Truncate	Upsert + Delete data blocks
Setattr (e.g. `chmod`)	Upsert
Rename	Copy files
Symlink	Upsert
Lookup (i.e. `lstat`)	Point Query
Readlink	Point Query
Readdir	Range Query
File write	Upsert
File read	Range Query
MMap readpage(s)	Point/Range Query
MMap writepage(s)	Upsert(s)

Table 2: B*e*trFS implementation strategies for basic file-system operations. Almost all operations are implemented using efficient upserts, point queries, or range queries. Unlink, Truncate, and Rename currently scale with file and/or directory sizes.

B*e*trFS uses variable-sized values to avoid zero-padding the last block of each file. This optimization avoids the CPU overhead of zeroing out unused regions of a buffer, and then compressing the zeros away before writing a node to disk. For small-file benchmarks, this optimization yielded a significant reduction in overheads. For instance, this optimization improves throughput on TokuBench (§7) by 50–70%.

4.2 Implementing B*e*trFS Operations

Favoring blind writes. A latent assumption in much file system code is that data must be written at disk-sector granularity. As a result, a small write must first bring the surrounding disk block into the cache, modify the block, and then write it back. This pattern is reflected in the Linux page cache helper function `__block_write_begin()`. B*e*trFS avoids this read-modify-write pattern, instead issuing blind writes—writes without reads—whenever possible.

Reading and writing files in B*e*trFS. B*e*trFS implements file reads using range queries in the data index. B^ε-trees can load the results of a large range query from disk at effectively disk bandwidth.

B*e*trFS supports efficient file writes of any size via upserts and inserts. Application writes smaller than one 4K block become messages of the form:

$$\text{UPSERT}(\text{WRITE}, (\mathit{path}, n), \mathit{offset}, v, \ell),$$

which means the application wrote ℓ bytes of data, v, at

the given *offset* into block *n* of the file specified by *path*. Upsert messages completely encapsulate a block modification, obviating the need for read-modify-write. Writes of an entire block are implemented with an insert (also called a put), which is a blind replacement of the block, and behaves similarly.

As explained in Subsection 2.2, upserts and inserts are messages inserted into the root node, which percolate down the tree. By using upserts for file writes, a write-optimized file system can aggregate many small random writes into a single large write to disk. Thus, the data can be committed to disk by performing a single seek and one large write, yielding an order-of-magnitude boost in performance for small random writes.

In the case of large writes spanning multiple blocks, inserts follow a similar path of copying data from the root to the leaves. The B^ε-tree implementation has some optimizations for large writes to skip intermediate nodes on the path to a leaf, but they are not aggressive enough to achieve full disk bandwidth for large sequential file writes. We leave this issue for future work, as a solution must carefully address several subtle issues related to pending messages and splitting leaf nodes.

File-system metadata operations in B*e*trFS. As shown in Table 2, B*e*trFS also converts almost all metadata updates, such timestamp changes, file creation and symbolic linking, into upserts.

The only metadata updates that are not upserts in B*e*trFS are unlink, truncate, and rename. We now explain the obstacles to implementing these operations as upserts, which we leave for future work.

Unlink and truncate can both remove blocks from a file. B*e*trFS performs this operation in the simplest possible way: it performs a range query on the blocks that are to be deleted, and issues a TokuDB delete for each such block in the data index. Although TokuDB delete operations are implemented using upserts, issuing $O(n)$ upserts can make this an expensive task.

Second, keying by full path makes recursive directory traversals efficient, but makes it non-trivial to implement efficient renames. For example, our current implementation renames files and directories by re-inserting all the key-value pairs under their new keys and then deleting the old keys, effectively performing a deep copy of the file or directory being renamed. One simple solution is to add an inode-style layer of indirection, with a third index. This approach is well-understood, and can sacrifice some read locality as the tree ages. We believe that data-structure-level optimizations can improve the performance of rename, which we leave for future work.

Although the schema described above can use upserts to make most changes to the file system, many POSIX file system functions specify preconditions that the OS must check before changing the file system. For example, when creating a file, POSIX requires the OS to check that the file doesn't already exist and that the user has write permission on the containing directory. In our experiments, the OS cache of file and directory information was able to answer these queries, enabling file creation etc., to run at the full speed of upserts.

Crash consistency. We use the TokuDB transaction mechanism for crash consistency. TokuDB transactions are equivalent to full data journaling, with all data and metadata updates logged to a file in the underlying `ext4` file system. Log entries are retired in-order, and no updates are applied to the tree on disk ahead of the TokuDB logging mechanism. Entries are appended to one of two in-memory log buffers (16 MB by default). These buffers are rotated and flushed to disk every second or when a buffer overflows.

Although exposing a transactional API may be possible, B*e*trFS currently uses transactions only as an internal consistency mechanism. B*e*trFS generally uses a single transaction per system call, except for writing data, which uses a transaction per data block. In our current implementation, transactions on metadata execute while holding appropriate VFS-level mutex locks, making transaction conflicts and aborts vanishingly rare.

