B*e*trFS: Write-Optimization in a Kernel File System

WILLIAM JANNEN, JUN YUAN, YANG ZHAN, and AMOGH AKSHINTALA,
Stony Brook University
JOHN ESMET, Tokutek
YIZHENG JIAO, ANKUR MITTAL, PRASHANT PANDEY, and PHANEENDRA REDDY,
Stony Brook University
LEIF WALSH, Tokutek
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

The ***B*^ε^*-tree File System***, or B*e*trFS (pronounced "better eff ess"), is the first in-kernel file system to use a write-optimized data structure (WODS). WODS are promising building blocks for storage systems because they support both microwrites and large scans efficiently. Previous WODS-based file systems have shown promise but have been hampered in several ways, which B*e*trFS mitigates or eliminates altogether. For example, previous WODS-based file systems were implemented in user space using FUSE, which superimposes many reads on a write-intensive workload, reducing the effectiveness of the WODS. This article also contributes several techniques for exploiting write-optimization within existing kernel infrastructure. B*e*trFS dramatically improves performance of certain types of large scans, such as recursive directory traversals, as well as performance of arbitrary microdata operations, such as file creates, metadata updates, and small writes to files. B*e*trFS can make small, random updates within a large file 2 orders of magnitude faster than other local file systems. B*e*trFS is an ongoing prototype effort and requires additional data-structure tuning to match current general-purpose file systems on some operations, including deletes, directory renames, and large sequential writes. Nonetheless, many applications realize significant performance improvements

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, IIS-1247750, CCF-1314547, and the Office of the Vice President for Research at Stony Brook University.
Authors' addresses: W. Jannen, J. Yuan, Y. Zhan, and A. Akshintala, New Computer Science, Stony Brook University, Stony Brook, NY 11794-2424; emails: {wjannen, junyuan, yazhan, aakshintala}@cs.stonybrook.edu; J. Esmet, AppNexus, 7901 4th Ave., Apt. A22, Brooklyn, NY 11209; email: john.esmet@gmail.com; Y. Jiao, New Computer Science, Stony Brook University, Stony Brook, NY 11794-2424; email: yjiao@cs.stonybrook.edu; A. Mittal, Microsoft, 15817 NE 90th St., F147, Redmond, WA 98052; email: ankurmit2006@gmail.com; P. Pandey, New Computer Science, Stony Brook University, Stony Brook, NY 11794-2424; email: ppandey@cs.stonybrook.edu; P. Reddy, EMC, 13 Walnut Street, Edison, New Jersey 08817; email: pyreddy@cs.stonybrook.edu; L. Walsh, Two Sigma, Two Sigma Investments, 100 Avenue of the Americas, 16th Floor, New York, NY 10013; email: leif@twosigma.com; M. A. Bender, New Computer Science, Stony Brook University, Stony Brook, NY 11794-2424; email: bender@cs.stonybrook.edu; M. Farach-Colton, Department of Computer Science, Rutgers University, Piscataway, NJ 08854; email: farach@cs.rutgers.edu; R. Johnson, New Computer Science, Stony Brook University, Stony Brook, NY 11794-2424; email: rob@cs.stonybrook.edu; B. C. Kuszmaul, Massachusetts Institute of Technology, 32 Vassar St., 32-g766, Cambridge, MA 02139; email: bradley@mit.edu; D. E. Porter, New Computer Science, Stony Brook University, Stony Brook, NY 11794-2424; email: porter@cs.stonybrook.edu.

DOI: http://dx.doi.org/10.1145/2798729

on B*ε*trFS. For instance, an in-place rsync of the Linux kernel source sees roughly 1.6–22× speedup over commodity file systems.

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

General Terms: Design, Performance, Theory

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

ACM Reference Format:
William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. B*ε*trFS: Write-optimization in a kernel file system. ACM Trans. Storage 11, 4, Article 18 (November 2015), 29 pages.
DOI: http://dx.doi.org/10.1145/2798729

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 require 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 data structures (WODS) [Bender et al. 2007; Brodal et al. 2010; Brodal and Fagerberg 2003; Buchsbaum et al. 2000; O'Neil et al. 1996; Sears et al. 2008; Sears and Ramakrishnan 2012] are exciting because they have the potential to implement both efficient microwrites and large scans. The key strength of the best WODS are that they can ingest data up to 2 orders of magnitude faster than B-trees while matching or improving on the B-tree's point- and range-query performance [Bender et al. 2007; Brodal and Fagerberg 2003].

WODS have been successful in commercial key-value stores and databases [Apache 2015a, 2015b; Chang et al. 2008; Google, Inc. 2015; Lakshman and Malik 2010; Tokutek, Inc. 2013a, 2013b], and previous research on WODS in file systems has shown promise [Esmet et al. 2012; Ren and Gibson 2013; Shetty et al. 2013]. However, progress toward a production-quality write-optimized file system has been hampered in several ways:

—**Code complexity.** A production-quality WODS can easily run to some 50,000 lines of complex code, which is difficult to shoehorn into an OS kernel. Previous WODS-based file systems have been implemented in user space.
—**FUSE squanders microwrite performance.** FUSE [FUSE 2015] issues a query to the underlying file system before almost every update, superimposing intensive searches on top of small-write-intensive workloads. Although WODS are no worse for point queries than any other sensible data structure, small writes in a WODS are much faster than reads, and injecting needless point queries can nullify the advantages of write optimization.
—**Mapping file system abstractions onto a WODS.** We cannot fully realize the potential performance benefits of write-optimization by simply replacing the B-tree or other on-disk data structure in a file system with a WODS. The mapping from VFS-level hooks onto WODS-level operations, as well as the use of kernel infrastructure, must be adapted to the performance characteristics of a WODS.

This article 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 Tokutek's Fractal Tree index, called TokuDB [Tokutek, Inc. 2013a]—a mature, well-engineered B$^\varepsilon$-tree implementation. B*e*trFS's design is tailored to the performance characteristics of Fractal Tree indexes, but otherwise uses Fractal Trees 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, compared to conventional file systems, B*e*trFS can give one to two orders-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 article are:

—The design and implementation of an in-kernel, write-optimized file system.
—A schema for mapping VFS operations to a WODS that unlocks the potential for fast writes while also ensuring locality on disk for fast reads.
—A design that uses unmodified OS kernel infrastructure, which has been tuned for the needs of traditional file systems, yet minimizes the impact on write optimization. For instance, BetrFS uses the OS kernel's page cache to service file reads but can service small writes without blocking to read the surrounding file block into RAM.
—A thorough evaluation of the performance of the B*e*trFS prototype. For instance, our results show almost 2 orders of magnitude improvement on a small-write microbenchmark and a 1.6–22× speedup on an in-place `rsync`.

Our results suggest that a well-designed B$^\varepsilon$-tree-based file system can match or outperform traditional file systems on almost every operation, some by more than 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 operations that are slower in our prototype than in standard file systems are not fundamental to WODS but can be addressed in future work with a combination of algorithmic advances and engineering effort.

2. MOTIVATION AND BACKGROUND

This section explains the role of microwrites in file-system design, how WODS work, and how they can improve file-system performance.

2.1. The Microwrite Problem

A *microwrite* is a write that is too small to utilize the bandwidth to the underlying storage device because of overheads such as seek time or I/O setup time. Microwrites can waste storage bandwidth if the system doesn't take steps to aggregate them.

On a rotating disk, a microwrite is a write for which the seek time dominates the data-transfer time. For example, if a disk has a 5ms seek time and 100MB/s of bandwidth, then a write of 0.5MB utilizes 50% of the disk bandwidth; a write much smaller than this wastes most of the bandwidth. In SSDs, seek time is not an issue, and an SSD can perform many random I/Os per second compared to a rotating disk. Nonetheless, SSDs can still face performance bottlenecks when writes are too small. This is because SSDs perform writes in pages; a microwrite much smaller than the SSD's page size incurs an additional overhead of rewriting data that is already on the device. Small writes can also trigger garbage collection, which consumes internal bandwidth, slowing down all I/O to the SSD.

Microwrites are pervasive in file systems. Metadata updates, such as file creations, permission changes, and timestamps are all examples of microwrites in file systems, and they can pose performance challenges. For example, maintaining atime is a well-known performance problem in many file systems [Douthitt 2011]. Unix and Linux system administrators commonly report 10%–20% performance improvements by disabling the atime option, but this is not always possible because some applications require accurate atime values. Similarly, applications such as email servers can create many small files or make small updates within large files, depending on how a user's mailbox is stored.

