How to Copy Files

Yang Zhan, *The University of North Carolina at Chapel Hill and Huawei;* Alexander Conway, *Rutgers University;* Yizheng Jiao and Nirjhar Mukherjee, *The University of North Carolina at Chapel Hill;* Ian Groombridge, *Pace University;* Michael A. Bender, *Stony Brook University;* Martin Farach-Colton, *Rutgers University;* William Jannen, *Williams College;* Rob Johnson, *VMWare Research;* Donald E. Porter, *The University of North Carolina at Chapel Hill;* Jun Yuan, *Pace University*

https://www.usenix.org/conference/fast20/presentation/zhan

This paper is included in the Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-12-0

Open access to the Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20) is sponsored by

How to Copy Files

Yang Zhan
UNC Chapel Hill and Huawei

Alex Conway
Rutgers Univ.

Yizheng Jiao
UNC Chapel Hill

Nirjhar Mukherjee
UNC Chapel Hill

Ian Groombridge
Pace Univ.

Michael A. Bender
Stony Brook Univ.

Martín Farach-Colton
Rutgers Univ.

William Jannen
Williams College

Rob Johnson
VMware Research

Donald E. Porter
UNC Chapel Hill

Jun Yuan
Pace Univ.

Abstract

Making logical copies, or clones, of files and directories is critical to many real-world applications and workflows, including backups, virtual machines, and containers. An ideal clone implementation meets the following performance goals: (1) creating the clone has low latency; (2) reads are fast in all versions (i.e., spatial locality is always maintained, even after modifications); (3) writes are fast in all versions; (4) the overall system is space efficient. Implementing a clone operation that realizes all four properties, which we call a *nimble clone*, is a long-standing open problem.

This paper describes nimble clones in BetrFS, an open-source, full-path-indexed, and write-optimized file system. The key observation behind our work is that standard copy-on-write heuristics can be too coarse to be space efficient, or too fine-grained to preserve locality. On the other hand, a write-optimized key-value store, as used in BetrFS or an LSM-tree, can decouple the logical application of updates from the granularity at which data is physically copied. In our write-optimized clone implementation, data sharing among clones is only broken when a clone has changed enough to warrant making a copy, a policy we call *copy-on-abundant-write*.

We demonstrate that the algorithmic work needed to batch and amortize the cost of BetrFS clone operations does not erode the performance advantages of baseline BetrFS; BetrFS performance even improves in a few cases. BetrFS cloning is efficient; for example, when using the clone operation for container creation, BetrFS outperforms a simple recursive copy by up to two orders-of-magnitude and outperforms file systems that have specialized LXC backends by 3–4×.

1 Introduction

Many real-world workflows rely on logically copying files and directories. Backup and snapshot utilities logically copy the entire file system on a regular schedule [36]. Virtual-machine servers create new virtual machine images by copying a pristine disk image. More recently, container infrastructures like Docker make heavy use of logical copies to package and deploy applications [34, 35, 37, 44], and new container creation typically begins by making a logical copy of a reference directory tree.

Duplicating large objects is so prevalent that many file systems support *logical* copies of directory trees without making full *physical* copies. Physically copying a large file or directory is expensive—both in time and space. A classic optimization, frequently used for volume snapshots, is to implement copy-on-write (CoW). Many logical volume managers support block-level CoW snapshots [24], and some file systems support CoW file or directory copies [29] via `cp --reflink` or other implementation-specific interfaces. Marking a directory as CoW is quick, especially when the file system can mark the top-level directory as CoW and lazily propagate the changes down the directory tree. Initially, this approach is also space efficient because blocks or files need not be rewritten until they are modified. However, standard CoW presents a subtle tradeoff between write amplification and locality.

The main CoW knob to tune is the copy granularity. If the copy granularity is large, such as in file-level CoW, the cost of small changes is amplified; the first write to any CoW unit is high, drastically increasing update latency, and space is wasted because sharing is broken for *all* data. If the copy granularity is small, updates are fast but fragmented; sequentially reading the copy becomes expensive. Locality is crucial: poor locality can impose a persistent tax on the performance of all file accesses and directory traversals until the file is completely rewritten or the system defragmented [8–10].

Nimble clones. An ideal logical copy—or ***clone***—implementation will have strong performance along several dimensions. In particular, clones should:

- be fast to create;
- have excellent read locality, so that logically related files can be read at near disk bandwidth, even after modification;
- have fast writes, both to the original and the clone; and
- conserve space, in that the write amplification and disk footprint are as small as possible, even after updates to the original or to the clone.

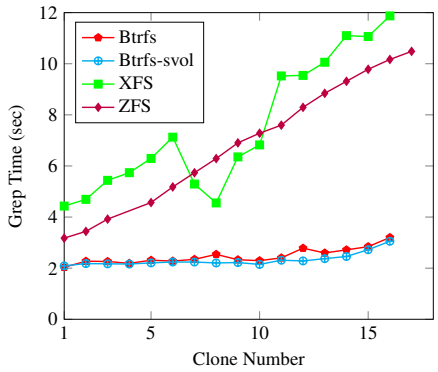


Figure 1: Grep Time for a logically copied 256MiB directory, as a function of the number of prior copies with small edits. (Lower is better.) Btrfs-svol is a volume snapshot, Btrfs and XFS use `cp --reflink`. Full experiment details are in §5.1.

We call a clone with this constellation of features ***nimble***. Existing CoW clone implementations are not nimble.

Figure 1 illustrates how performance can degrade using standard CoW techniques in two file systems with copy optimizations. We start by creating a two-level directory hierarchy with 64 4-MiB files (256MiB total), and the experiment proceeds for several rounds. Each round does a volume snapshot or a reflink copy (depending on what the file system supports) and then performs a small, 16-byte edit to each file. We report the time to do a recursive, cold-cache grep over the entire directory at the end of each round. The experiment is detailed further in §5.1.

After each copy and modification, read performance degrades. In the case of XFS and ZFS, we see a factor of 3–4× after only 16 rounds. Btrfs degrades more gradually, about 50% over the same period. In both cases, however, the degradation appears monotonic.

The critical issue here is the need to decouple the granularity of *writes* to a clone from the granularity of *copies* of the shared data. It makes perfect sense to copy a large file that is effectively overwritten. But, for very small changes, it is more IO efficient to keep a "delta" in scratch space until enough changes accrue to justify the cost of a substantial rewrite. In other words, the CoW *copy size* should be tuned to preserve locality (e.g., set to an efficient transfer size for the device), not to whatever granularity a single workload happens to use.

Contributions. In this paper, we present a logical copy specification, which we call a clone, and a set of performance criteria that a *nimble* clone must satisfy. We present the design for a file system and nimble clone implementation that meets all of these criteria.

One key insight into our solution is that the write-optimized message-batching model used in systems such as BetrFS is well suited to decouple writes from copies. There is already a mechanism in place to buffer and apply small changes, although implement the semantics of cloning requires substantial, additional data-structural work.

We extend BetrFS 0.4, an open-source, full-path-indexed, write-optimized file system. BetrFS performance matches or exceeds other local Linux file systems on a range of applications [8, 17, 39, 42], but BetrFS 0.4 does not support cloning. BetrFS 0.5's clone implements a policy we call ***Copy-on-Abundant-Write***, or ***CAW***, by buffering small changes to a cloned file or directory in messages until enough changes accrue to warrant the cost of unsharing the cloned data.

