BetrFS: A right-optimized, write-optimized file system

William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter

Stony Brook University, Tokutek Inc., Rutgers University, Massachusetts Institute of Technology
ext4 is good at sequential I/O

- Disk bandwidth spec: 125 MB/s
- Workload: 1GiB sequential write
- ext4 bandwidth: 104 MB/s
ext4 struggles with random writes

- Disk bandwidth spec: 125 MB/s
- Workload: Small, random writes of cached data
- **ext4** write bandwidth:
  - 1.5 MB/s
What is going on here?

• Random write performance dominated by seeks

• Back-of-the-envelope:
  ▪ Average disk seek time is 11ms
  ▪ Seek for every 4KB write
  ▪ Implies maximum 0.4MB/s bandwidth
    • Previous benchmark benefits from locality, good I/O scheduling
Avoiding seeks: log-structured file systems

• Pros:
  ▪ writing data is just an append to the log

• Cons:
  ▪ file blocks can become scattered on disk
  ▪ reading data becomes slow

• Logging still presents a tradeoff between random-write and sequential-I/O performance
BetrFS

• Use **write-optimized indexes (WOIs)**
  - on-disk data structures that rapidly ingest new data while maintaining logical locality

• Create a schema that maps file operations to efficient WOI operations

• Implemented in the Linux kernel
  - exposed new performance opportunities
Advancing write-optimized FSes

• Prior work: WOIs can accelerate FS operations
  ▪ TokuFS [Esmet, Bender, Farach-Colton, Kuszmaul ‘12], KVFS [Shetty, Spillane, Malpani, Andrews, Seyster, and Zadok ‘13], TableFS [Ren and Gibson ‘13],
  ▪ Prior WOFs in user space
• BetrFS goal: explore all the ways write-optimization can be used in a file system
  ▪ explore the impact of write-optimization on the interaction with the rest of the system
BetrFS uses $B^\varepsilon$-Trees

- $B^\varepsilon$-trees: an asymptotically optimal key-value store
- $B^\varepsilon$-trees asymptotically dominate log-structured merge-trees
- We use Fractal Trees, an open-source $B^\varepsilon$-tree implementation from Tokutek

For this talk, we treat $B^\varepsilon$ as a black box that performs fast insertions and fast point and range queries
**B^φ-Tree Operations**

- Implement a dictionary on key-value pairs
  - `insert(k, v)`
  - `v = search(k)`
  - `delete(k)`
  - `k' = successor(k)`
  - `k' = predecessor(k)`

- New operation:
  - `upsert(k, f)`

get, put, and delete elements one-at-a-time
query a range of values
\( B^\varepsilon \)-trees search/insert asymmetry

- Queries (point and range) comparable to B-trees
  - with caching, ~1 seek + disk bandwidth
  - hundreds of random queries per second
- Extremely fast inserts
  - tens of thousands per second

To get the best possible performance, we want to do blind inserts (without searches)
**upsert = update + insert**

\[ \text{upsert}(k, f) \]

- An *upsert* specifies a **mutation** to a value
  - e.g. increment a reference count
  - e.g. modify the 5\(^{\text{th}}\) byte of a string
- *upserts* are encoded as messages and inserted into the tree
  - defer and batch expensive queries
  - we can perform tens of thousands of *upserts* per second
File System $\rightarrow$ $B^\varepsilon$ Tree

• Maintain two separate $B^\varepsilon$-tree indexes:

  metadata index: $\text{path} \rightarrow \text{struct stat}$

  data index: $(\text{path}, \text{blk}\#) \rightarrow \text{data}[4096]$

• Implications:
  - fast directory scans
  - data blocks are laid out sequentially
# Operation Roundup

<table>
<thead>
<tr>
<th>Operation</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>range query</td>
</tr>
<tr>
<td>write</td>
<td>upsert</td>
</tr>
<tr>
<td>metadata update</td>
<td>upsert</td>
</tr>
<tr>
<td>readdir</td>
<td>range query</td>
</tr>
<tr>
<td>mkdir/rmdir</td>
<td>upsert</td>
</tr>
<tr>
<td>unlink</td>
<td>*delete each block</td>
</tr>
<tr>
<td>rename</td>
<td>*delete then</td>
</tr>
<tr>
<td></td>
<td>reintert each block</td>
</tr>
</tbody>
</table>

- **Fast atime**
- **Efficient directory scans**
- **cannot map to single WOI operation**
Integrating BetrFS with the page cache

- Write-back caching can convert single-byte to full-page writes
- upserts enable BetrFS to avoid this write amplification
Page cache integration #1: blind write

write(/home/bill/foo.txt, [])

Is the target page cached?

upsert(/home/bill/foo.txt, [])

Page cache

/home/bill/foo.txt

No cached page.

upsert(/home/bill/foo.txt, [])
Page cache integration #2: write-after-read

write(/home/bill/foo.txt, [])

Is the target page cached?