Because microwrites are so common, efficiently handling them is one of the main drivers of file system design.

One way to mitigate the microwrite problem is to aggregate or batch microwrites. Logging, for example, batches many microwrites, turning them into large writes. The challenge with pure logging is preserving read locality: Simple logging can convert a sequential read of a large file into a series of much more expensive *microreads* scattered throughout storage—trading one problem for another.

2.2. Logging versus in-Place File Systems

Conventional file-system data structures force file-system designers to choose between optimizing for efficient microwrites and efficient scans of files and directories.

Update-in-place file systems [Card et al. 1994; Sweeny et al. 1996] optimize reads by keeping related items, such as entries in a directory or consecutive blocks in a file, near each other. These file systems can support file and directory scans at near disk bandwidth. On the other hand, since items are updated in place, microdata update performance is often limited by the random-write latency of the underlying disk.

At the other extreme, log-structured file systems [Bent et al. 2009; Rosenblum and Ousterhout 1992; Seltzer et al. 1993, 1995] optimize for writes, often at the expense of large reads and searches. Logging ensures that files can be created and updated rapidly but can leave file and directory updates spread throughout the log, leading to poor performance when the system later accesses these items. These performance problems are particularly noticeable in reads that are too large to be serviced from the system's page cache in RAM, such as recursive directory traversals, virus scans, and backups. The original Log-Structured File System design explicitly assumed sufficient RAM capacity to service all reads [Rosenblum and Ousterhout 1992].

PLFS leverages application-specific behavior to provide both good microwrite and sequential scan performance by essentially implementing both logging and update-in-place [Bent et al. 2009]. PLFS is designed for HPC checkpointing workloads, which are split into separate microwrite-intensive phases and sequential-read-intensive phases. PLFS logs data during the write phases, then sorts and rewrites the data so that it can serve reads quickly during the read phases. However, a general-purpose file system cannot rely on this type of sorting, both because updates and reads are interspersed and because general-purpose file systems cannot tolerate long down times for sorting data. Thus, the PLFS design illustrates the challenge for general-purpose file systems: How to maintain performance for both microwrites and large sequential I/Os for general file system workloads.

We now explain how this tradeoff stems from the performance characteristics of commonly used file-system data structures and how B^{ε}-trees largely eliminate this trade-off.

2.3. B-trees

B-trees are a widely used on-disk data structure in file systems and databases. A B-tree is a search tree with fat nodes. The node size is chosen to be a multiple of the underlying storage device's block size. Typical B-tree node sizes range from 512 bytes up to a fraction of a megabyte.

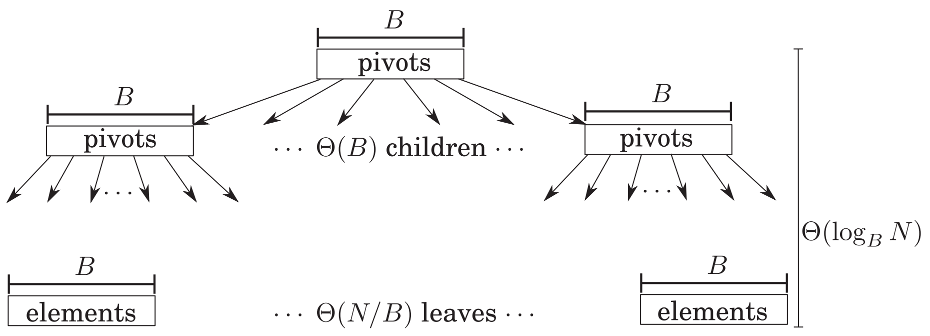

B
pivots
B
pivots
$\cdots\ \Theta(B)$ children $\cdots$
B
pivots
B
elements
$\cdots\ \Theta(N/B)$ leaves $\cdots$
B
elements
$\Theta(\log_B N)$

Fig. 1. A B-tree.

The standard textbook B-tree analysis assumes that all keys have unit size, so that each B-tree node can hold up to B pivot keys and child pointers. Thus, the fanout of a B-tree is $\Theta(B)$ and the depth is $\Theta(\log_B N)$, as shown in Figure 1.

To search for an element, follow the path from the root to the target leaf. To insert or delete an element, first search for the target leaf, update that leaf, and then perform the necessary modifications to keep the tree balanced. The search cost dominates the rebalancing cost since it takes many insertions, along with their searches, before a leaf node overflows. To find all the elements in a range $[a, b]$, first search for a and then scan through the leaves containing elements less than or equal to b.

Our performance analysis assumes that accessing a block of size B requires one I/O, and we measure performance of an algorithm or data structure in terms of the number of I/Os that the algorithm makes during its execution [Aggarwal and Vitter 1988].[1] Our analysis focuses on I/O-bound workloads and thus ignores computational costs, which are typically dwarfed by the I/O costs.

In the worst case, a search requires one I/O per level, for a total cost of $O(\log_B N)$ I/Os. An update requires a search and possibly splits or merges of nodes. The split and merge costs are almost always constant and $O(\log_B N)$ in the worst case, for a total cost of $O(\log_B N)$ I/Os. A range query that returns k items takes $O(\log_B N)$ I/Os for the search and $O(k/B)$ I/Os to read all the items, for a total cost of $O(\log_B N + k/B)$ I/Os.

B-tree implementations face a tradeoff between update and range query performance depending on the node size (B). Larger nodes improve range query performance because more data can be read per seek (the denominator of the k/B term is larger). However, larger nodes make updates more expensive because a node must be completely rewritten each time a new item is added to the index.

Thus, many B-tree implementations use small nodes, resulting in suboptimal range query performance. Especially as free space on disk becomes fragmented, B-tree nodes may become scattered on disk (sometimes called *aging*), thus requiring a range query to perform a seek for each node in the scan and resulting in poor bandwidth utilization. For example, with 4KB nodes on a disk with a 5ms seek time and 100MB/s bandwidth, searches and updates incur almost no bandwidth costs. Range queries, however, must perform a seek every 4KB, resulting in a net bandwidth of 800KB/s, less than 1% of the disk's potential bandwidth.

[1]In the algorithms literature, this model is often called the Disk-Access model, External-Memory model, or I/O model.

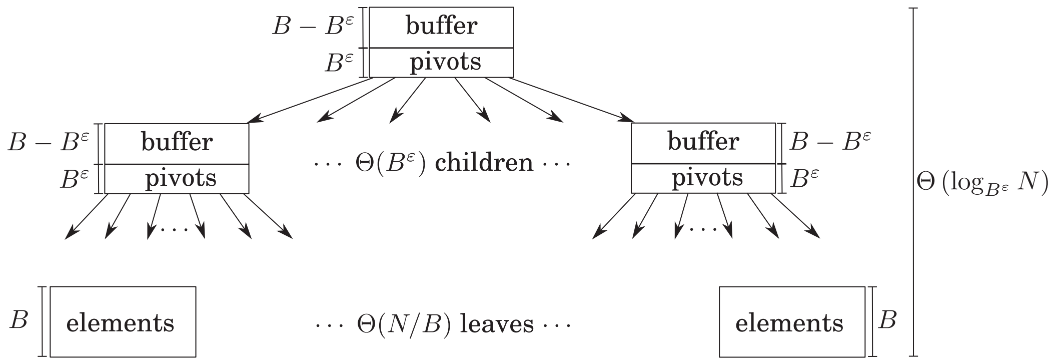

$B - B^{\varepsilon}$
buffer
B^{ε}
pivots
$\cdots\ \Theta(B^{\varepsilon})$ children $\cdots$
B
elements
$\cdots\ \Theta(N/B)$ leaves $\cdots$
$\Theta\left(\log_{B^{\varepsilon}} N\right)$

Fig. 2. A B^{ε}-tree.

2.4. B^{ε}-trees

A B^{ε}-tree is an example of a *write-optimized data structure* (WODS). B^{ε}-trees were proposed by Brodal and Fagerberg [2003] as a way to demonstrate an asymptotic performance tradeoff curve between B-trees [Comer 1979] and buffered repository trees [Buchsbaum et al. 2000]. Both data structures support the same operations, but a B-tree favors queries, whereas a buffered repository tree favors inserts.

Researchers, including the authors of this article, have recognized the practical utility of a B^{ε}-tree when configured to occupy the "middle ground" of this curve—realizing query performance comparable to a B-tree but insert performance orders of magnitude faster than a B-tree.