This paper also contributes several data-structural techniques to write-optimized dictionaries, in order to implement nimble clones in BetrFS. First, we enable different traversal paths to re-use the same physical data by transforming BetrFS's B$^\varepsilon$-tree [3, 6] data structure into a B$^\varepsilon$-DAG (directed acyclic graph). Second, in order to realize very fast logical copies, we develop new techniques that apply write-optimization, which has previously been used to accelerate changes to *data* stored in the key-value store, towards batching changes to the *topology* of the data structure itself, i.e., its *pivots* and *internal pointers*. An essential limitation of the state of the art, including BetrFS, is that renames, which modify the tree structure, cannot be batched; rather, renames must be completed immediately, including applying all pending changes to the relevant portions of the file system namespace. We introduce a ***GOTO message***, which can rapidly persist a logical copy into the message buffer, and is as fast as any small write. With `GOTOs`, B$^\varepsilon$-DAG-internal housekeeping work is piggy-backed onto any writes to the logically copied region. Third, we introduce a ***translation prefix*** abstraction that can—at rest—logically correct stale keys in shared data, facilitating both deferred copies and correct queries of partially shared data. As a result of these techniques, BetrFS can rapidly persist large logical copies much faster than the current state of the art (33%–6.8×), without eroding read, write, or space efficiency.

The contributions of this paper are as follows:

- A design and implementation of a B$^\varepsilon$-DAG data structure, which supports nimble CAW clones. The B$^\varepsilon$-DAG extends the B$^\varepsilon$-tree buffered-message substrate to store and logically apply small changes to a clone, until enough changes accrue to warrant the cost of rewriting a clone.
- A write-optimized clone design, wherein one can persist a clone by simply writing a message into the root of the DAG. The work of the clone is batched with other operations and amortized across other modifications.
- An asymptotic analysis, indicating that adding cloning does not harm other operations, and that cloning itself has a cost that is logarithmic in the size of the B$^\varepsilon$-DAG.
- A thorough evaluation of BetrFS, which demonstrates that it meets the nimble performance goals, does not erode the advantages of baseline BetrFS on unrelated workloads, and can improve performance of real-world applications. For instance, we wrote an LXC (Linux Container) backend that uses cloning to create containers, and BetrFS is 3–4×

faster than other file systems with cloning support, and up to 2 orders of magnitude faster than those without.

2 BetrFS Background

This section presents B$^\varepsilon$-tree and BetrFS background that is necessary to understand the cloning implementation presented in the rest of the paper.

BetrFS [17, 18, 39, 40, 42, 43] is an in-kernel, local file system built on a key-value store (KV-store) substrate. A BetrFS instance keeps two KV-stores. The metadata KV-store maps full paths (relative to the mountpoint, e.g., `/foo/bar/baz`) to `struct stat` structures, and the data KV-store maps {full path + block number} keys to 4KiB blocks.

The B$^\varepsilon$-tree. BetrFS is named for its KV-store data structure, the B$^\varepsilon$-tree [3, 6]. A B$^\varepsilon$-tree is a write-optimized KV-store in the same family of data structures as an LSM-tree [25] or COLA [2]. Like B-tree variants, B$^\varepsilon$-trees store key-value pairs in leaves. A key feature of the B$^\varepsilon$-tree is that interior nodes buffer pending mutations to the leaf contents, encoded as *messages*. Messages are inserted into the root of the tree, and, when an interior node's buffer fills with messages, messages are *flushed* in large batches to one or more children's buffers. Eventually, messages reach the leaves and the updates are applied. As a consequence, random updates are inexpensive—the B$^\varepsilon$-tree effectively logs updates at each node. And since updates move down the tree in batches, the IO savings grow with the batch size.

A key B$^\varepsilon$-tree invariant is that all pending messages for a given key-value pair are located on the root-to-leaf traversal path that is defined by its key. So a point query needs to read and apply all applicable buffered messages on the traversal path to construct a correct response. Messages have a logical timestamp, and one can think of the contents of these buffered messages as a history of mutations since the last time the leaf was written.

Range operations. BetrFS includes optimizations for contiguous ranges of keys. These are designed to optimize operations on subtrees of the file system namespace (e.g., `mv`).

Importantly, because BetrFS uses full-path keys, the contents of a directory are encoded using keys that have a common prefix and thus are stored nearly contiguously in the B$^\varepsilon$-tree, in roughly depth-first order. One can read a file or recursively search a directory with a *range query* over all keys that start with the common directory or file prefix. As a result, BetrFS can use a *range delete* message to delete an entire file or recursively (and logically) delete a directory tree with a single message. The range delete is lazily applied to physically delete and recover the space.

Full-path indexing and renaming. Efficient `rename` operations pose a significant challenge for full-path-indexed file systems. BetrFS has a `range rename` operation, which can *synchronously* change the prefix of a contiguous range of keys in the B$^\varepsilon$-tree [42]. In a nutshell, this approach *slices* out the source and destination subtrees, such that there is a single pointer at the same B$^\varepsilon$-tree level to the source and destination subtrees. The range rename then does a "pointer swing", and the tree is "healed" in the background to ensure balance and that nodes are within the expected branching factor. Some important performance intuition about this approach is that the slicing work is logarithmic in the size of the renamed data (i.e., the slicing work is only needed on the right and left edge of each subtree).

BetrFS ensures that range rename leaves most of the *on-disk* subtree untouched by ***lifting*** out common key prefixes. Consider a subtree T whose range is defined at T's parent by pivots p_1 and p_2. Then the longest common prefix of p_1 and p_2, denoted $\text{lcp}(p_1, p_2)$, must be a prefix of all the keys in T. A lifted B$^\varepsilon$-tree omits $\text{lcp}(p_1, p_2)$ from all keys in T. We say that $\text{lcp}(p_1, p_2)$ has been ***lifted out of*** T, and that lcp-T is lifted. The lifted B$^\varepsilon$-tree maintains the lifting invariant, i.e. that every subtree is lifted at all times. Maintaining the lifting invariant does not increase the IO cost of insertions, queries, flushes, node splits or merges, or any other B$^\varepsilon$-tree operations.

With the combination of tree surgery and lifting, BetrFS renames are competitive with inode-based file systems [42].

Crash consistency. BetrFS's B$^\varepsilon$-tree nodes are copy-on-write. Nodes are identified using a logical node number, and a ***node translation table*** maps logical node numbers to on-disk locations. The node translation table also maintains a bitmap of free and allocated disk space. Node writeback involves allocating a new physical location on disk and updating the node translation table. This approach removes the need to modify a parent when a child is rewritten.

All B$^\varepsilon$-tree modifications are logged in a logical redo log. The B$^\varepsilon$-tree is checkpointed to disk every 60 seconds; a checkpoint writes all dirty nodes and the node translation table to disk and then truncates the redo log. After a crash, one need only replay the redo log since the last checkpoint.

Physical space is reclaimed as part of the checkpointing process with the invariant that one can only reuse space that is not reachable from the last stable checkpoint (otherwise, one might not recover from a crash that happens before the next checkpoint). As a result, node reclamation is relatively straightforward: when a node is overwritten, the node translation table tracks the pending free, and then applies that free at the next checkpoint. We note that range delete of a subtree must identify all of the nodes in the subtree and mark them free as part of flushing the range delete message; the node translation table does not store the tree structure.

3 Cloning in BetrFS 0.5

This section describes how we augment BetrFS to support cloning. We begin by defining clone semantics, then describe how to extend the lifted B$^\varepsilon$-tree data structure to a lifted B$^\varepsilon$-

DAG (directed acyclic graph), and finally describe how to perform mutations on this new data structure. The section concludes with a brief asymptotic analysis of the B$^\varepsilon$-DAG.

When considering the design, it helps to differentiate the three layers of the system: the file system directory hierarchy, the KV-store keyspace, and the internal B$^\varepsilon$-tree structure. We first define the clone operation semantics in terms of their effect on file system directory tree. However, because all file system directories and their descendants are mapped onto contiguous KV-store keys based on their full paths, we then focus the BetrFS clone discussion on the KV-store keyspace and the internal B$^\varepsilon$-tree structure implementation.

CLONE operation semantics. A CLONE operation takes as input two paths: (1) a source path—either a file or directory tree root—and (2) a destination path. The file system directory tree is changed so that a logically identical copy of the source object exists at the location specified by the destination path. If a file or directory was present at the destination before the clone, that file or directory is unlinked from the directory tree. The clone operation is atomic.