Compression. Compression is important to performance, especially for keys. Both indexes use full paths as keys, which can be long and repetitive, but TokuDB's compression mitigates these overheads. Using `quicklz` [24], the sorted path names in our experiments compress by a factor of 20, making the disk-space overhead manageable.

The use of data compression also means that there isn't a one-to-one correspondence between reading a file-system-level block and reading a block from disk. A leaf node is typically 4 MB, and compression can pack more than 64 file system blocks into a leaf. In our experience with large data reads and writes, data compression can yield a boost to file system throughput, up to 20% over disabling compression.

5 Write-Optimization in System Design

In designing B*e*trFS, we set the goal of working within the existing Linux VFS framework. An underlying challenge is that, at points, the supporting code assumes that reads are as expensive as writes, and necessary for update-in-place. The use of write optimization violates these assumptions, as sub-block writes can be faster than a read. This section explains several strategies we found for improving the B*e*trFS performance while retaining Linux's supporting infrastructure.

5.1 Eternal Sunshine of the Spotless Cache

B*e*trFS leverages the Linux page cache to implement efficient small reads, avoid disk reads in general, and facilitate memory-mapped files. By default, when an application writes to a page that is currently in cache, Linux marks it as dirty and writes it out later. This way, several application-level writes to a page can be absorbed in the cache, requiring only a single write to disk. In B*e*trFS, however, small writes are so cheap that this optimization does not always make sense.

In B*e*trFS, the `write` system call never dirties a clean page in the cache. When an application writes to a clean cached page, B*e*trFS issues an upsert to the B^{ε}-tree and applies the write to the cached copy of that page. Thus the contents of the cache are still in sync with the on-disk data, and the cached page remains clean.

Note that B*e*trFS' approach is not always better than absorbing writes in the cache and writing back the entire block. For example, if an application performs hundreds of small writes to the same block, then it would be more efficient to mark the page dirty and wait until the application is done to write the final contents back to disk. A production version of B*e*trFS should include heuristics to detect this case. We found that performance in our prototype was good enough without this optimization, though, so we have not yet implemented it.

The only situation where a page in the cache is dirtied is when the file is memory-mapped for writing. The memory management hardware does not support fine-grained tracking of writes to memory-mapped files — the OS knows only that something within in the page of memory has been modified. Therefore, B*e*trFS' `mmap` implementation uses the default read and write page mechanisms, which operate at page granularity.

Our design keeps the page cache coherent with disk. We leverage the page cacahe for faster warm-cache reads, but avoid unnecessary full-page writebacks.

5.2 FUSE is Write De-Optimized

We implemented B*e*trFS as an in-kernel file system because the FUSE architecture contains several design decisions that can ruin the potential performance benefits of a write-optimized file system. FUSE has well-known overheads from the additional context switches and data marshalling it performs when communicating with user-space file systems. However, FUSE is particularly damaging to write-optimized file systems for completely different reasons.

FUSE can transform write-intensive into read-intensive workloads because it issues queries to the user-space file system before (and, in fact, after) most file system updates. For example, FUSE issues `GETATTR` calls (analogous to calling `stat()`) for the entire path of a file lookup, every time the file is looked up by an application. For most in-kernel file systems, subsequent lookups could be handled by the kernel's directory cache, but FUSE conservatively assumes that the underlying file system can change asynchronously (which can be true, e.g. in network file systems).

These searches can choke a write-optimized data structure, where insertions are two orders of magnitude faster than searches. The TokuFS authors explicitly cite these searches as the cause of the disappointing performance of their FUSE implementation [15].

The TableFS authors identified another source of FUSE overhead: double caching of inode information in the kernel [25]. This reduces the cache's effective hit rate. For slow file systems, the overhead of a few extra cache misses may not be significant. For a write-optimized data structure working on a write-intensive workload, the overhead of the cache misses can be substantial.

5.3 Ext4 as a Block Manager

Since TokuDB stores data in compressed nodes, which can have variable size, TokuDB relies on an underlying file system to act as a block and free space manager for the disk. Conventional file systems do a good job of storing blocks of large files adjacently on disk, especially when writes to the file are performed in large chunks.

Rather than reinvent the wheel, we stick with this design in our kernel port of TokuDB. B*e*trFS represents tree nodes as blocks within one or more large files on the underlying file system, which in our prototype is unmodified `ext4` with ordered data mode and direct IO. We rely on `ext4` to correctly issue barriers to the disk write cache, although disabling the disk's write cache did not significantly impact performance of our workloads. In other words, all B*e*trFS file system updates, data or metadata, generally appear as data writes, and an `fsync` to the underlying `ext4` file system ensures durability of a B*e*trFS log write. Although there is some duplicated work between the layers, we expect ordered journaling mode minimizes this, as a typical B*e*trFS instance spans 11 files from `ext4`'s perspective. That said, these redundancies could be streamlined in future work.