We first describe the B^{ε}-tree structure and analyze its performance in the same model used earlier. We then explain how B^{ε}-trees enable implementers to use much larger blocks, resulting in range-query performance at near disk bandwidth. We describe *upserts*, a mechanism for improving performance by avoiding searches in some application-level operations. Finally, we compare B^{ε}-trees to log-structured merge-trees, another popular WODS. Bender et al. present a longer version of the tutorial in this section [Bender et al. 2015].

Like a B-tree, the node size in a B^{ε}-tree is chosen to be a multiple of the underlying storage device's block size. Typical B^{ε}-tree node sizes range from a few hundred kilobytes to a few megabytes. In both B-trees and B^{ε}-trees, internal nodes store pivot keys and child pointers, and leaves store key-value pairs, sorted by key. For simplicity, one can think of each key-value or pivot-pointer pair as being unit size; both B-trees and B^{ε}-trees can store keys and values of different sizes in practice. Thus, a leaf of size B holds B key-value pairs, which we call **items** herein.

The distinguishing feature of a B^{ε}-tree is that internal nodes also allocate some space for a buffer, as shown in Figure 2. The buffer in each internal node is used to store **messages**, which encode updates that will eventually be applied to leaves under this node. This buffer is not an in-memory data structure; it is part of the node and is written to disk, evicted from memory, or the like whenever the node is. The value of ε, which must be between 0 and 1, is a tuning parameter that selects how much space internal nodes use for pivots ($\approx B^{\varepsilon}$) and how much space is used as a buffer ($\approx B - B^{\varepsilon}$).

Inserts and deletes. Insertions are encoded as "insert messages," addressed to a particular key and added to the buffer of the root node of the tree. When enough messages have been added to a node to fill the node's buffer, a batch of messages are flushed to one of the node's children. Generally, the child with the most pending messages is selected. Over the course of flushing, each message is ultimately delivered to the appropriate leaf node, and the new key and value are added to the leaf. When

Table I. Asymptotic I/O Costs of Important Operations

Data Structure	Insert	Point Query no Upserts	Point Query w/Upserts	Range Query
B$^\varepsilon$-tree	$\frac{\log_B N}{\varepsilon B^{1-\varepsilon}}$	$\frac{\log_B N}{\varepsilon}$	$\frac{\log_B N}{\varepsilon}$	$\frac{\log_B N}{\varepsilon} + \frac{k}{B}$
B$^\varepsilon$-tree ($\varepsilon = 1/2$)	$\frac{\log_B N}{\sqrt{B}}$	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$
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$^\varepsilon$-trees simultaneously support efficient inserts, point queries (even in the presence of upserts), and range queries. These complexities apply for $0 < \varepsilon \leq 1$. Note that ε is a design-time constant. We show the complexity for general ε and evaluate the complexity when ε is set to a typical value of $1/2$. The $1/\varepsilon$ factor evaluates to a constant that disappears in the asymptotic analysis.

a leaf node becomes too full, it splits, just as in a B-tree. Similar to a B-tree, when an interior node gets too many children, it splits and the messages in its buffer are distributed between the two new nodes.

Moving messages down the tree in batches is the key to the B$^\varepsilon$-tree's insert performance. By storing newly inserted messages in a buffer near the root, a B$^\varepsilon$-tree can avoid seeking all over the disk to put elements in their target locations. The B$^\varepsilon$-tree only moves messages to a subtree when enough messages have accumulated for that subtree to amortize the I/O cost. Although this involves rewriting the same data multiple times, this can *improve* performance for smaller, random inserts, as our later analysis shows.

B$^\varepsilon$-trees delete items by inserting "tombstone messages" into the tree. These tombstone messages are flushed down the tree until they reach a leaf. When a tombstone message is flushed to a leaf, the B$^\varepsilon$-tree discards both the deleted item and the tombstone message. Thus, a deleted item, or even entire leaf node, can continue to exist until a tombstone message reaches the leaf. Because deletes are encoded as messages, deletions are algorithmically very similar to insertions.

Point and range queries. Messages addressed to a key k are guaranteed to be applied to k's leaf or stored in a buffer along the root-to-leaf path toward key k. This invariant ensures that point and range queries in a B$^\varepsilon$-tree have a similar I/O cost to a B-tree.

In both a B-tree and a B$^\varepsilon$-tree, a point query visits each node from the root to the correct leaf. However, in a B$^\varepsilon$-tree, answering a query also means checking the buffers in nodes on this path for messages and applying relevant messages before returning the results of the query. For example, if a query for key k finds an entry (k, v) in a leaf and a tombstone message for k in the buffer of an internal node, then the query will return "NOT FOUND" since the entry for key k has been logically deleted from the tree. Note that the query need not update the leaf in this case—it will eventually be updated when the tombstone message is flushed to the leaf. A range query is similar to a point query except that messages for the entire range of keys must be checked and applied as the appropriate subtree is traversed.

Asymptotic Analysis. Table I lists the asymptotic complexities of each operation in a B-tree and B$^\varepsilon$-tree. Parameter ε is set at design time and ranges between 0 and 1. As we explain later in this section, making ε an exponent simplifies the asymptotic analysis and creates an interesting tradeoff curve.

The point-query complexities of a B-tree and a B$^\varepsilon$-tree are both logarithmic in the number of items ($O(\log_B N)$); a B$^\varepsilon$-tree adds a constant overhead of $1/\varepsilon$. Compared to a B-tree with the same node size, a B$^\varepsilon$-tree reduces the fanout from B to B^ε fanout[2], making the tree taller by a factor of $1/\varepsilon$. Thus, for example, querying a B$^\varepsilon$-tree, where $\varepsilon = 1/2$, will require at most twice as many I/Os.

Range queries incur a logarithmic search cost for the first key, as well as a cost that is proportional to the size of the range and how many disk blocks the range is distributed across. The scan cost is roughly the number of keys read (k) divided by the block size (B). The total cost of a range query is $O(k/B + \log_B N)$ I/Os.

Inserting a message has a similar complexity to a point query, except that *the cost of insertion is divided by the batch size* ($B^{1-\varepsilon}$). For example, if $B = 1024$ and $\varepsilon = 1/2$, a B$^\varepsilon$-tree can perform inserts $\approx \varepsilon B^{1-\varepsilon} = \frac{1}{2}\sqrt{1024} = 16$ times faster than a B-tree.

Intuitively, the tradeoff with parameter ε is how much space in the node is used for storing pivots and child pointers ($\approx B^\varepsilon$) and how much space is used for message buffers ($\approx B - B^\varepsilon$). As ε increases, so does the branching factor (B^ε), causing the depth of the tree to decrease and searches to run faster. As ε decreases, the buffers get larger, batching more inserts for every flush and improving overall insert performance.

At one extreme, when $\varepsilon = 1$, a B$^\varepsilon$-tree is just a B-tree, since interior nodes contain only pivot keys and child pointers. At the other extreme, when $\varepsilon = 0$, a B$^\varepsilon$-tree is a binary search tree with a large buffer at each node, called a buffered repository tree [Buchsbaum et al. 2000].

The most interesting configurations place ε strictly between 0 and 1, such as $\varepsilon = 1/2$. For such configurations, a B$^\varepsilon$-tree has the same asymptotic point query performance as a B-tree, but asymptotically better insert performance.

Write optimization. Batching small, random inserts is an essential feature of WODS, such as a B$^\varepsilon$-tree or LSM-tree. Although a WODS may issue a small write multiple times as a message moves down the tree, once the I/O cost is divided among a large batch, the cost per insert or delete is much smaller than one I/O per operation. In contrast, a workload of random inserts to a B-tree requires a *minimum* of one I/O per insert to write the new element to its target leaf.

The B$^\varepsilon$-tree flushing strategy is designed to ensure that it can always move elements in large batches. Messages are only flushed to a child when the buffer of a node is full, containing $B - B^\varepsilon \approx B$ messages. When a buffer is flushed, not all messages are necessarily flushed—messages are only flushed to children with enough pending messages to offset the cost of rewriting the parent and child nodes. Specifically, at least $(B - B^\varepsilon)/B^\varepsilon \approx B^{1-\varepsilon}$ messages are moved from the parent's buffer to the child's on each flush. Consequently, any node in a B$^\varepsilon$-tree is only rewritten if a sufficiently large portion of the node will change.

Large Nodes. B$^\varepsilon$-trees can perform range queries at near disk bandwidth because they use large nodes that can amortize the cost of seeks. Unlike B-trees, which face a tradeoff between search and insert costs, batching in a B$^\varepsilon$-tree causes the bandwidth per insert to grow much more slowly as B increases. As a result, B$^\varepsilon$-trees use node sizes of a few hundred kilobytes to a few megabytes.