In the KV-store keyspace, $clone(s,d)$ copies all keys with prefix s to new keys with prefix s replaced with prefix d. It also removes any prior key-value pairs with prefix d.

3.1 Lifted B$^\varepsilon$-DAGs

Our goal in making a lifted B$^\varepsilon$-DAG is to share, along multiple graph traversal paths, a large amount of cloned data, and to do so without immediately rewriting any child nodes. Intuitively, we should be able to immediately add one edge to the graph, and then tolerate and lazily repair any inconsistencies that appear in traversals across that newly added edge. As illustrated in Figure 2, we construct the lifted B$^\varepsilon$-DAG by extending the lifted B$^\varepsilon$-tree in three ways.

First, we maintain reference counts for every node so that nodes can be shared among multiple B$^\varepsilon$-DAG search paths. Reference counts are decoupled from the node itself and stored in the node translation table. Thus, updating a node's reference does not require modifying any node. Whenever a node's reference count reaches zero, we decrement all of its children's reference counts, and then we reclaim the node. Section 4 describes node reclamation.

A significant challenge for sharing nodes in a B$^\varepsilon$-tree or B$^\varepsilon$-DAG is that nodes are large (target node sizes are large enough to yield efficient transfers with respect to the underlying device, typically 2–4MiB) and packed with many key-value pairs, so a given node may contain key-value pairs that belong to unrelated logical objects. Thus, sharing a B$^\varepsilon$-DAG node may share more than just the target data.

For example, in Figure 2, the lower node is the common ancestor of all keys beginning with s, but the subtree rooted at the node also contains keys from q to v. We would like to be able to clone, say, s to p by simply inserting a new edge, with pivots p and pz, pointing to the common ancestor of all

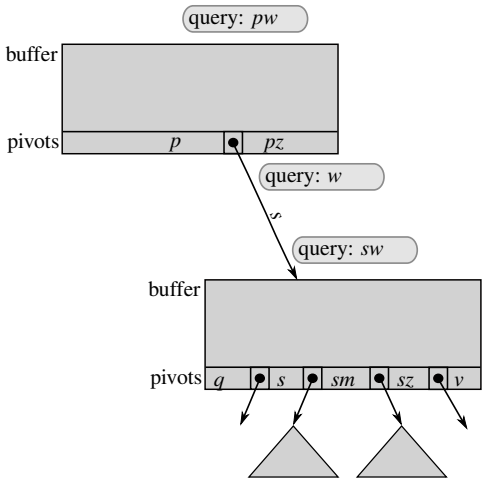


Figure 2: Query processing example in a lifted B$^\varepsilon$-DAG. Initially, the query pw arrives at the parent node. Since the target child's pointer is bracketed by pivots that share the common prefix p (pivots p and pw bracket the pointer to the child), the lifted B$^\varepsilon$-DAG lifts (i.e., removes) the common prefix p from the query term used for searching in the child, transforming the query from pw to w. Next, the query w reaches an edge with translation prefix s. The lifted B$^\varepsilon$-DAG prepends the translation prefix s to the query before continuing to the child. Thus, the query that finally arrives at the child is sw: the common prefix p was lifted out, and the translation prefix s was prepended. The query process proceeds recursively until a terminal node is reached.

s keys but, as the example illustrates, this could have the side effect of cloning some additional keys as well.

Thus, our second major change is to alter the behavior of pivot keys so that they can exclude undesirable keys from traversals. This filtering lets us tolerate unrelated data in a subgraph. A baseline B$^\varepsilon$-tree has an invariant that two pivot keys in a parent node must bound all key-value pairs in their child node (and sub-tree). In the B$^\varepsilon$-DAG, we must relax this invariant to permit node sharing, and we change the graph traversal behavior to simply ignore any key-value pair, message, or pivot that lies outside of the parent pivot keys' range. This partially addresses the issue of sharing a subgraph with extraneous data at its fringe.

The third, related change is to augment each edge with an optional ***translation prefix*** that alters the behavior of traversals that cross the edge. When cloning a source range of keys to a destination, part of the source key may not be lifted. A translation prefix on an edge specifies any remaining part of the source prefix that was not lifted at the time of cloning. As Figure 2 shows, whenever a query crosses an edge with translation prefix s, we prepend s to the query term before continuing to the child, so that the appropriate key-value pairs are found. Once completed, a query removes the translation prefix from any results, before the lifted destination key

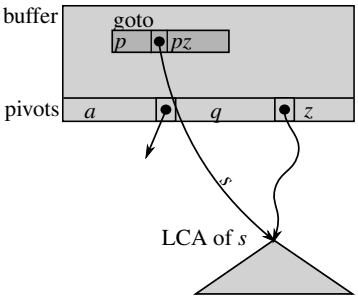


Figure 3: Creating a clone by inserting a GOTO message. Note that the GOTO message's bracketing pivots are (p, pz), and its child pointer contains translation prefix s. The GOTO message supersedes the node's other pivots during a traversal.

along the search path is added back. In the common case, the translation prefix will be NULL.

With these changes—reference counting, filtering pivots, and translation prefixes—a B$^\epsilon$-DAG can efficiently represent clones and share cloned data among different search paths.

3.2 Creating clones with GOTO messages

To clone all keys (and associated data) with prefix s to new keys with prefix p, we first find the lowest-common ancestor (LCA) of all s keys in the B$^\epsilon$-DAG, as shown in Figure 3. Intuitively, the LCA is the root of the lowest sub-graph that includes all source keys. We will call the LCA of all s keys node L_s. We then flush to L_s any pending messages for s keys, so that all information about s keys can be found within the sub-DAG rooted at node L_s. We also insert into the root's buffer a GOTO message (described below) for all p keys with target node L_s. We finally increment the reference count of L_s. This completes the cloning process.

GOTO messages. A GOTO message behaves like a pair of bracketing pivots and an associated child pointer. Each GOTO message specifies a range of keys, (a,b); a target height; and a node, U. Whenever a query for some key x reaches a node with a GOTO message, if x falls in the range (a,b), then the query continues directly to node U; said differently, a node's GOTO message supersedes the node's other pivots during a traversal. Like regular pivots, if the two pivots in a GOTO message share a common prefix, then that prefix is removed (lifted) from the query before continuing. Furthermore, like regular child pointers, the pointer in a GOTO message can specify a translation prefix that gets prepended to queries before they continue. Figure 3 illustrates a simple GOTO example, where s is cloned to p. There is a normal child pointer associated with node pivots that bracket prefix s, as well as a GOTO message that redirects queries for p to the LCA of s. In this example, we assume s has not been lifted from the LCA, and, thus, s is used as a translation prefix on the GOTO message.

Flushing GOTO messages. Unlike a regular pair of pivots that bracket a child pointer, a GOTO message can be flushed from one node to another, just like any other message. Encoding DAG structure inside a message is an incredibly powerful feature: we can quickly persist a logical clone and later batch any post-cloning clean-up work with subsequent writes. When subsequent traversals process buffered messages in logical order, a GOTO takes precedence over all older messages pertaining to the destination keys; in other words, a GOTO implicitly deletes all key-value pairs for the destination range, and redirects subsequent queries to the source sub-graph.

For performance, we ensure that all root-to-leaf B$^\epsilon$-DAG paths have the same length. Maintaining this invariant is important because, together with the B$^\epsilon$-DAG's fanout bounds, it guarantees that the maximum B$^\epsilon$-DAG path has logarithmic length, which means that all queries have logarithmic IO complexity. Thus, we must ensure that paths created by GOTO messages are not longer than "normal" root-to-leaf paths.

This length invariant constrains the minimum height of a GOTO message to be one level above the message's target node, U. At the time we flush to the LCA and create the GOTO message, we know the height of U; as long as the GOTO message is not flushed to the same level as U (or deeper), the maximum query path will not be lengthened.