6 Implementation

Rather than do an in-kernel implementation of a write-optimized data structure from scratch, we ported TokuDB into the Linux kernel as the most expedient way to obtain a write-optimized data structure implementation. Such data structure implementations can be com-

Component	Description	Lines
VFS Layer	Translate VFS hooks to TokuDB queries.	1,987
TokuDB	Kernel version of TokuDB. (960 lines changed)	44,293
`klibc`	Compatibility wrapper.	4,155
Linux	Modifications	58

Table 3: Lines of code in B*e*trFS, by component.

Class	ABIs	Description
Memory	4	Allocate buffer pages and heap objects.
Threads	24	Pthreads, condition variables, and mutexes.
Files	39	Access database backend files on underlying, disconnected file system.
zlib	7	Wrapper for kernel zlib.
Misc	27	Print errors, qsort, get time, etc..
Total	101	

Table 4: Classes of ABI functions exported by `klibc`.

plex, especially in tuning the I/O and asymptotic merging behavior.

In this section, we explain how we ported a large portion of the TokuDB code into the kernel, challenges we faced in the process, lessons learned from the experience, and future work for the implementation. Table 3 summarizes the lines of code in B*e*trFS, including the code interfacing the VFS layer to TokuDB, the `klibc` code, and minor changes to the Linux kernel code, explained below. The B*e*trFS prototype uses Linux version 3.11.10.

6.1 Porting Approach

We initially decided the porting was feasible because TokuDB has very few library requirements and is written in a C-like form of C++. In other words, the C++ features used by TokuDB are primarily implemented at compile time (e.g., name mangling and better type checking), and did not require runtime support for features like exceptions. Our approach should apply to other WOIs, such as an LSM tree, inasmuch as the implementation follows a similar coding style.

As a result, we were able to largely treat the TokuDB code we used as a binary blob, creating a kernel module (.ko file) from the code. We exported interfaces used by the B*e*trFS VFS layer to use C linkage, and similarly declared interfaces that TokuDB imported from `klibc` to be C linkage.

We generally minimized changes to the TokuDB code, and selected imported code at object-file granularity. In a few cases, we added compile-time macros to eliminate code paths or functions that would not be used yet required cumbersome dependencies. Finally, when a particularly cumbersome user-level API, such as `fork`, is used in only a few places, we rewrote the code to use a more suitable kernel API. We call the resulting set of dependencies imported by TokuDB `klibc`.

6.2 The `klibc` Framework

Table 4 summarizes the ABIs exported by `klibc`. In many cases, kernel ABIs were exported directly, such as `memcpy`, or were straightforward wrappers for features such as synchronization and memory allocation. In a few cases, the changes were more complex.

The use of `errno` in the TokuDB code presented a particular challenge. Linux passes error codes as negative return values, whereas `libc` simply returns negative one and places the error code in a per-thread variable `errno`. Checks for a negative value and reads of `errno` in TokuDB were so ubiquitous that changing the error-handling behavior was impractical. We ultimately added an `errno` field to the Linux `task` struct; a production implementation would instead rework the error-passing code.

Although wrapping pthread abstractions in kernel abstractions was fairly straightforward, static initialization and direct access to pthread structures created problems. The primary issue is that converting pthread abstractions to kernel abstractions replaced members in the pthread structure definitions. Static initialization would not properly initialize the modified structure. Once the size of pthread structures changed, we had to eliminate any code that imported system pthread headers, lest embedded instances of these structures calculate the wrong size.

In reusing `ext4` as a block store, we faced some challenges in creating module-level file handles and paths. File handles were more straightforward: we were able to create a module-level handle table and use the pread (cursor-less) API to `ext4` for reads and writes. We did have to modify Linux to export several VFS helper function that accepted a `struct file` directly, rather than walking the process-level file descriptor table. We also modified `ext4` to accept input for reads with the `O_DIRECT` flag that were not from a user-level address.

When B*e*trFS allocates, opens, or deletes a block store on the underlying `ext4` file system, the module essentially `chroots` into an `ext4` file system disconnected from the main tree. Because this is kernel code, we also wish to avoid permission checks based on the current process's credentials. Thus, path operations include a "context switch" operation, where the current task's file system root and credentials are saved and restored.

6.3 Changes to TokuDB

With a few exceptions, we were able to use TokuDB in the kernel without major modifications. This subsection outlines the issues that required refactoring the code.

The first issue we encountered was that TokuDB makes liberal use of stack allocation throughout. One function allocated a 12KB buffer on the stack! In con-

trast, stack sizes in the Linux kernel are fixed at compile time, and default to 8KB. In most cases, we were able to use compile-time warnings to identify large stack allocation and convert them to heap allocations and add free functions. In the case where these structures were performance-critical, such as a database cursor, we modified the TokuDB code to use faster allocation methods, such as a kernel cache or per-CPU variable. Similarly, we rewrote several recursive functions to use a loop. Nonetheless, we found that deep stacks of more modest-sized frames were still possible, and increased the stack size to 16 KB. We plan to reign in the maximum stack size in future work.