In the earlier comparison of insert complexities, we stated that a B$^\varepsilon$-tree with $\varepsilon = 1/2$ would be twice as deep as a B-tree because some fanout is sacrificed for buffer space. This is only true when the node size is the same. Because a B$^\varepsilon$-tree can use larger nodes in practice, a B$^\varepsilon$-tree can still have close to the same fanout and height as a B-tree.

[2]More precisely, the fanout is $B^\varepsilon + 1$ and the tree height is $O(\log_{B^\varepsilon+1} N)$, which explains why the fanout is 2 and the tree height is $O(\log N)$ when $\varepsilon = 0$. By removing this additive 1, the asymptotics simplify better, as given in Table I, but the simplification does not apply when $\varepsilon = 0$.

This analysis assumes that all keys have unit size and that nodes can hold B keys; real systems must deal with variable-sized keys, so B, and hence ε, are not fixed or known a priori. Variable-sized keys are challenging to analyze in B-trees [Bender et al. 2010]. Nonetheless, the main insight of B$^\varepsilon$-trees—that we can speed up insertions by buffering items in internal nodes and flushing them down the tree in batches—still applies in this setting.

In practice, B$^\varepsilon$-tree implementations select a fixed physical node size and fanout (B^ε). For the implementation in TokuDB and BetrFS, nodes are approximately 4MB, and the branching factor ranges from 4 to 16. As a result, the fractal tree can always flush data in batches of at least 256KB.

In summary, B$^\varepsilon$-tree implementations can match the search performance of B-trees, perform inserts and deletes orders of magnitude faster, and execute range queries at near disk bandwidth.

Performance Example. To get a feeling for B$^\varepsilon$-tree performance in practice, consider the following example. Suppose a key-value store holds 1TB of data, with 128-byte keys and 1KB records (key+value). Suppose that data are logged for durability, and periodically all updates in the log are applied to the main tree in a 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 two I/Os: one I/O to read in the target leaf and one I/O to write it back to disk after updating its contents.

For comparison, suppose a B$^\varepsilon$-tree has a branching factor of 16 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 for the entire set. 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.

When one compares the amortized cost to write one update in this example, a B$^\varepsilon$-tree improves small write throughput over a B-tree by a factor of almost 20. The B$^\varepsilon$-tree includes 1TB/1MB = 2^{20} leaves with a fanout of 16, yielding a height of $1+\log_{16} 2^{20} \approx 6$. Each item is therefore involved in seven I/Os: It is written once at each level, and there is an extra I/O to read in its leaf. But each write also moves 64 other items (1MB/16 = 64kB = 64 records). Thus, the average per-item cost of flushing an item to a leaf is $7/64$. Since a B-tree would require two I/Os for each item, the B$^\varepsilon$-tree is able to insert data $2/(7/64) = 18$ times faster than a B-tree. As key-value pairs get smaller, say for metadata updates, this speedup factor grows, because B-trees must perform two I/Os, no matter how small the update, whereas B$^\varepsilon$-trees share the incurred I/Os over more items.

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 because the B$^\varepsilon$-tree seeks only once every 1MB versus once every 4KB in the B-tree.

Upserts. B$^\varepsilon$-trees support *upserts*, an efficient method for updating a key-value pair in the tree without first searching for the old value. An example use case for an upsert is overwriting a small portion of a file block when that block is not in cache.

Upserts enable applications to overcome the B$^\varepsilon$-tree *search-insert asymmetry*. B$^\varepsilon$-trees can perform inserts orders of magnitude faster than a B-tree (or searches in either data structure), but searches are essentially no faster than in a B-tree. Because of this performance asymmetry, an application that performs a query-modify-insert operation to update a value in a B$^{\varepsilon}$-tree will be limited by the speed of the query.

Upserts are a generalization of the tombstone messages described earlier. With upserts, an application can update the value associated with key k in the B$^{\varepsilon}$-tree by inserting an "upsert message" $(k, (f, \Delta))$ into the tree, where f is a call-back function and Δ is auxiliary data specifying the update to be performed. This upsert message is semantically equivalent to performing a query followed by an insert:

$$v \leftarrow \texttt{query}(k); \quad \texttt{insert}(k, f(v, \Delta)).$$

However, the upsert does not perform these operations. Rather, the message $(k, (f, \Delta))$ is inserted into the tree like any other piece of data.

When an upsert message $(k, (f, \Delta))$ is flushed to a leaf, the value v associated with k in the leaf is replaced by $f(v, \Delta)$, and the upsert message is discarded. If the application queries k before the upsert message reaches a leaf, then the upsert message is applied to v before the query returns. If multiple upserts are pending for the same key, they are applied to the key in the order they were inserted into the tree.

Upserts have many applications in file systems. For example, upserts can be used to update a few bytes in a file block, to update a file's access time, or to increment a file's reference count. Upserts can encode any modification that is asynchronous and depends only on the key, the old value, and some auxiliary data that can be stored with the upsert message.

The upsert mechanism does not interfere with I/O performance of searches because the upsert messages for a key k always lie on the search path from the root of the B$^{\varepsilon}$-tree to the leaf containing k. Thus, the upsert mechanism can speed updates by one to 2 orders of magnitude without slowing down queries.

Log-structured merge trees. Log-structured merge trees (LSM trees) [O'Neil et al. 1996] are WODS with many variants [Sears and Ramakrishnan 2012; Shetty et al. 2013; Wu et al. 2015]. An LSM-tree typically consists of a logarithmic number of 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 level. The factor by which each level grows is a tunable parameter comparable to the branching factor (B^{ε}) in a B$^{\varepsilon}$-tree . For ease of comparison, Table I gives the I/O complexities of operations in an LSM-tree with growth factor B^{ε}.

LSM trees can be tuned to have the same insertion complexity as a B$^{\varepsilon}$-tree, but queries in a naïvely implemented LSM tree can require $O(\frac{\log_B^2 N}{\varepsilon})$ I/Os because the query must be repeated in $O(\log_B N)$ B-trees. Most LSM tree implementations use Bloom filters to avoid queries in all but one of the B-trees, thus improving point query performance to $O(\frac{\log_B N}{\varepsilon})$ I/Os [Apache 2015b; Chang et al. 2008; Google, Inc. 2015; Lakshman and Malik 2010].

One problem for LSM-trees is that the benefits of Bloom filters do not extend to range queries. Bloom filters are only designed to improve point queries and do not support range queries. Thus, a range query must be done on every level of the LSM-tree—squaring the search overhead in Table I and yielding strictly worse asymptotic performance than a B$^{\varepsilon}$-tree or a B-tree.

A second advantage of a B$^{\varepsilon}$-tree over an LSM-tree is that B$^{\varepsilon}$-trees can effectively use upserts, whereas upserts in an LSM-tree will ruin the performance advantage of adding Bloom filters. As discussed earlier, upserts address a search-insert asymmetry common to any WODS, including LSM-trees. When an application uses upserts, it is possible for *every* level of the tree to contain pending messages for a key. Thus, a

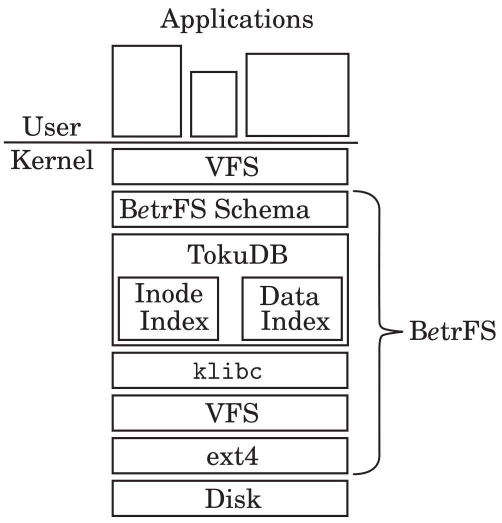


Fig. 3. The B*e*trFS architecture.

subsequent point query will still have to query every level of the tree, thus defeating the purpose of adding Bloom filters. Note that querying every level of an LSM-tree also squares the overhead compared to a B$^{\varepsilon}$-tree or B-tree and is more expensive than walking the path from root to leaf in a B$^{\varepsilon}$-tree .

In summary, Bloom filter-enhanced LSM-trees can match the performance of B$^{\varepsilon}$-trees for some but not all workloads. B$^{\varepsilon}$-trees asymptotically dominate LSM-tree performance. In particular, B$^{\varepsilon}$-trees are asymptotically faster than LSM-trees for small range queries and point queries in upsert-intensive workloads. For this reason, B*e*trFS uses B$^{\varepsilon}$-trees, as implemented in Tokutek's Fractal Tree indexes.