So, for example, if the root node in Figure 3 is at height 7 and the LCA of s is at height 3, then the GOTO message will get lazily flushed down the tree until it resides in the buffer of some node at height 4. At that point the GOTO will be converted to a regular bracketing pair of node pivots and a child pointer, as shown in Figure 4.

In flushing a GOTO above the target height, the only additional work is possibly deleting obviated internal nodes. In the simple case, where a GOTO covers the same key range as one child, flushing simply moves the message down the DAG one level, possibly lifting some of the destination key. One may also delete messages obviated by the GOTO as part of flushing. The more difficult case is when a GOTO message covers more than one child pointer in a node. In this case, we retain only the leftmost and rightmost nodes. We flush the GOTO to the leftmost child and adjust the pivot keys to include both the left "fringe" and the GOTO message's key range. We similarly adjust the rightmost pivot's keys to exclude any keys covered by the GOTO message (logically deleting these keys, but deferring clean-up). Any additional child pointers and pivots between the left and rightmost children covered by the GOTO are removed and the reference counts on those nodes are reduced by one, effectively deleting those paths.

Converting a GOTO message into node pivots and a child pointer is conceptually similar to flushing a GOTO. As with flushing, a GOTO message takes precedence over any older messages or pre-existing node pivots and child pointers that it overlaps. This means that any messages for a child that are obviated by the GOTO may be dropped before the GOTO is applied.

The simplest case is where a single child is exactly covered

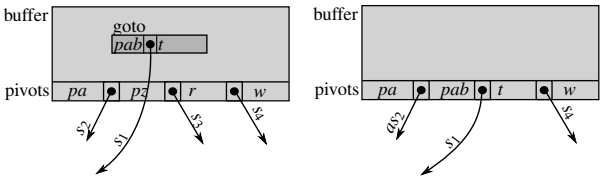


Figure 4: Converting a GOTO message (left) into a pair of bracketing pivots and a child pointer (right). Note that the GOTO message's pivots *pab* and *t* completely cover the range specified by the pre-existing node pivots *pz* and *r*, so the GOTO's pivots replace those pivots in the new node (right). Additionally, the translation prefix s_2 is changed to as_2. This is because, in the original node (left), the prefix *p* is lifted by pivots *pa* and *pz*, but in the new node (right), new prefix *pa* is lifted by pivots *pa* and *pab*; *a* must therefore be prepended to the translation prefix in order to maintain traversal equivalence. (Not shown: the reference counts of covered children are dropped.)

by the GOTO; here, we just replace the pointer and decrement the original child's reference count. For example, in Figure 4, the GOTO message's range (pab, t) completely covers the old pivot range (pz, r). Thus, when converting the GOTO message into regular pivots, we drop the node pointer with translation prefix s_3, and we decrement the reference count of the node to which it pointed.

Partial overlap with a pivot range is handled by a combination of adjusting pivots and adding new pointers. In Figure 4, the GOTO message partially overlaps the old pivot ranges (pa, pz) and (r, w), and there is live data on the "left" fringe of this child (keys between *pa* and *pab* are not covered by this GOTO). We modify the original pivot keys so that subsequent traversals through their child pointers only consider live data, but we leave the child nodes untouched and defer physically deleting their data and relifting their keys. Note that in this example, the subtree between updated pivots *pa* and *pb* should lift *pa* instead of just *p*, so we add *a* to the translation prefix until the next time this child is actually flushed and re-lifted. We finally replace the covered pivots with new pivot keys and a child pointer for the GOTO's target (the pointer between *pab* and *t* in the right portion Figure 4). In the case where a GOTO message overlaps a single child with live data on the left and right fringe (not illustrated), we would create a third pointer back to the original child and increment its reference count accordingly, with appropriate translation prefixes and pivots to only access the live data on the "right" side.

Finally, as with flushing a GOTO, if a GOTO covers multiple children, we remove all of the references to the "interior" children, and replace them with a single child pointer to the GOTO target. We note that this can temporarily violate our target fanout; we allow the splitting and merging process, described next, to restore the target fanout in the background.

3.3 Flushes, splits, and merges

We now explain how node flushes, splits, and merges interact with reference counting, node sharing, translation prefixes, and GOTO messages.

At a high level, we break flushing, splitting, and merging into two steps: (1) convert all involved children into ***simple*** children (defined below), then (2) apply the standard lifted B^ε-tree flushing, splitting, or merging algorithm.

A child is ***simple*** if it has reference count 1 and the edge pointing to the child has no translation prefix. When a child is simple, the B^ε-DAG locally looks like a lifted B^ε-tree, so we can use the lifted B^ε-tree flushing, splitting, and merging algorithms, since they all make only local modifications to the tree.

The B^ε-DAG has an invariant that one may only flush into a simple child. Thus, one of two conditions that will cause a node to be made simple is the accumulation of enough messages in the parent of a node—i.e., a copy-on-abundant-write. The second condition that can cause a node to become simple is the need to split or merge the node by the background, healing thread; this can be triggered by healing a node that has temporarily violated the target fanout, or any other condition in the baseline B^ε-tree that would cause a split or merge.

We present the process for converting a child into a simple child as a separate step for clarity only. In our implementation, the work of making a child simple is integrated with the flushing, splitting and merging algorithms. Furthermore, all the transformations described are performed on in-memory copies of the node, and the nodes are written out to disk only once the process is completed. Thus simplifying children does not change the IO costs of flushes, splits, or merges.

The first step in simplifying a child is to make a private copy of the child, as shown in Figure 5. When we make a private copy of the child, we have to increment the reference counts of all of the child's children.

Once we have a private copy of the child, we can discard any data in the child that is not live, as shown in the first two diagrams of Figure 6. For example, if the edge to the child has translation prefix s_1, then all queries that reach the child will have s_1 as a prefix, so we can discard any messages in the child that don't have this prefix, because no query can ever see them. Similarly, we can drop any children of the child that are outside of the range of s_1 keys, and we can update pivots to be entirely in the range of s_1 keys. When we adjust pivots in the child, we may have to adjust some of the child's outgoing translation prefixes, similar to when we converted GOTO messages to regular pivots.

Finally, we can relift the child to "cancel out" the translation prefix on the edge pointing to the child and all the s_1 prefixes inside the child. Concretely, we can delete the s_1 translation prefix on the child's incoming edge and delete the s_1 prefix on all keys in the child.

A consequence of this restriction is that translation prefixes

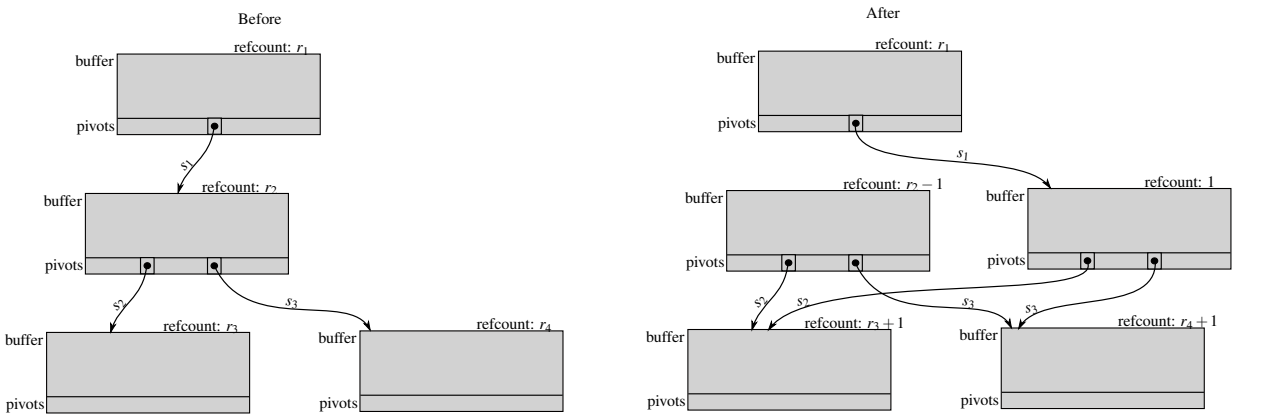


Figure 5: Creating a private copy of a shared child. The original node's contents are copied, and its reference count is decremented. Since the private copy points to all of the original node's children, those children have their reference count increased by one. (Pivot keys are omitted for clarity; they remain unchanged.)