Finally, we found a small mismatch between the behavior of futexes and kernel wait queues that required code changes. Essentially, recent implementations of pthread condition variables will not wake a sleeping thread up due to an irrelevant interrupt, making it safe (though perhaps inadvisable) in user space not to double-check invariants after returning from `pthread_cond_wait`. The Linux-internal equivalents, such as `wait_event`, can spuriously wake up a thread in a way that is difficult to distinguish without re-checking the invariant. Thus, we had to place all `pthread_cond_wait` calls in a loop.

6.4 Future Work and Limitations

The B*e*trFS prototype is an ongoing effort. The effort has reached sufficient maturity to demonstrate the power of write optimization in a kernel file system. However, there are several points for improvement in future work.

The most useful feature currently missing from the TokuDB codebase is range upserts; upserts can only be applied to a single key, or broadcast to all keys. Currently, file deletion must be implemented by creating a remove upsert for each data block in a file; the ability to create a single upsert applied to a limited range would be useful, and we leave this for future work. The primary difficulty in supporting such an abstraction is tuning how aggressively the upsert should be flushed down to the leaves versus applied to point queries on demand; we leave this issue for future work as well.

One subtle trade-off in organizing on-disk placement is between rename and search performance. B*e*trFS keys files by their path, which currently results in rename copying the file from one disk location to another. This can clearly be mitigated by adding a layer of indirection (i.e., an inode number); however, this is at odds with the goal of preserving data locality within a directory hierarchy. We plan to investigate techniques for more efficient directory manipulation that preserve locality. Similarly, our current prototype does not support hard links.

Our current prototype also includes some double caching of disk data. Nearly all of our experiments measure cold-cache behavior, so this does not affect the fidelity of our results. Profligate memory usage is nonetheless problematic. In the long run, we intend to better integrate these layers, as well as eliminate emulated file handles and paths.

7 Evaluation

We organize our evaluation around the following questions:

- Are microwrites on B*e*trFS more efficient than on other general-purpose file systems?
- Are large reads and writes on B*e*trFS at least competitive with other general-purpose file systems?
- How do other file system operations perform on B*e*trFS?
- What are the space (memory and disk) overheads of B*e*trFS?
- Do applications realize better overall performance on B*e*trFS?

Unless otherwise noted, benchmarks are cold-cache tests. All file systems benefit equally from hits in the page and directory caches; we are interested in measuring the efficiency of cache misses.

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 250 GB, 7200 RPM ATA disk. Each file system used a 4096-byte block size. The system ran Ubuntu 13.10, 64-bit, with Linux kernel version 3.11.10. Each experiment compared with several general purpose file systems, including `btrfs`, `ext4`, `XFS`, and `zfs`. Error bars and $\pm$ ranges denote 95% confidence intervals.

7.1 Microwrites

We evaluated microwrite performance using both metadata and data intensive microbenchmarks. To exercise file creation, we used the TokuBench benchmark [15] to create 500,000 200-byte files in a balanced directory tree with a fanout of 128. The results are shown in Figure 3. TokuBench also measures the scalability of the file system as threads are added; we measured up to 4 threads since our machine has 4 cores.

B*e*trFS exhibited substantially higher throughput than the other file systems. The closest competitor was `zfs` at 1 thread; as more threads were added, the gap widened considerably. Compared to `ext4`, `XFS`, and `btrfs`, B*e*trFS throughput was an order of magnitude higher.

This performance distinction is attributable to both fewer total writes and fewer seeks per byte written—i.e., better aggregation of small writes. Based on profiling from `blktrace`, one major distinction is total bytes written: B*e*trFS writes 4–10$\times$ fewer total MB to disk, with

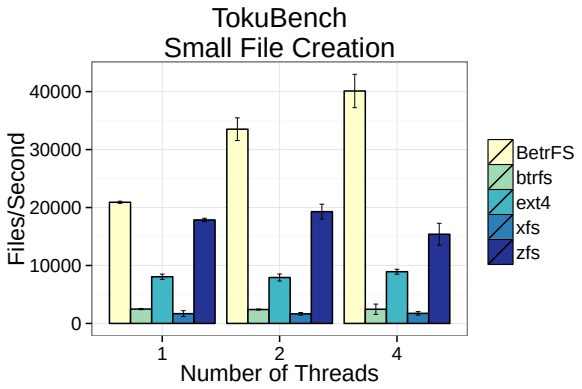


Figure 3: Total time to create 500,000 200-byte files, using 1, 2, and 4 threads. We measure the number of files created per second. Higher is better.

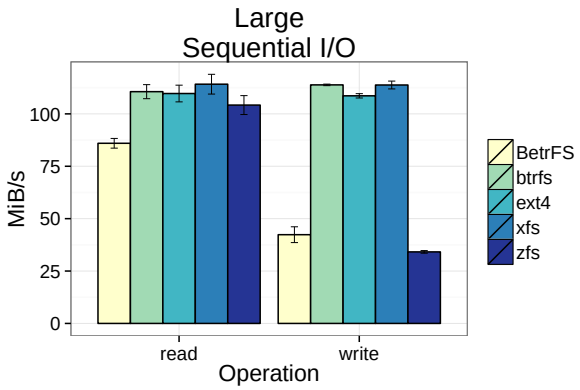


Figure 4: Large file I/O. We sequentially read and write 1GiB files. Higher is better.