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$^{\varepsilon}$-trees. The overall system architecture is illustrated in Figure 3. In short, B*e*trFS runs a kernel-level port of TokuDB on top of another file system (`ext4` in our prototype, although we do not use any `ext4`-specific features) and is supported by a shim layer we call `klibc`. The B*e*trFS schema translates VFS-level operations into queries against the TokuDB key-value store.

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.

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

B*e*trFS's stacked file-system design cleanly separates the complex task of implementing write-optimized data structures from other file system implementation tasks such as 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 nontrivial task. We ported TokuDB into the kernel by writing a shim layer, which we call `klibc`, that translates the TokuDB

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

Fig. 4. Example of the sort order used in BetrFS.

external dependencies into kernel functions for locking, memory allocation, and file I/O. Section 6 describes `klibc` and summarizes our experiences and lessons learned from the project.

4. THE BeTRFS FILE-SYSTEM SCHEMA

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

4.1. BetrFS Data and Metadata Indexes

BetrFS 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, BetrFS is able to pack many index entries into each B^ε-tree node.

The metadata index. The BetrFS prototype maintains an index mapping full path-names (relative to the mount point) to file metadata (roughly equivalent to the contents of `struct stat`):

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

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 lexico-graphically. Thus, items within the same directory are stored consecutively, as illus-trated in Figure 4. With this ordering, scanning a directory, either recursively or not, can be implemented as a range query, and the layout of metadata entries on disk is equivalent to a level-order traversal of the file-system tree.

The data index. Although keys and values may be variable-sized, the BetrFS proto-type breaks files into blocks of up to 4,096-bytes in size for better integration with the page cache. Thus, the data index maps (file, offset) tuples to blocks:

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

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 Section 2.4, almost always physically adjacent on disk. This enables file contents to be read sequentially at near disk bandwidth.

Because the 4KB block size is only a schema-level parameter, for small and sparse files, BetrFS can avoid writing zero-padded bytes into TokuDB and lower layers. BetrFS implements sparse files by simply omitting the sparse blocks from the data index. BetrFS 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. This optimiza-tion significantly reduces the overheads on small-file benchmarks. For instance, this optimization improves throughput on TokuBench (Section 7) by 50%–70%.

4.2. Implementing B*ε*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()`. Whenever possible, B*ε*trFS avoids this read-modify-write pattern, instead issuing blind writes or writes without reads.

Reading and writing files in BεtrFS. B*ε*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*ε*trFS supports efficient file writes of any size via upserts and inserts. Application writes smaller than one 4K block become messages of the form:

$$\text{U}\textsc{psert}(\text{WRITE}, (\mathit{path}, n), \mathit{offset}, v, \ell),$$

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.

As explained in Section 2.4, 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 because a solution must carefully address several subtle issues related to pending messages and splitting leaf nodes.

File-system metadata operations in BεtrFS. As shown in Table II, B*ε*trFS also converts almost all metadata updates, such timestamp changes, file creation, and symbolic linking, into blind inserts. Our initial design used upserts [Jannen et al. 2015]; however, all VFS-level inode modifications require having the inode in cache. Thus, we found metadata updates simpler to implement as an insert of the entire inode. Because inodes are small in B*ε*trFS (144 bytes), converting from an upsert to an insert had negligible impact on performance.

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

Unlink and truncate can both remove blocks from a file. Assuming the file inode is already in the VFS metadata cache to check permissions on the operation, B*ε*trFS already knows the length of the file and thus can compute the data index keys for the operation. B*ε*trFS removes the inode and data blocks in the simplest possible way: By issuing a TokuDB delete for each block in the data index. TokuDB delete operations are small messages that are as efficient as upserts; nonetheless, issuing $O(n)$ deletes can make this an expensive task. The evaluation shows that file deletion time is currently linear in the file's size (Figure 8).

Keying by full path also makes recursive directory traversals efficient but makes it nontrivial to implement efficient renames. For example, our current implementation renames files and directories by reinserting all the key-value pairs under their new keys

Table II. Implementation Strategies for Basic File-system Operations

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

Almost all operations are implemented using efficient upserts, blind inserts, point queries, or range queries. Unlink, truncate, and rename currently scale with file and/or directory sizes.

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. 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 and still preserve file locality as the system ages. We leave these optimizations for future work.

Although the schema described can use efficient inserts, upserts, and delete messages 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. These precondition checks induce queries to the metadata index in B*e*trFS. Since queries in B$^\varepsilon$-trees are only as fast as B-tree queries, preceding every upsert with a query could potentially ruin any performance gains on upsert-based operations, such as creations. In our experiments, the OS cache of file and directory information was able to answer these queries, enabling file creation and the like to run at the full speed of upserts. We expect that OS caching will suffice for all but the largest file systems.

Crash consistency. We use the TokuDB transaction mechanism for crash consistency. TokuDB transactions are equivalent to full data journaling, with all B*e*trFS data and metadata updates logged to a file in the underlying `ext4` file system. Each operation, including all inserted values, is placed in a logical redo log. Log entries are retired in order, and no updates are applied to the tree on disk ahead of the TokuDB logging mechanism.

TokuDB implements recovery with a combination of checkpointing the B$^\varepsilon$-tree and redo logging. Entries are appended to one of two in-memory log buffers (16MiB by default). These buffers are rotated and flushed to disk every second or when a log buffer overflows.

Checkpoints are taken every 60 seconds, which includes flushing a complete, consistent copy of the B$^\varepsilon$-tree to disk. After the checkpoint, writes to B$^\varepsilon$-tree nodes on disk are applied copy-on-write; any nodes rendered unreachable after a checkpoint are garbage-collected. Once a checkpoint completes, previous log entries can be discarded. Recovery simply replays all logged updates to the checkpointed tree. Any blocks written between a checkpoint and a crash are also garbage-collected as part of recovery.

Because our log writes all data at least twice in our current design, our throughput for large writes is limited to, at most, half of the raw disk bandwidth.

TokuDB includes transactional updates to keys, implemented using multiversion concurrency control (explained further in Section 5.3). B*e*trFS internally uses transactions to ensure crash consistency across metadata operations. 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, thus making transaction conflicts and aborts vanishingly rare.

Transactions are also beneficial to the performance of some file system operations. In B*e*trFS, the file system `sync` operation is reduced to the cost of a TokuDB log flush. These benefits are large for sync-heavy workloads, as shown in Section 7.6.

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` [QuickLZ 2015], 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. We note that our experiments are careful to use random or nontrivial data patterns for a fair measurement.

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 because subblock writes can be faster than a read. This section explains several strategies we used to improve 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 the page as dirty and later writes the page out (i.e., write-back caching). In 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 (i.e., write-through caching). When an application writes to a clean cached page, B*e*trFS issues an upsert to the on-disk 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.

This policy can speed up small writes by reducing write amplification due to the coarse granularity of block-based I/O. Under the conventional policy, the file system must always write back a full disk block, even if the application only writes a single byte to the page. B*ε*trFS achieves finer-than-block granularity for its writes by encoding small writes as upserts, which are written to the B*ε*trFS log, and group commit lets B*ε*trFS aggregate and pack many of its subblock writes into a single block I/O.

Note that B*ε*trFS's 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*ε*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*ε*trFS's `mmap` implementation uses the default read and write page mechanisms, which operate at page granularity.

Our write-through caching design keeps the page cache coherent with disk. B*ε*trFS leverages the page cache for faster warm-cache reads but avoids unnecessary full-page write-backs.

5.2. FUSE Is Write De-Optimized

We implemented B*ε*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 marshaling 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 workloads into read-intensive workloads because FUSE 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, such as in network file systems.

This design choice is benign when the underlying data structure of the file system is a B-tree. Since searches and inserts have roughly the same cost, and the update must be issued at block granularity, adding a query before every update will not cause a significant slowdown and may be required for subblock metadata updates. In contrast, these searches can choke a write-optimized data structure, where insertions are 2 orders of magnitude faster than searches. The TokuFS authors explicitly cite these searches as the cause of the disappointing performance of their FUSE implementation [Esmet et al. 2012].

The TableFS authors identified another source of FUSE overhead: Double caching of inode information in the kernel [Ren and Gibson 2013]. 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. TokuDB Overview

This subsection explains additional TokuDB implementation details that are relevant to understanding the performance of B*ε*trFS. Recall that TokuDB implements Fractal Tree indexes, an extension to the B$^{\varepsilon}$-tree data structure described in Section 2.4.