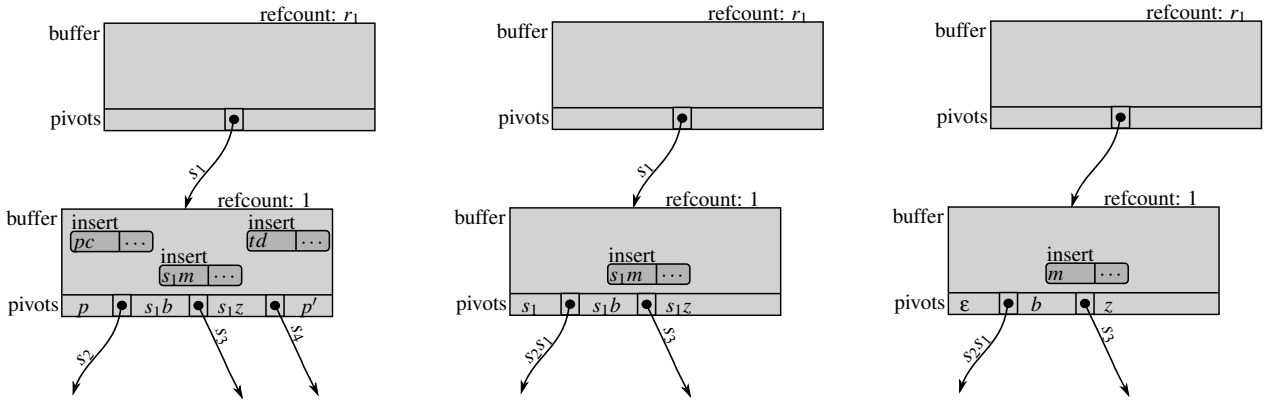


Figure 6: Eliminating a child's translation prefix. The original child node (left) is a private copy with reference count one. First, nodes with unreachable keys are deleted and reclaimed (center). Then the translation prefix s_1 is removed from the incident edge and logically applied to all pivot keys and all keys in buffered messages (right).

should always be NULL after a flush. Intuitively, one only needs a translation prefix to compensate for the effect on lifting of logically deleted data still in a node; after a flush, this data is physically deleted and the node is re-lifted, obviating the need for a translation prefix.

As described, these steps slightly modify the amortized and background work to "heal" irregularities in the B$^\varepsilon$-DAG. This work is primarily driven by subsequent writes to the affected range; a shared node that is not modified on any path can remain shared indefinitely. In our current prototype, we do track shared nodes with very little live data, and mark them for flushing either in the background or under space pressure to reclaim space. The key feature of this design is the flexibility to rewrite nodes only when it is to the advantage of the system—either to reclaim space or recover locality for future queries.

3.4 Putting it all together

The remaining lifted B$^\varepsilon$-tree operations are unchanged in a B$^\varepsilon$-DAG. Inserts, deletes, and clones just add messages to the root node's buffer. When an internal node's buffer becomes full, we flush to one of its children (after making the child simple, if necessary). When a leaf becomes too large or too small, we split or merge it (after making the leaf simple). When an internal node has too many or too few children, we split or merge it (again after making it simple).

3.5 Asymptotic Analysis

This subsection shows that adding cloning does not affect the asymptotics of other operations, and that the cost of a clone is logarithmic in the size of the tree.

Insert, query, and clone complexity all depend on the B$^\varepsilon$-DAG height, which is bounded by the height of a lifted B$^\varepsilon$-tree with the same logical state. To see why, consider the following straightforward transformation of a B$^\varepsilon$-DAG to a B$^\varepsilon$-tree: first

flush all GOTO messages until they become regular pivots, then break the CoW sharing of all nodes. Since this conversion can only increase the height of the data structure, a logically equivalent lifted B^ε-tree is at least as tall as a B^ε-DAG.

The height of a B^ε-tree is $O(\log_B N)$, where N is the total number of items that have been inserted into the tree. Hence the height of a B^ε-DAG is $O(\log_B N)$, where N is the number of keys that have been created, either through insertion or cloning, in the B^ε-DAG.

Queries. Since the height of the B^ε-DAG is $O(\log_B N)$, the IO cost of a query is always $O(\log_B N)$.

Insertions. The B^ε-DAG insertion IO cost is the same as in a B^ε-tree, i.e., $O(\frac{\log_B N}{B^{1-\varepsilon}})$. This is because the IO cost of an insertion is $h \times c/b$, where h is the height of the B^ε-DAG, c is the IO cost of performing a flush, and b is the minimum number of items moved to child during a flush. Flushes cost $O(1)$ IOs in a B^ε-DAG, just as in a B^ε-tree, and flushes move at least $\Omega(B/B^{1-\varepsilon})$ items, since the buffer in each node has size $\Omega(B)$, and the fanout of each node is $O(B^\varepsilon)$.

Clones. The cost to create a clone can be broken into the online cost, i.e,. the costs of all tasks that must be completed before the clone is logically in place, and the offline costs, i.e., the additional work that is performed in the background as the GOTO message is flushed down the tree and eventually converted to regular pivots.

The online cost of cloning s to d is merely the cost to push all s messages to s's LCA and insert the GOTO message. The cost of pushing all the messages to the LCA is $O(\log_B N)$ IOs. Inserting the new GOTO message costs less than 1 IO, so the total cost of creating a clone is $O(\log_B N)$ IOs.

The background cost is incurred by the background thread that converts all edges with a translation prefix into simple edges. We bound the IO cost of this work as follows. A clone from s to d can result in edges with translation prefixes only along four root-to-leaf paths in the B^ε-DAG: the left and right fringes of the sub-dag of all s keys, and the left and right fringes of the sub-dag of all d keys. Thus the total IO cost of the background work is $O(\log_B N)$.

4 Implementation and Optimizations

In this section, we describe two optimizations that reduce the total cost of clones. Although they do not alter the asymptotics, we leverage the file system namespace and BetrFS design to save both background and foreground IO.

Preferential splitting. Most background cloning work involves removing unrelated keys and unlifted prefix data from ***fringe*** nodes, i.e., nodes that contain both cloned and non-cloned data. Thus, we could save work by reducing the number of fringe nodes.

Baseline BetrFS picks the middle key when splits a leaf node. With ***preferential splitting***, we select the key that maximizes the common prefix of the leaf, subject to the constraint that both new leaves should be at least 1/4 full. Since data in the same file share the same prefix (as do files in the same directory), preferential splitting reduces the likelihood of having fringe nodes in a clone.

A naïve approach would compare the central half of all leaf keys and pick the two adjacent keys with the shortest common prefix. However, this scan can be costly. We can implement preferential splitting and only read two keys: because the shortest common prefix among adjacent keys is the same as the common prefix of the smallest and the largest candidate keys (the keys at 1/4 and 3/4 of the leaf), we can construct a good parent pivot from these two keys.

Node reclamation. We run a thread in the background that reclaims any node whose reference count reaches 0. As part of the node reclamation process, we decrement each child node's reference count, including nodes pointed to by GOTO messages. Node reclamation proceeds recursively on children whose reference counts reach zero, as well.

This thread also checks any node with a translation prefix. In an extreme case, a node with no reachable data may have a positive reference count due to translation prefixes. For example, if the only incident edge to a sub-DAG has translation prefix s, but no key in the entire sub-DAG has s as a prefix, then all data in the sub-DAG is reclaimable. As part of space reclamation, BetrFS finds and reclaims nodes with no live data, or possibly unshares and merges nodes with relatively little live data.