File System	Time (s)
BεtrFS	0.17 ± 0.01
`ext4`	11.60 ± 0.39
`XFS`	11.71 ± 0.28
`btrfs`	11.60 ± 0.38
`zfs`	14.75 ± 1.45

Table 5: Time in seconds to execute 1000 4-byte microwrites within a 1GiB file. Lower is better.

an order of magnitude fewer total write requests. Among the other file systems, `ext4`, `XFS`, and `zfs` wrote roughly the same amount of data, but realized widely varying underlying write throughput. The only file system with a comparable write throughput was `zfs`, but it wrote twice as much data using 12.7× as many disk requests.

To measure microwrites to files, we wrote a custom benchmark that performs 1,000 random 4-byte writes within a 1GiB file, followed by an `fsync()`. Table 5 lists the results. BεtrFS was two orders of magnitude faster than the other file systems.

These results demonstrate that BεtrFS improves microwrite performance by one to two orders of magnitude compared to current general-purpose file systems.

7.2 Large Reads and Writes

We measured the throughput of sequentially reading and writing a 1GiB file, 10 blocks at a time. We created the file using random data to avoid unfairly advantaging compression in BεtrFS. In this experiment, BεtrFS benefits from compressing keys, but not data. We note that with compression and moderately compressible data, BεtrFS can easily exceed disk bandwidth. The results are illustrated in Figure 4.

In general, most general-purpose file systems can read and write at disk bandwidth. In the case of a large sequential reads, BεtrFS can read data at roughly 85 MiB/s. This read rate is commensurate with overall disk utilization, which we believe is a result of less aggressive read-ahead than the other file systems. We believe this can be addressed by re-tuning the TokuDB block cache prefetching behavior.

In the case of large writes, the current BεtrFS prototype achieved just below half of the disk's throughput. The reason for this is that each block write must percolate down the interior tree buffers; a more efficient heuristic would detect a large streaming write and write directly to a leaf. As an experiment, we manually forced writes to the leaf in an empty tree, and found write throughput comparable to the other file systems. That said, applying this optimization is somewhat tricky, as there are a number of edge cases where leaves must be read and rewritten or messages must be flushed. We leave this issue for future work.

7.3 Directory Operations

In this section, we measure the impact of the BεtrFS design on large directory operations. Table 6 reports the time taken to run `find`, `grep -r`, `mv`, and `rm -r` on the Linux 3.11.10 source tree, starting from a cold cache. The `grep` test recursively searches the file contents for the string "cpu_to_be64", and the `find` test searches for files named "wait.c". The `rename` test renames the entire kernel source tree, and the `delete` test does a recursive removal of the source.

Both the `find` and `grep` benchmarks demonstrate the value of sorting files and their metadata lexicographically by full path, so that related files are stored near each other on disk. BεtrFS can search directory metadata and file data one or two orders of magnitude faster than other file systems, with the exception of `grep` on `btrfs`, which is

FS	find	grep	dir rename	delete
BetrFS	0.36 ± 0.06	3.95 ± 0.28	21.17 ± 1.01	46.14 ± 1.12
btrfs	14.91 ± 1.18	3.87 ± 0.94	0.08 ± 0.05	7.82 ± 0.59
ext4	2.47 ± 0.07	46.73 ± 3.86	0.10 ± 0.02	3.01 ± 0.30
XFS	19.07 ± 3.38	66.20 ± 15.99	19.78 ± 5.29	19.78 ± 5.29
zfs	11.60 ± 0.81	41.74 ± 0.64	14.73 ± 1.64	14.73 ± 1.64

Table 6: Directory operation benchmarks, measured in seconds. Lower is better.

FS	chmod	mkdir	open	read	stat	unlink	write
BetrFS	4913 ± 0.27	67072 ± 25.68	1697 ± 0.12	561 ± 0.01	1076 ± 0.01	47873 ± 7.7	32142 ± 4.35
btrfs	4574 ± 0.27	24805 ± 13.92	1812 ± 0.12	561 ± 0.01	1258 ± 0.01	26131 ± 0.73	3891 ± 0.08
ext4	4970 ± 0.14	41478 ± 18.99	1886 ± 0.13	556 ± 0.01	1167 ± 0.05	16209 ± 0.2	3359 ± 0.04
XFS	5342 ± 0.21	73782 ± 19.27	1757 ± 0.12	1384 ± 0.07	1134 ± 0.02	19124 ± 0.32	9192 ± 0.28
zfs	36449 ± 118.37	171080 ± 307.73	2681 ± 0.08	6467 ± 0.06	1913 ± 0.04	78946 ± 7.37	18382 ± 0.42

Table 7: Average time in cycles to execute a range of common file system calls. Lower is better.

comparable.