Each node in the B$^\varepsilon$-tree is roughly 4MiB in order to ensure that all updates are large enough to amortize the seek costs of the write. The on-disk node size can vary because its size is measured before compression. Disk blocks are allocated at 4KiB granularity in contiguous runs, which are found using a first-fit algorithm.

Internal (non-leaf) nodes contain three kinds data: child pointers, pivot keys, and operation messages. For internal nodes, the child pointers and pivot keys provide the same indexing data found in a B-tree. The tree employs B-tree rebalancing, with internal nodes containing up to 16 children. The messages contain data that are traveling from the root of the tree to the leaves. For example, a typical message would contain an "insert" command that specifies a transaction identifier, a key, and a value. When the message arrives at a leaf, it updates the appropriate key-value record.

When an internal node's buffers fill up, messages are flushed to the child or children with enough pending messages to amortize the cost of rewriting the parent and child. This structured flushing is essential to how B$^\varepsilon$-trees batch small writes.

Leaf nodes of a B$^\varepsilon$-tree contain key-value records. A leaf is stored in a roughly 4MiB contiguous range on disk but is internally organized into a height-1 subtree composed of *basement nodes*, each roughly 64KiB. For a point query, the system reads the appropriate 64KiB basement node directly from disk, not the entire height-1 subtree. For a write operation, however, the system rewrites the entire height-1 subtree. Each basement node is compressed separately: The 64KiB and 4MiB sizes are measured before compression, and so the size of a leaf node on disk depends on the compression factor.

To implement multiversion concurrency control, each key-value record contains a key and one or more values, with each value tagged by a transaction identifier. One possible value is a tombstone, which indicates that the record is deleted. When the value produced by a transaction is no longer visible, the system may remove that value from the key-value record.

TokuDB includes its own cache of on-disk nodes, which are retained according to an LRU policy. In future work, we will better integrate the cache of on-disk nodes with the Linux page cache.

5.4. Ext4 as a Block Manager

Since TokuDB stores data in compressed nodes that 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. 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 that ordered journaling mode minimizes this because a typical B*e*trFS instance spans just 11 files from `ext4`'s perspective. That said, these redundancies could be streamlined in future work.

5.5. Representation of Files

Each block of a file in B*e*trFS is indexed logically by path and offset, rather than by physical location or in extents, as is common practice in many file systems. Rather, mapping this logical key to a physical location is encapsulated in the B$^\varepsilon$-tree implementation.

Table III. Lines of Code in B*ε*trFS by Component

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

Encapsulating data placement within the B^ε-tree has many advantages. Sparse files can be trivially implemented by using the inode size field and omitting keys from the data index. For example, a sparse file `/home/foo` containing only blocks 1 and 100 will have the values of `/home/foo:001` and `/home/foo:100` stored consecutively in the tree—allowing the entire file to be read in one efficient I/O. No space is wasted, neither in an indexing structure nor in the tree via zero-filled blocks. Separating data indexing from the inode also keeps the size of B*ε*trFS metadata small. B*ε*trFS is able to pack the inode information of several files into the space of a single disk block. Pushing the indexing to the B^ε-tree also facilitates the implementation of upserts, multiversion concurrency control, and variable sized blocks because it is difficult to manage subblock and variable-sized blocks in a physical index.

The disadvantage of this representation is that a key must be stored for every value. Keys are highly repetitive, and this overhead is negated with compression.

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 complex, 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 III summarizes the lines of code in B*ε*trFS, including the code interfacing the VFS layer to TokuDB, the `klibc` code, and minor changes to the Linux kernel code, explained later. The B*ε*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 WODS, 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*ε*trFS VFS layer to use C linkage and similarly declared interfaces that TokuDB imported from `klibc` to use 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 but 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` .

Table IV. Classes of ABI Functions Exported by `klibc`

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	

6.2. The `klibc` Framework

Table IV 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 1 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 functions 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 contrast, 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 rein 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 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 a thread in a way that is difficult to distinguish without rechecking the invariant. Thus, we had to place all `pthread_cond_wait` calls in a loop.

6.4. Future Work and Limitations

The B*ε*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 tradeoff in organizing on-disk placement is between rename and search performance. B*ε*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 preserves 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*ε*trFS more efficient than on other general-purpose file systems?
—Are large reads and writes on B*ε*trFS at least competitive with other general-purpose file systems?
—How do other file system operations perform on B*ε*trFS?
—What are the space (memory and disk) overheads of B*ε*trFS?
—Do applications realize better overall performance on B*ε*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 3.40GHz Intel Core i7 CPU, 4GB RAM, and a 250GB, 7200 RPM ATA disk. Each file system used a 4,096-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,

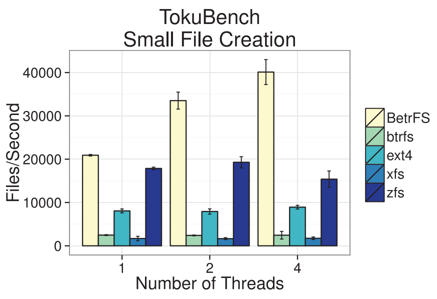


Fig. 5. Total time to create 500,000 200-byte files, using 1, 2, and 4 threads. We measure the number of files created per second. Higher is better.

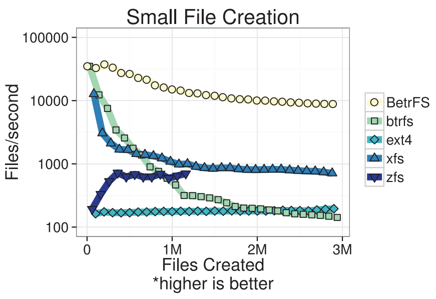


Fig. 6. Sustained rate of file creation for 3 million 200-byte files, using 4 threads. Higher is better.

including BTRFS, ext4, XFS, and ZFS. Error bars and ± ranges denote 95% confidence intervals.

7.1. Microwrites

We evaluated microwrite performance using both meta-data and data-intensive microbenchmarks. To exercise file creation, we used two configurations of the TokuBench benchmark [Esmet et al. 2012]. In the first configuration, shown in Figure 5, we created 500,000 200-byte files in a balanced directory tree with a fanout of 128. TokuBench also measures the scalability of the file system as threads are added; in this experiment, we measured up to four threads since our machine has four cores.

B*e*trFS exhibited substantially higher cumulative throughput than the other file systems. The closest competitor was ZFS at one 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.

The second configuration of the TokuBench benchmark is shown in Figure 6. This setup demonstrates the sustained rate of file creation. We again create 200-byte files in a balanced directory tree with fanout of 128, but for this experiment, we create 3 million files. Each data point represents the cumulative throughput at the time the i^{th} file was created. Four threads were used for all file systems. In the case of ZFS, the file system crashed before completing the benchmark, so we reran the experiment five times and used data from the longest running iteration. B*e*trFS is initially among the fastest file systems and continues to perform well for the duration of the experiment. The steady-state performance of B*e*trFS is an order of magnitude faster than the other file systems.

This performance distinction is attributable to both fewer total writes and fewer seeks per byte written—that is, better aggregation of small writes. Based on profiling from blktrace, one major distinction is total bytes written: B*e*trFS writes 4–10× fewer total MB to disk, with 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 V lists the results. B*e*trFS was 2 orders of magnitude faster than the other file systems.

These results demonstrate that B*e*trFS improves microwrite performance by 1 to 2 orders of magnitude compared to current general-purpose file systems.

Table V. Random Write Performance
Time in seconds to execute 1,000 4-byte microwrites within a 1GiB file. Lower is better.

File System	Time (s)
BetrFS	0.17 ± 0.01
`ext4`	11.60 ± 0.39
`XFS`	11.71 ± 0.28
`BTRFS`	11.60 ± 0.38
`ZFS`	14.75 ± 1.45

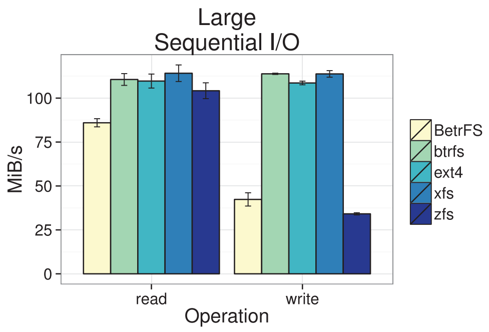


Fig. 7. Large file I/O. We sequentially read and write 1GiB files. Higher is better.