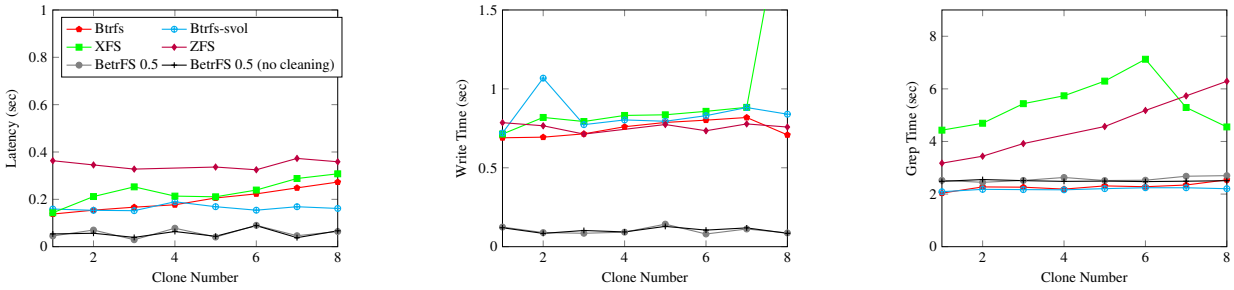
Concurrency. B-tree concurrency is a classic problem, since queries and inserts proceed down the tree, but splits and merges proceed up the tree, making hand-over-hand locking tricky. B^ε-trees have similar issues, since they also perform node splits and merges, and many of the B-tree-based solutions, such as preemptive splitting and merging [28] or sibling links [20], apply to B^ε-trees, as well.

We note here that our cloning mechanism is entirely top-down. Messages get pushed down to the LCA, GOTO messages get flushed down the tree, and non-simple edges get converted to simple edges in a top-to-bottom manner. Thus cloning imposes no new concurrency issues within the tree.

Background cleaning. BetrFS includes a background process that flushes messages for frequently queried items down the tree. The intention of this optimization is to improve range and point query performance on frequently queried data: once messages are applied to key-value pairs in B^ε-tree leaves, future queries need not reprocess those messages.

We found that, in the presence of clones, this background task increased BetrFS 0.5's space consumption because, by flushing small changes, the cleaner would break B^ε-DAG nodes' copy-on-write sharing.

Thus we modified the cleaner to never flush messages into any node with a reference count greater than 1; such messages instead wait to be flushed in normal write-optimized batches once enough work has accrued to warrant rewriting the node.



(a) Time to clone a directory. (b) Small write latency. (c) Grep Time.

Figure 7: Latency to clone, write, and read as a function of the number of times a directory tree has been cloned. Lower is better for all measures.

5 Evaluation

This section evaluates BetrFS 0.5 performance. The evaluation centers around the following questions:

- Do BetrFS 0.5 clones meet the performance goals of simultaneously achieving (1) low latency clone creation, (2) reads with good spatial locality, even after modifications, (3) fast writes, and (4) space efficiency? (§5.1)
- Does the introduction of cloning harm the performance of unrelated operations? (§5.2)
- How can cloning improve the performance of a real-world application? (§5.3)

All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4GiB RAM, and a 500GB, 7200 RPM SATA disk, with a 4096-byte block size. We boot from a USB stick with the root file system, isolating the file system under test to only the workload. The system runs 64-bit Ubuntu 14.04.5.

We compare BetrFS 0.5 to baseline BetrFS, ext4, Btrfs, XFS, ZFS, and NILFS2. We used BetrFS version 0.4 from `github.com/oscarlab/betrfs`, ZFS 0.6.5.11 from `zfsonlinux.org` and kernel default versions of the other file systems. Unless noted, each experiment was run a minimum of 5 times. We present the mean and display bars that indicate the minimum and maximum times over all runs. Similarly, $\pm$ terms bound the minimum and maximum values over all runs. Unless noted, all benchmarks are cold-cache tests.

BetrFS only works on a modified 3.11.10 kernel, so we run BetrFS on that kernel; all other file systems run on 4.9.142. We note that we ran experiments on both kernel versions, and performance was generally better on the newer kernel; we present the numbers for the newer kernel.

5.1 Cloning Performance

To evaluate the performance of cloning (and similar copy-on-write optimizations in other file systems), we wrote a microbenchmark that begins by creating a directory hierarchy with eight directories, each containing eight 4MiB-files. The microbenchmark then proceeds in rounds. In each round, we create a new clone of the original directory hierarchy and measure the clone operation's latency (Figure 7a). We next write 16 bytes to a 4KiB-aligned offset in each newly cloned file—followed by a sync—in order to measure the impact of copy-on-write (Figure 7b) on writes. We then clear the file system caches and grep the newly copied directory to measure cloning's impact on read time (Figure 7c). Finally, we record the change in space consumption for the whole file system at each step (Table 1). We call this workload *Dookubench*.

We compare directory-level clone in BetrFS 0.5 to 3 Linux file systems that either support volume snapshots (Btrfs and ZFS) or reflink copies of files (Btrfs and XFS). We compare in both modes; the label Btrfs-svol is in volume-snapshot mode. For the file systems that support only file-level clones (XFS and Btrfs without svol), the benchmark makes a copy of the directory structure and clones the files.

For BetrFS 0.5, we present data in two modes. In "no cleaner" mode, we disable the background process in BetrFS 0.5 that flushes data down the tree (Section 4). We found that this background work created a lot of noise in our space experiments, so we disabled it to get more precise measurements. We also run the benchmark in BetrFS 0.5's default mode (with the cleaner enabled). As reported below, the cleaner made essentially no difference on any benchmark, except to increase the noise in the space measurements.

Figure 7a shows that BetrFS 0.5's cloning time is around 60ms, which is 33% faster than the closest data point from another file system (the first clone on XFS), 58% faster than a volume clone on Btrfs, and an order of magnitude faster than the worst case for the competition. Furthermore, BetrFS 0.5's clone performance is essentially flat throughout the experiment. Thus we have achieved our objective of cheap clones. Btrfs and ZFS also have flat volume-cloning performance, but worse than in BetrFS 0.5. Both Btrfs and XFS file-level clone latencies, on the other hand, degrade as a function of the number of prior clones; after 8 iterations, clone latency is roughly doubled.

In terms of write costs, the cost to write to a cloned file or

FS	Δ KiB/round	σ
Btrfs	176 ± 112	56.7
Btrfs-svol	32 ± 0	0
XFS	32.6 ± 95.4	50.9
ZFS	250 ± 750	462.9
BetrFS 0.5 (no cleaning)	31.3 ± 29.8	19.9
BetrFS 0.5	16.3 ± 950.8	460.8

Table 1: Average change in space usage after each Dookubench round (a directory clone followed by small, 4KiB-aligned modifications to each newly cloned file).

volume is flat for all file systems, although BetrFS 0.5 can ingest writes 8–10× faster. Thus we have not sacrificed the excellent small-write performance of BetrFS.

Figure 7c shows that scans in BetrFS 0.5 are competitive with the best grep times from other file systems in our benchmarks. Furthermore, grep times in BetrFS 0.5 do not degrade during the experiment. XFS and ZFS degrade severely—after six clones, the grep time is nearly doubled. For XFS, there appears to be some work that temporarily improves locality, but the degradation trend resumes after more iterations. Btrfs degrades by about 20% for file-level clones and 10% for volume level clones after eight clones. This trend continues: after 17 iterations (not presented for brevity), Btrfs read performance degrades by 50% with no indication of leveling off.

Table 1 shows the change in file system space usage after each microbenchmark round. BetrFS 0.5 uses an average of 16KiB per round, which is half the space of the next best file system, Btrfs in volume mode. BetrFS 0.5's space usage is very noisy due to its cleaner—unsurprisingly, space usage is *less* after some microbenchmark rounds complete, decreasing by up to 693KiB. When the cleaner is completely disabled, space usage is very consistent around 32KiB. Thus enabling the cleaner reduces average space consumption but introduces substantial variation. Overall, these results show that BetrFS 0.5 supports space-efficient clones.