Both the rename and delete tests show the worst-case behavior of BetrFS. Because BetrFS does not include a layer of indirection from pathname to data, renaming requires copying all data and metadata to new points in the tree. We also measured large-file renames, and saw similarly large overheads—a function of the number of blocks in the file. Although there are known solutions to this problem, such as by adding a layer of indirection, we plan to investigate techniques that can preserve the appealing lexicographic locality without sacrificing rename and delete performance.

7.4 System Call Nanobenchmarks

Finally, Table 7 shows timings for a nanobenchmark that measures various system call times. Because this nanobenchmark is warm-cache, it primarily exercises the VFS layer. BetrFS is close to being the fastest file system on `open`, `read`, and `stat`. On `chmod`, `mkdir`, and `unlink`, BetrFS is in the middle of the pack.

Our current implementation of the `write` system call appears to be slow in this benchmark because, as mentioned in Section 5.1, writes in BetrFS issue an upsert to the database, even if the page being written is in cache. This can be advantageous when a page is not written often, but that is not the case in this benchmark.

7.5 Space Overheads

The Fractal Tree index implementation in BetrFS includes a cachetable, which caches tree nodes. Cachetable memory is bounded. BetrFS triggers background flushing when memory exceeds a low watermark and forces writeback at a high watermark. The high watermark is currently set to one eighth of total system memory. This is configurable, but we found that additional cachetable memory had little performance impact in our workloads.

	Total BetrFS Disk Usage (GiB)		
Input Data	After Writes	After Deletes	After Flushes
4	4.14 ± 0.07	4.12 ± 0.00	4.03 ± 0.12
16	16.24 ± 0.06	16.20 ± 0.00	10.14 ± 0.21
32	32.33 ± 0.02	32.34 ± 0.00	16.22 ± 0.00
64	64.57 ± 0.06	64.59 ± 0.00	34.36 ± 0.18

Table 8: BetrFS disk usage, measured in GiB, after writing large incompressible files, deleting half of those files, and flushing B^ε-tree nodes.

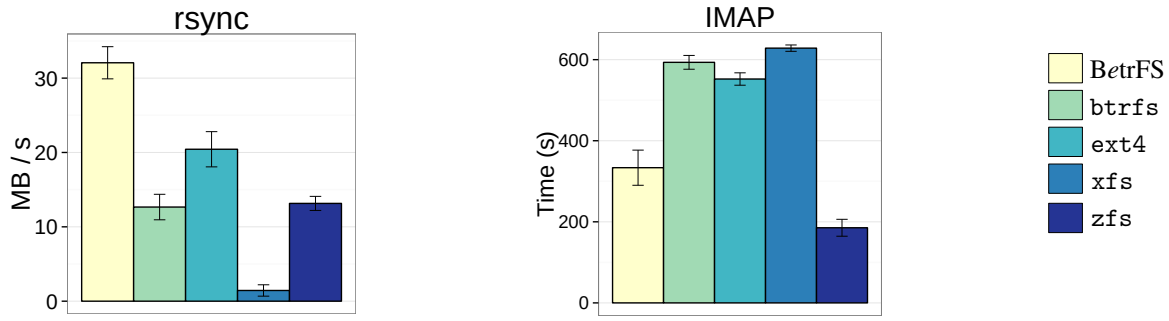
No single rule governs BetrFS disk usage, as stale data may remain in non-leaf nodes after delete, rename, and overwrite operations. Background cleaner threads attempt to flush pending data from 5 internal nodes per second. This creates fluctuation in BetrFS disk usage, but overheads swiftly decline at rest.

To evaluate the BetrFS disk footprint, we wrote several large incompressible files, deleted half of those files, and then initiated a B^ε-tree flush. After each operation, we calculated the BetrFS disk usage using `df` on the underlying `ext4` partition.

Writing new data to BetrFS introduced very little overhead, as seen in Table 8. For deletes, however, BetrFS issues an upsert for every file block, which had little impact on the BetrFS footprint because stale data is lazily reclaimed. After flushing, there was less than 3GiB of disk overhead, regardless of the amount of live data.

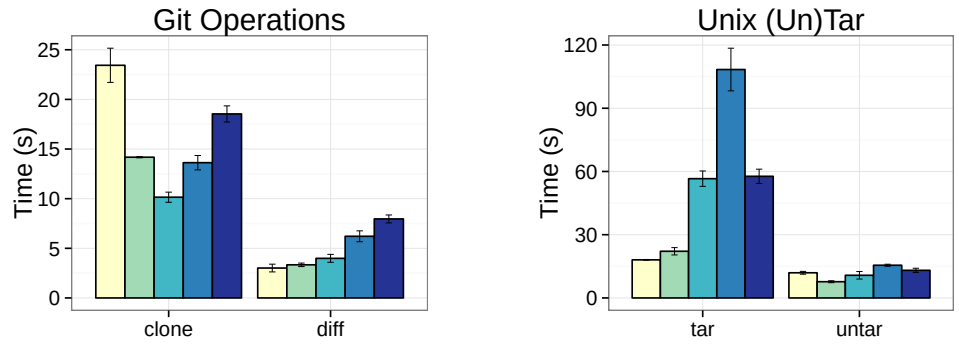
7.6 Application Performance

Figure 5 presents performance measurements for a variety of metadata-intensive applications. We measured the time to `rsync` the Linux 3.11.10 source code to a new di-



(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, 50% message reads, 50% marking and moving messages among inboxes. Execution time in seconds, lower is better.