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 BetrFS. In this experiment, BetrFS benefits from compressing keys, but not data. We note that with compression and moderately compressible data, BetrFS can easily exceed disk bandwidth. The results are illustrated in Figure 7.

Most general-purpose file systems can read and write at disk bandwidth. In the case of a large sequential reads, BetrFS 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 retuning the TokuDB block cache prefetching behavior.

In the case of large writes, the current BetrFS prototype achieved just below half of the disk's throughput. One reason for this is that all writes are written twice—once to the log and once to the B^{ε}-tree. Another reason 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. We leave optimizing these cases for future work.

7.3. Directory Operations

In this section, we measure the impact of the BetrFS design on large directory operations. Table VI 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

Table VI. Directory Operation Benchmarks
Time is measured in seconds. Lower is better.

FS	find	grep	dir rename	delete
B*e*trFS	0.36 ± 0.06	3.95 ± 0.28	21.17 ± 1.01	46.14 ± 1.12
BTRFS	3.87 ± 0.94	14.91 ± 1.18	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

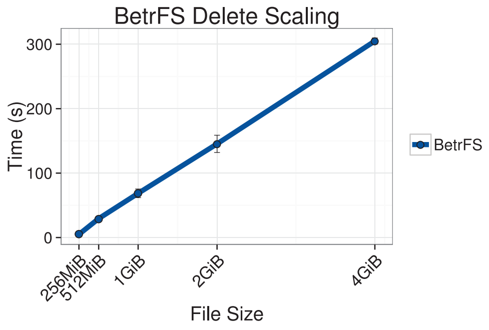


Fig. 8. Time, in seconds, to delete a file in B*e*trFS. Lower is better. We observe a linear relationship between the time to delete a file and its size.

Table VII. Average Time in Cycles to Execute a Range of Common File System Calls
Lower is better.

FS	chmod	mkdir	open	read	stat	unlink	write
B*e*trFS	4,913 ± 0	67,072 ± 26	1,697 ± 0	561 ± 0	1,076 ± 0	47,873 ± 8	32,142 ± 4
BTRFS	4,574 ± 0	24,805 ± 14	1,812 ± 0	561 ± 0	1,258 ± 0	26,131 ± 1	3,891 ± 0
ext4	4,970 ± 0	41,478 ± 19	1,886 ± 0	556 ± 0	1,167 ± 0	16,209 ± 0	3,359 ± 0
XFS	5,342 ± 0	73,782 ± 19	1,757 ± 0	1,384 ± 0	1,134 ± 0	19,124 ± 0	9,192 ± 0
ZFS	36,449 ± 118	171,080 ± 308	2,681 ± 0	6,467 ± 0	1,913 ± 0	78,946 ± 7	18,382 ± 0

on disk. B*e*trFS can search directory metadata and file data 1 or 2 orders of magnitude faster than the other file systems.

The B*e*trFS schema was designed to map VFS operations to efficient B$^\varepsilon$-tree operations whenever possible. Unfortunately, there are some VFS operations that do not map to an individual B$^\varepsilon$-tree operation. Section 7.3 explains that directory deletes and renames require a B$^\varepsilon$-tree operation for each data and metadata entry involved in the operation. Figure 8 shows measurements of file deletions of increasing sizes, showing that the cost is indeed linear in the number of file data blocks. This is because, for each data block, B*e*trFS issues an individual B$^\varepsilon$-tree delete followed by a delete for the file metadata.

Both the rename and delete tests show the worst-case behavior of B*e*trFS. Because B*e*trFS does not include a layer of indirection from pathname to data, renaming requires copying all data and metadata to new points in the tree. 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 VII shows timings for a nanobenchmark that measures various system call times. Because this nanobenchmark is warm-cache, it primarily exercises the VFS

Table VIII. B*ε*trFS Disk Usage, Measured in GiB, after Writing Large Incompressible Files, Deleting Half of Those Files, and Flushing B^{ε}-tree Nodes

	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

layer. B*ε*trFS is close to being the fastest file system on `open`, `read`, and `stat`. On `chmod`, `mkdir`, and `unlink`, B*ε*trFS 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 B*ε*trFS 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

This section details space overheads, both in terms of disk and memory.

The Fractal Tree index implementation in B*ε*trFS includes an LRU cache of B^{ε}-tree nodes. Node-cache memory is bounded. B*ε*trFS triggers background flushing of dirty nodes 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 node-cache memory had little performance impact on our workloads.

The interaction between the Linux page cache and the B^{ε}-tree node cache is complex. As explained in Section 5.1, B*ε*trFS rarely writes dirty cached pages, and they may be easily reclaimed under memory pressure. Writes to uncached data are inserted directly into the B^{ε}-tree. However, the Linux page cache will often double-cache B*ε*trFS data that are read because those data are retrieved from one or more cached B^{ε}-tree nodes. And although the B^{ε}-tree node cache size is bounded, the page cache size is not. From the perspective of B*ε*trFS, the page cache is largely a read accelerator, and page cache pages can be quickly and easily reclaimed.

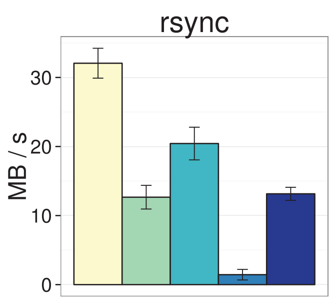
No single rule governs B*ε*trFS disk usage because stale data may remain in nonleaf nodes after delete, rename, and overwrite operations. Background cleaner threads attempt to flush pending data from five internal nodes per second. This creates fluctuation in B*ε*trFS disk usage, but overheads swiftly decline at rest.

To evaluate the B*ε*trFS 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 B*ε*trFS disk usage using `df` on the underlying `ext4` partition.

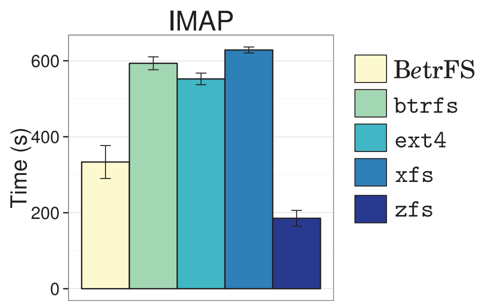
Writing new data to B*ε*trFS introduced very little overhead, as seen in Table VIII. For deletes, however, B*ε*trFS issues an upsert for every file block, which had little impact on the B*ε*trFS footprint because stale data are lazily reclaimed. After flushing, there was less than 3GiB of disk overhead,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

7.6. Application Performance

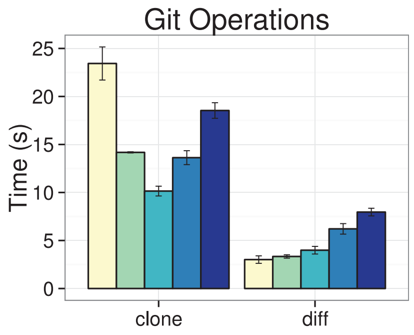
Figure 9 presents performance measurements for a variety of metadata-intensive applications. We measured the time to `rsync` the Linux 3.11.10 source code to a new directory on the same file system using the `--in-place` option to avoid temporary file creation (Figure 9a). We performed a benchmark using 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.



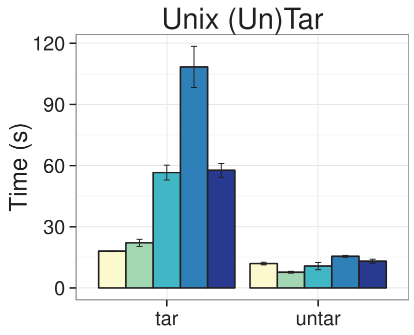
(a) `rsync` of Linux 3.11.10 source. The data source and destination are within the same partition and file system. Throughput in MB/s, higher is better.



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



(c) Git operations. The 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.

Fig. 9. Application benchmarks.

The balance of requests was 50% reads, 50% flags or moves (Figure 9b). We exercised the git version control system using a `git-clone` of the local BetrFS source tree and a `git-diff` between two milestone commits (Figure 9c). Finally, we measured the time to `tar` and un-`tar` the Linux 3.11.10 source (Figure 9d).

BetrFS 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 BetrFS 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 more than 26K `fsync()` calls over 334 seconds, each forcing a full BetrFS log flush. `rsync` on BetrFS realized significantly higher throughput because writing a large number of modestly sized files in lexicographic order is, on BetrFS, 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 BetrFS 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 BetrFS 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, BetrFS can deliver substantial performance benefits.