In total, these results indicate that BetrFS 0.5 supports a seemingly paradoxical combination of performance features: clones are fast and space-efficient, and random writes are fast, yet preserve good locality for sequential reads. No other file system in our benchmarks demonstrated this combination of performance strengths, and some also showed significant performance declines with each additional clone.

5.2 General Filesystem Performance

This section evaluates whether adding cloning erodes the performance advantages of write-optimization in BetrFS. Our overarching goal is to build a file system that performs well on *all* operations, not just clones; thus, we measure a wide range of of microbenchmarks and application benchmarks.

Sequential IO. We measure the time to sequentially write a 10GiB file to disk (the benchmarking machine has only 4GiB of RAM, so this is more than double the available RAM), and then sequentially re-read the data from disk. Figure 8 shows the throughput of both operations. All the filesystems perform sequential IO relatively well. BetrFS 0.5 performs sequential reads at comparable throughput to BetrFS, ZFS, and NILFS2, which is only about 19% less than ext4, Btrfs and XFS. Sequential writes in BetrFS 0.5 are within 6% to the fastest file system (Btrfs). We attribute this improvement to preferential splitting, which creates a pivot matching the maximum file data key at the beginning of the workload, avoiding expensive leaf relifting in subsequent node splits.

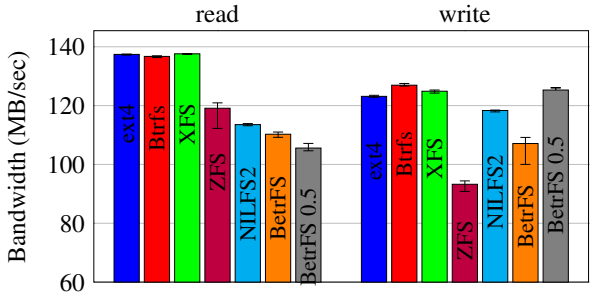


Figure 8: Bandwidth to sequentially read and write a 10 GiB file (higher is better).

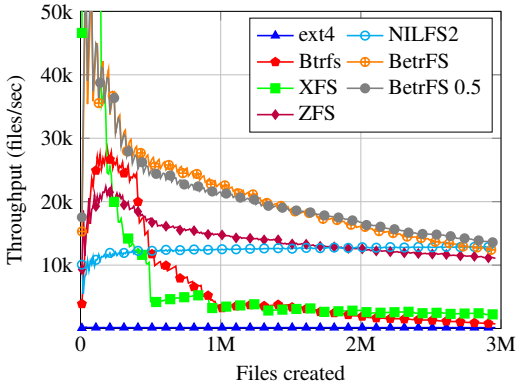


Figure 9: Cumulative file creation throughput during the Tokubench benchmark (higher is better).

Random IO. We measure random write performance with a microbenchmark that issues 256K 4-byte overwrites at random offsets within a 10GiB file, followed by an fsync. This number of overwrites was chosen to run for at least several seconds on the fastest filesystem. Similarly, we measure random read performance by issuing 256K 4-byte reads at random offsets within an existing 10GiB file.

Table 2 shows the execution time of the random write and random read microbenchmarks. BetrFS 0.5 performs these random writes 39–67× faster than conventional filesystems and 8.5% slower than BetrFS. BetrFS 0.5 performs random reads 12% slower than the fastest file system.

Tokubench. We evaluate file creation using the Tokubench benchmark [13]. Tokubench creates three million 200-byte files in a balanced directory tree (no directory is allowed to

FS	random write (s)	random read (s)
ext4	2770.6 ± 21.3	1947.9 ± 5.9
Btrfs	2069.1 ± 14.6	1907.5 ± 6.4
XFS	2863.4 ± 14.1	2023.3 ± 27.8
ZFS	3410.6 ± 937.4	2163.9 ± 112.2
NILFS2	2022.0 ± 4.8	1931.1 ± 26.6
BetrFS	4.7 ± 0.2	2201.1 ± 2.9
BetrFS 0.5	5.5 ± 0.1	2129.8 ± 6.8

Table 2: Time to perform 256K 4-byte random writes/reads (1 MiB total IO, lower is better).

Back-end	FS	lxc-clone (s)
Dir	ext4	19.514 ± 1.214
	Btrfs	14.822 ± 0.076
	ZFS	16.194 ± 0.538
	XFS	55.104 ± 1.033
	NILFS2	26.622 ± 0.396
	BetrFS 0.5	8.818 ± 1.073
ZFS	ZFS	0.478 ± 0.019
Btrfs	Btrfs	0.396 ± 0.036
BetrFS 0.5	BetrFS 0.5-clone	0.118 ± 0.010

Table 4: Latency of cloning a container.

have more than 128 children). BetrFS 0.5 matches BetrFS throughput, which is strictly higher than any other file system, (except for one point at the end where NILFS2 is 8.7% higher), and as much as 95× higher throughput than ext4.

Directory Operations. Table 3 lists the execution time of three common directory operations—grep, find or delete—on the Linux 3.11.10 kernel source tree.

BetrFS 0.5 is comparable to the baseline BetrFS on all of these operations, with some marginal (4–5%) overhead on grep and delete from adding cloning. We also note that we observed a degradation for BetrFS on larger directory deletions; the degradation is unrelated to cloning and we leave investigation of this for future work. Overall, BetrFS 0.5 maintains the order-of-magnitude improvement over the other file systems on find and grep.

Application Benchmarks. Figure 10 reports performance of the following application benchmarks. We measure two BetrFS 0.5 variants: one with no clones in the file system (labeled BetrFS 0.5), and one executing in a cloned Linux-3.11.10 source directory (labeled BetrFS 0.5-clone).

The `git clone` workload reports the time to clone a local Linux source code repository, which is cloned from `github.com/torvalds/linux`, and `git diff` reports the time to diff between the v4.14 and v4.7 tags. The tar workload measures the time to `tar` or un-`tar` the Linux-3.11.10 source. The rsync workload copies the Linux-3.11.10 source tree from a source to a destination directory within the same partition and file system. With the `--in-place` option, `rsync` writes data directly to the destination file rather than creating a temporary file and updating via atomic rename. The IMAP

FS	find (s)	grep (s)	delete (s)
ext4	2.22 ± 0.0	37.71 ± 7.1	3.38 ± 2.2
Btrfs	1.03 ± 0.0	8.88 ± 0.3	2.88 ± 0.0
XFS	6.81 ± 0.2	57.79 ± 10.4	10.33 ± 1.4
ZFS	10.50 ± 0.2	38.64 ± 0.4	9.18 ± 0.1
NILFS2	6.72 ± 0.1	8.75 ± 0.2	9.41 ± 0.4
BetrFS	0.23 ± 0.0	3.71 ± 0.1	3.22 ± 0.4
BetrFS 0.5	0.21 ± 0.0	3.87 ± 0.0	3.37 ± 0.1

Table 3: Time to perform recursive grep, find and delete of the Linux 3.11.10 source tree (lower is better)

server workload initializes a Dovecot 2.2.13 mailserver with 10 folders, each containing 2500 messages, then measures throughput of 4 threads, each performing 1000 operations with 50% reads and 50% updates (marks, moves, or deletes).

In most of these application benchmarks, BetrFS 0.5 is the highest performing file system, and generally matches the other file systems in the worst cases. In a few cases, where the application is write-intensive, such as git clone and rsync, BetrFS 0.5-clone degrades relative to BetrFS 0.5, attributable to the extra work of unsharing nodes, but the performance is still competitive with, or better than, the baseline file systems. These application benchmarks demonstrate that extending write-optimization to include clones does not harm—and can improve—application-level performance.