(c) Git operations. The B*e*trFS 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 5: Application benchmarks

rectory on the same file system, using the `--in-place` option to avoid temporary file creation (Figure 5a). We performed a benchmark using version 2.2.13 of the Dovecot mail server to simulate IMAP client behavior under a mix of read requests, requests to mark messages as read, and requests to move a message to one of 10 other folders. The balance of requests was 50% reads, 50% flags or moves. We exercised the git version control system using a `git-clone` of the local B*e*trFS source tree and a `git-diff` between two milestone commits (Figure 5c). Finally, we measured the time to `tar` and un-`tar` the Linux 3.11.10 source (Figure 5d).

B*e*trFS yielded substantially higher performance on several applications, primarily applications with microwrites or large streaming reads or writes. In the case of the IMAP benchmark, marking or moving messages is a small-file rename in a large directory—a case B*e*trFS handled particularly well, cutting execution time in half compared to most other file systems. Note that the IMAP test is a sync-heavy workload, issuing over 26K `fsync()` calls over 334 seconds, each forcing a full B*e*trFS log flush. `rsync` on B*e*trFS realized significantly higher throughput because writing a large number of modestly sized files in lexicographic order is, on B*e*trFS, aggregated into large, streaming disk writes. Similarly, `tar` benefited from both improved reads of many files in lexicographic order, as well as efficient aggregation of writes. `tar` on B*e*trFS was only marginally better than `btrfs`, but the execution time was at least halved compared to `ext4` and `XFS`.

The only benchmark significantly worse on B*e*trFS was `git-clone`, which does an `lstat` on every new file before creating it—despite cloning into an empty directory. Here, a slower, small read obstructs a faster write. For comparison, the `rsync --in-place` test case illustrates that, if an application eschews querying for the existence of files before creating them, B*e*trFS can deliver substantial performance benefits.

These experiments demonstrate that several real-world, off-the-shelf applications can benefit from exe-

cuting on a write-optimized file system *without application modifications.* With modest changes to favor blind writes, applications can perform even better. For most applications not suited to write optimization, performance is not harmed or could be tuned.

8 Related Work

Previous write-optimized file systems. TokuFS [15] is an in-application library file system, also built using B^ε-trees. TokuFS showed that a write-optimized file system can support efficient write-intensive and scan-intensive workloads. TokuFS had a FUSE-based version, but the authors explicitly omitted disappointing measurements using FUSE.

KVFS [31] is based on a transactional variation of an LSM-tree, called a VT-tree. Impressively, the performance of their transactional, FUSE-based file system was comparable to the performance of the in-kernel `ext4` file system, which does not support transactions. One performance highlight was on random-writes, where they outperformed `ext4` by a factor of 2. They also used stitching to perform well on sequential I/O in the presence of LSM-tree compaction.

TableFS [25] uses LevelDB to store file-system metadata. They showed substantial performance improvements on metadata-intensive workloads, sometimes up to an order of magnitude. They used `ext4` as an object store for large files, so sequential I/O performance was comparable to `ext4`. They also analyzed the FUSE overhead relative to a library implementation of their file system and found that FUSE could cause a 1000× increase in disk-read traffic (see Figure 9 in their paper).

If these designs were ported to the kernel, we expect that they would see some, but not all, of the performance benefits of B*e*trFS. Because the asymptotic behavior is better for B^ε-trees than LSMs in some cases, we expect the performance of an LSM-based file system will not be completely comparable.

Other WOIs. COLAs [4] are an LSM tree variant that uses fractional cascading [13] to match the performance of B^ε-trees for both insertions and queries, but we are not aware of any full featured, production-quality COLA implementation. xDict [8] is a cache-oblivious WOI with asymptotic behavior similar to a B^ε-tree.

Key-Value Stores. WOIs are widely used in key-value stores, including BigTable [12], Cassandra [20], HBase [3], LevelDB [17], TokuDB [33] and TokuMX [34]. BigTable, Cassandra, and LevelDB use LSM-tree variants. TokuDB and TokuMX use Fractal Tree indexes. LOCS [35] optimizes LSM-trees for a key-value store on a multi-channel SSD.

Instead of using WOIs, FAWN [1] writes to a log and maintains an in-memory index for queries. SILT [22] further reduces the design's memory footprint during the merging phase.