Is the target page dirty?

upsert(/home/bill/foo.txt, [])

Page cache

/home/bill/foo.txt

Target page is cached.

Target page is clean.

Cached page is now consistent with disk.

upsert(/home/bill/foo.txt, [])
Page cache integration #3: write to mmap’ed file

```c
write(/home/bill/foo.txt, [] )
```

- Is the target page cached?
- Is the target page dirty?

Page cache is cached.

Target page is dirty.

We wait for page writeback to persist our new data.
Page-cache takeaways

• By rethinking the interaction between the page cache and the file system, we benefit more than simply speeding up individual operations
  ▪ use upserts to avoid unnecessary reads
  ▪ use upserts to avoid write amplification
System Architecture

- BetrFS Kernel module registered with the VFS
- Imported as a binary blob
- Use an existing file system as block manager

Diagram:

- VFS
- BetrFS
- B\textsuperscript{e} Tree
- ext4
- Page Cache
- Disk

Legend:
- unmodified*
- new code
Performance Questions

- Do we meet our performance goals for small, random, unaligned writes?
- Is BetrFS competitive for sequential I/O?
- Do any real-world applications benefit?
Experimental Setup

- Dell optiplex desktop:
  - 4-core 3.4 GHz i7, 4 GB RAM
  - 7200RPM 250GB Seagate Barracuda

- Compare with btrfs, ext4, xfs, zfs
  - default settings for all

- All tests are cold cache
Small, random, unaligned writes are an order-of-magnitude faster

- 1 GiB file, random data
- 1,000 random 4-byte writes
- `fsync()` at end

1000 Random 4-byte writes

*lower is better

Log Scale

0.17s vs. > 10s

BetrFS benefits from blind and sub-block writes
Small file creates are an order-of-magnitude faster

- create 3 million files and write 200-bytes to each balanced directory tree with fanout 128
- performance over time

Small File Creation

Files/second

- higher is better

Files Created

BetrFS
btrfs
ext4
xfs
zfs

Log Scale

After creating the 1 millionth file, what is the throughput
Sequential I/O

- Write random data to file, 10 4K-blocks at a time
- Sequentially read data back

Write all data at least 2x (B^e-tree journaling)
BetrFS forgoes indirection for locality: delete, rename $O(n)$

- write random data to file, `fsync()` it
- delete file

### BetrFS Delete Scaling

![Graph showing BetrFS Delete Scaling](image)

**File Size**

- 256MiB
- 512MiB
- 1GiB
- 2GiB
- 4GiB

**Time (s)**

- 0
- 100
- 200
- 300

**O(n) scaling:**

must delete each block individually
BetrFS forgoes indirection for locality: fast directory scans

- recursive scans from root of Linux 3.11.10 source
- GNU `find` scans file metadata
- `grep -r` scans file contents

full-path keys let BetrFS efficiently implement scans using range queries
BetrFS Benefits Mailserver Workloads

IMAP (50% read, 50% mark or move)

- Dovecot 2.2.13 mail server using `maildir`
- 26,000 `sync()` operations

*lower is better*
BetrFS Benefits rsync

- rsync Linux source tree to new directory on same FS
- copying to an empty directory

---in-place flag lets BetrFS issue blind writes
Performance Questions

• Do we meet our performance goals for small, random writes?

• Is BetrFS competitive for sequential I/O?
  * More work to do here

• Do any real-world applications benefit?
  * More experiments in paper
BetrFS

• Cake & Eat: One file system can have good sequential and random I/O performance
• WOI performance requires revisiting many design decisions
  ▪ inodes
  ▪ write-through vs. write-back caching
  ▪ perform blind writes whenever possible

betrfs.org – github.com/oscarlab/betrfs