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

In general, we expect applications that perform small, random, or unaligned writes to see substantial performance gains on B*e*trFS. Applications that issue frequent `sync` operations should also benefit from its ability to persist new data quickly. Applications that are bandwidth-bound will continue to be bandwidth-bound, but may run more slowly due to B*e*trFS logging. The only applications that we expect to perform poorly are those that issue extraneous queries. This behavior has little consequence in a system where each write is preceded by a read, but the read-write asymmetry of write-optimization renders such behavior insensible.

8. RELATED WORK

Previous write-optimized file systems. TokuFS [Esmet et al. 2012] is an in-application library file system also built using B$^{\varepsilon}$-trees. TokuFS showed that a write-optimized file system can support efficient write- and scan-intensive workloads. TokuFS had a FUSE-based version, but the authors explicitly omitted disappointing measurements using FUSE.

KVFS [Shetty et al. 2013] is based on a transactional variation of an LSM-tree called a VT-tree. Impressively, the performance of this 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 [Ren and Gibson 2013] uses LevelDB to store file-system metadata. The authors 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 1,000× 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$^{\varepsilon}$-trees than LSMs in some cases, we expect that the performance of an LSM-based file system will not be completely comparable.

Other WODS. COLAs [Bender et al. 2007] are an LSM tree variant that uses fractional cascading [Chazelle and Guibas 1986] to match the performance of B$^{\varepsilon}$-trees for both insertions and queries, but we are not aware of any full featured, production-quality COLA implementation. xDict [Brodal et al. 2010] is a cache-oblivious WODS with asymptotic behavior similar to a B$^{\varepsilon}$-tree.

Key-Value Stores. WODS are widely used in key-value stores, including BigTable [Chang et al. 2008], Cassandra [Lakshman and Malik 2010], HBase [Apache 2015b], LevelDB [Google, Inc. 2015], TokuDB [Tokutek, Inc. 2013a], and TokuMX [Tokutek, Inc. 2013b]. BigTable, Cassandra, and LevelDB use LSM-tree variants. TokuDB and TokuMX use Fractal Tree indexes. LOCS [Wang et al. 2014] optimizes LSM-trees for a key-value store on a multichannel SSD.

Instead of using WODS, FAWN [Andersen et al. 2009] writes to a log and maintains an in-memory index for queries. SILT [Lim et al. 2011] further reduces the design's memory footprint during the merging phase.

Alternatives to update-in-place. No-overwrite (copy-on-write) file systems perform in-memory modifications to existing file blocks before writing them persistently at new physical locations. Thus, file data can be logically, but not physically, overwritten. Log-structured file systems are one class of no-overwrite file system with a very specific data placement policy, but other no-overwrite data placement policies also exist.

Log-structured File Systems and their derivatives [Andersen et al. 2009; Lee et al. 2015; Lim et al. 2011; Rosenblum and Ousterhout 1992] 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.

BTRFS, WAFL, and ZFS are feature-rich no-overwrite file systems whose policies leverage copy-on-write semantics to implement advanced features like versioning, self-consistency, and checkpoints [Bonwick 2004; Hitz et al. 1994; Rodeh et al. 2013]. The Write Anywhere File Layout (WAFL) uses files to store its metadata, giving it incredible flexibility in its block allocation and layout policies [Hitz et al. 1994]. One of WAFL's main goals is to provide efficient copy-on-write snapshots. BTRFS uses copy-on-write B-trees throughout, providing both versioning and checkpoints. BTRFS also implements back-references to help with defragmentation. ZFS rethinks the division of labor within the storage stack and seeks to ease both the use and management of file systems. ZFS ensures that all data on disk are self-consistent, provides pooled storage, and supports numerous features including deduplication. All three of these file systems provide advanced features beyond what traditional file systems offer, but none addresses the microwrite problem.

The no-overwrite mechanism is powerful and flexible. However, it is impossible to maintain physical locality in the presence of versions and updates since file blocks can be physically adjacent in exactly one file version. Extra bookkeeping for back-references can help to aid defragmentation in exchange for added complexity [Macko et al. 2010].

Block indexing. As discussed in Section 5.5, B*e*trFS does not store location information in its inodes. Instead, it relies on queries to the WODS to transparently retrieve its data. Alternative techniques for indexing file data include extents and direct pointers to blocks.

For large files with few updates, extents are very space efficient. A single range can represent the location information for an entire file, regardless of size. However, extents introduce complexity when files become fragmented, which can occur for various reasons: insufficient contiguous free space to fit an entire file, copy-on-write updates to a file, or file versioning. In these cases, many extents must be managed per file, and logic for merging and splitting ranges must be introduced.

Block pointers are simple and flexible, with a single pointer for each file block. Reading or writing a file requires following each pointer. Fragmentation does not impact file representation in pointer-based indexing schemes.

Logical logging. This 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 [Gray and Reuter 1993]. 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$^{\varepsilon}$-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 data structures are a powerful tool for file-system developers. In some cases, B*e*trFS outperforms 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$^\varepsilon$-tree implementation from which we built BetrFS. We thank the anonymous reviewers and Geoff Kuenning for their insightful comments on earlier drafts of the work.

REFERENCES

Alok Aggarwal and Jeffrey Scott Vitter. 1988. The input/output complexity of sorting and related problems. *Communications of the ACM* 31, 9 (Sept. 1988), 1116–1127.

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

Apache. 2015a. Accumulo. Retrieved May 16, 2015 from http://accumulo.apache.org.

Apache. 2015b. HBase. Retrieved May 16, 2015 from http://hbase.apache.org.

Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious streaming b-trees. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 81–92.

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

Michael A. Bender, Haodong Hu, and Bradley C. Kuszmaul. 2010. Performance guarantees for b-trees with different-sized atomic keys. In *Proceedings of the 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'10)*. 305–316.

John Bent, Garth A. Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. 2009. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC'09)*. 1–12.

Jeff Bonwick. 2004. ZFS: The Last Word in File Systems. Retrieved from https://blogs.oracle.com/video/entry/zfs_the_last_word_in.

Gerth Stølting Brodal, Erik D. Demaine, Jeremy T. Fineman, John Iacono, Stefan Langerman, and J. Ian Munro. 2010. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1448–1456.

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

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

Rémy Card, Theodore Ts'o, and Stephen Tweedie. 1994. Design and implementation of the second extended filesystem. In *Proceedings of the 1st Dutch International Symposium on Linux*. 1–6.

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

Bernard Chazelle and Leonidas J. Guibas. 1986. Fractional cascading: I. A data structuring technique. *Algorithmica* 1, 1–4, 133–162.

Douglas Comer. 1979. The ubiquitous B-tree. *ACM Computing Surveys* 11, 2 (June 1979), 121–137.

David Douthitt. 2011. Instant 10-20% Boost in Disk Performance: The "Noatime" Option. Retrieved from http://administratosphere.wordpress.com/2011/07/29/instant-10-20-boost-in-disk-performance-the-noatime-option/.

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

FUSE. 2015. File System in Userspace. Retrieved May 16, 2015 from http://fuse.sourceforge.net/.

Google, Inc. 2015. LevelDB: A fast and lightweight key/value database library by Google. Retrieved May 16, 2015 from http://github.com/leveldb/.

Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

Dave Hitz, James Lau, and Michael Malcolm. 1994. *File System Design for an NFS File Server Appliance*. Technical Report. NetApp.

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

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2, 35–40.

Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'13)*. 273–286.

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

Peter Macko, Margo Seltzer, and Keith A. Smith. 2010. Tracking back references in a write-anywhere file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'13)*. 15–28.

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

QuickLZ. 2015. Fast Compression Library for C, C#, and Java. Retrieved May 16, 2015 from http://www.quicklz.com/.

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

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

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

Russell Sears, Mark Callaghan, and Eric A. Brewer. 2008. Rose: Compressed, log-structured replication. *Proceedings of the VLDB Endowment* 1, 1, 526–537.

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

Margo Seltzer, Keith Bostic, Marshall Kirk Mckusick, and Carl Staelin. 1993. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Conference Proceedings*. 3.

Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. 1995. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*. 21.

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

Adam Sweeny, Doug Doucette, Wwei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Technical Conference*. CA, 1–14.

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

Tokutek, Inc. 2013b. TokuMX—MongoDB Performance Engine. Retrieved from http://www.tokutek.com/products/tokumx-for-mongodb/.

Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. 16:1–16:14.

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

Received May 2015; accepted June 2015