Alternatives to update-in-place. The Write Anywhere File Layout (WAFL) uses files to store its metadata, giving it incredible flexibility in its block allocation and layout policies [19]. WAFL does not address the microwrite problem, however, as its main goal is to provide efficient copy-on-write snapshots.

Log-structured File Systems and their derivatives [1, 21, 22, 26] are write-optimized in the sense that they log data, and are thus very fast at ingesting file system changes. However, they still rely on read-modify-write for file updates and suffer from fragmentation.

Logical logging is a technique used by some databases in which operations, rather than the before and after images of individual database pages, are encoded and stored in the log [18]. Like a logical log entry, an upsert message encodes a mutation to a value in the key-value store. However, an upsert is a first-class storage object. Upsert messages reside in B^ε-tree buffers and are evaluated on the fly to satisfy queries, or to be merged into leaf nodes.

9 Conclusion

The B*e*trFS prototype demonstrates that write-optimized indexes are a powerful tool for file-system developers. In some cases, B*e*trFS out-performs traditional designs by orders of magnitude, advancing the state of the art over previous results. Nonetheless, there are some cases where additional work is needed, such as further data-structure optimizations for large streaming I/O and efficient renames of directories. Our results suggest that further integration and optimization work is likely to yield even better performance results.

Acknowledgments

We thank the engineers at Tokutek for developing and open-sourcing a wonderful B^ε-tree implementation from which we built B*e*trFS. We thank the anonymous reviewers and our shepherd, Geoff Kuenning, for their insightful comments on earlier drafts of the work. Vrushali Kulkarni, Oleksii Starov, Sourabh Yerfule, and Ahmad Zaraei contributed to the B*e*trFS prototype. This research was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, CNS-1149229, CNS-1161541, CNS-1228839, CNS-1408782, IIS-1247750, CCF-1314547 and the Office of the Vice President for Research at Stony Brook University.

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 22nd symposium on Operating systems principles* (2009), pp. 1–14.

[2] APACHE. Accumulo. `http://accumulo.apache.org`.

[3] APACHE. HBase. `http://hbase.apache.org`.

[4] 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 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (2007), pp. 81–92.

[5] BENT, J., GIBSON, G. A., GRIDER, G., MCCLELLAND, B., NOWOCZYNSKI, P., NUNEZ, J., POLTE, M., AND WINGATE, M. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC)* (2009).

[6] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (1970), 422–426.

[7] BONWICK, J. ZFS: the last word in file systems. `https://blogs.oracle.com/video/entry/zfs_the_last_word_in`, Sept. 14 2004.

[8] 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 *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2010), pp. 1448–1456.

[9] BRODAL, G. S., AND FAGERBERG, R. Lower bounds for external memory dictionaries. In *Proceedings of the 14th Annual ACM-SIAM symposium on Discrete Algorithms (ACM)* (2003), pp. 546–554.

[10] BUCHSBAUM, A. L., GOLDWASSER, M., VENKATASUBRAMANIAN, S., AND WESTBROOK, J. R. On external memory graph traversal. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2000), pp. 859–860.

[11] CARD, R., TS'O, T., AND TWEEDIE, S. Design and implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux* (1994), pp. 1–6.

[12] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[13] CHAZELLE, B., AND GUIBAS, L. J. Fractional cascading: I. A data structuring technique. *Algorithmica 1*, 1-4 (1986), 133–162.

[14] DOUTHITT, D. Instant 10-20% boost in disk performance: the "noatime" option. `http://administratosphere.wordpress.com/2011/07/29/instant-10-20-boost-in-disk-performance-the-noatime-option/`, July 2011.

[15] ESMET, J., BENDER, M. A., FARACH-COLTON, M., AND KUSZMAUL, B. C. The TokuFS streaming file system. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage (HotStorage)* (June 2012).

[16] File system in userspace. `http://fuse.sourceforge.net/`.

[17] GOOGLE, INC. LevelDB: A fast and lightweight key/value database library by Google. `http://code.google.com/p/leveldb/`.

[18] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[19] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. Tech. rep., NetApp, 1994.

[20] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[21] 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).

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

[23] 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.

[24] QUICKLZ. Fast compression library for c, c#, and java. `http://www.quicklz.com/`.

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

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

[27] SEARS, R., CALLAGHAN, M., AND BREWER, E. A. Rose: compressed, log-structured replication. *Proceedings of the VLDB Endowment 1*, 1 (2008), 526–537.

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

[29] SELTZER, M., BOSTIC, K., MCKUSICK, M. K., AND STAELIN, C. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Conference Proceedings* (San Diego, CA, Jan. 1993), p. 3.

[30] SELTZER, M., SMITH, K. A., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (New Orleans, LA, Jan. 1995), p. 21.

[31] SHETTY, P., SPILLANE, R. P., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with VT-trees. In *FAST* (2013), pp. 17–30.

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

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

[34] TOKUTEK, INC. TokuMX—MongoDB Performance Engine. `http://www.tokutek.com/products/tokumx-for-mongodb/`.

[35] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14, pp. 16:1–16:14.