5.3 Cloning Containers

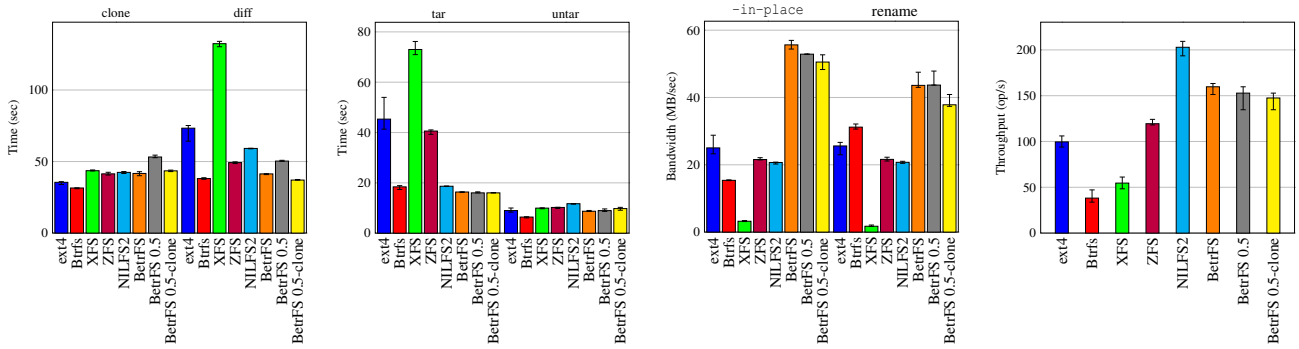
Linux Containers (LXC) is one of several popular container infrastructures that has adopted a number of storage back-ends in order to optimize container creation. The default backend (dir) does a `rsync` of the component directories into a single, chroot-style working directory. The ZFS and Btrfs back-ends use subvolumes and clones to optimize this process. We wrote a BetrFS 0.5 backend using directory cloning.

Table 4 shows the latency of cloning a default Ubuntu 14.04 container using each backend. Interestingly, BetrFS 0.5 using clones is 3–4× faster than the other cloning backends, and up to two orders of magnitude faster than the others.

6 Related work

File systems with snapshots. Many file systems implement a *snapshot* mechanism to make logical copies at whole-file-system-granularity [27]. Tree-based file systems, like WAFL [15], ZFS [41], and Btrfs [29], implement fast snapshots by copying the root. WAFL FlexVols [12] add a level of indirection between the file system and disks, supporting writable snapshots and multiple active file system instances.

FFS [21] implements read-only file system *views* by creating snapshot inode with a pointer to each disk block; the first time a block is modified, FFS copies the block to a new address and updates the block pointer in the snapshot inode.



(a) Git latency. Lower is better.

(b) Tar latency. Lower is better.

(c) Rsync throughput. Higher is better.

(d) IMAP server throughput. Higher is better.

Figure 10: Application benchmarks.

NILFS [19] is a log-structured file system that writes B-tree checkpoints as part of each logical segment. NILFS can create a snapshot by making a checkpoint block permanent.

GCTree [11] implements snapshots on top of ext4 by creating chains of metadata block versions. Each pointer in the metadata block has a "borrowed bit" to indicate whether the target block was inherited from the previous version. Ext3cow [26] snapshots are defined by an epoch. Ext3cow can render any epoch's file-system view by fetching entries alive at that epoch. NOVA-Fortis [38] supports snapshots by adding private logs to each inode.

File or directory clones. AFS [16] introduced the idea of volumes as a granularity for cloning and backup in a large, distributed file system; volumes isolate performance disruption from cloning one user's data from other users. Episode [7] can create immutable *fileset* clones by copying all the fileset's anodes (inodes) and marking all block pointers copy-on-write. Btrfs [29] can create *file* clones by sharing a file's extents. Windows® 2000 Single Instance Storage (SIS) [5] uses deduplication techniques to implement a new type of link that has copy semantics. Creating the first SIS link requires a complete data copy to a shared store. Writes are implemented copy-on-close: once all open references to an SIS link are closed, sharing is broken at whole-file granularity. Copy-on-close optimizes for the case of complete overwrites.

Versioning file systems. Versioning files is an old idea, dating back to at least TENEX system [4]. Versioning file systems have appeared in a number of OSes [1, 22, 31], but often with limitations such as a fixed number of versions per file and no directory versioning. The Elephant File System [30] automatically versions all files and directories, creating/finalizing a new file version when the file is opened/closed. Each file has an inode log that tracks all versions. CVFS [32] suggests journal-based metadata and multi-version B-trees as two ways to save space in versioning file systems. Versionfs [23] is a stackable versioning file system where all file versions are maintained as different files in the underlying file system.

Exo-clones [33] were recently proposed as a file format for efficiently serializing, deserializing, and transporting volume clones over a network. Exo-clones build upon an underlying file system's mechanism for implementing snapshots or versions. Nimble clones in BetrFS 0.5 have the potential to make exo-clones faster and smaller than on a traditional copy-on-write snapshotting system.

Database indexes for dynamic hierarchical data. The closest work to ours in databases is the BO-tree [14], a B-tree indexing scheme for hierarchical keys that supports moving key subtrees from one place to another in the hierarchy. They even support moving internal nodes of the key hierarchy, which we do not. However, they do not support clones—only moves—and their indexes are not write optimized.

7 Conclusion

This paper demonstrates how to use write-optimization to decouple writes from copies, rendering a cloning implementation with the *nimble* performance properties: efficient clones, efficient reads, efficient writes, and space efficiency. This technique does not harm performance of unrelated operations, and can unlock improvements for real applications. For instance, we demonstrate from 3–4× improvement in LXC container cloning time compared to optimized back-ends. The technique of applying batched updates to the data structure itself likely generalize. Moreover, our cloning implementation in the B^ε-DAG could be applied to any application built on a key-value store, not just a file system.

Acknowledgments

We thank the anonymous reviewers and our shepherd Changwoo Min for their insightful comments on earlier drafts of the work. This research was supported in part by NSF grants CCF-1715777, CCF-1724745, CCF-1725543, CSR-1763680, CCF-1716252, CCF-1617618, CCF-1712716, CNS-1938709, and CNS-1938180. The work was also supported by VMware, by EMC, and by NetApp Faculty Fellowships.

References

[1] Vax/VMS System Software Handbook, 1985.

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

[3] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to B^ε-trees and write-optimization. *:login; Magazine*, 40(5):22–28, Oct 2015.

[4] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson. Tenex, a paged time sharing system for the pdp - 10. *Commun. ACM*, 15(3):135–143, March 1972.

[5] Bill Bolosky, Scott Corbin, David Goebel, and John (JD) Douceur. Single instance storage in windows 2000. In *Proceedings of 4th USENIX Windows Systems Symposium*. USENIX, January 2000.

[6] Gerth Stolting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 546–554, 2003.

[7] Sailesh Chutani, Owen T Anderson, Michael L Kazar, Bruce W Leverett, W Anthony Mason, Robert N Sidebotham, et al. The episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, 1992.

[8] Alex Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Martin Farach-Colton. File systems fated for senescence? nonsense, says science! In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, pages 45–58, 2017.

[9] Alex Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Martin Farach-Colton. How to fragment your file system. *:login; Magazine*, 42(2):22–28, Summer 2017.

[10] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald Porter, and Martin Farach-Colton. Filesystem aging: It's more usage than fullness. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[11] Chris Dragga and Douglas J. Santry. Gctrees: Garbage collecting snapshots. *ACM Transactions on Storage*, 12(1):4:1–4:32, 2016.

[12] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. Flexvol: Flexible, efficient file volume virtualization in wafl. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 129–142, 2008.

[13] John Esmet, Michael A Bender, Martin Farach-Colton, and Bradley C Kuszmaul. The tokufs streaming file system. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, 2012.

[14] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Faerber. Indexing highly dynamic hierarchical data. In *VLDB*, 2015.

[15] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 19–19, 1994.

[16] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[17] 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. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 301–315, 2015.

[18] 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. BetrFS: Write-optimization in a kernel file system. *ACM Transactions on Storage*, 11(4):18:1–18:29, 2015.

[19] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[20] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems*, 6(4), December 1981.

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

[22] Lisa Moses. TOPS-20 User's manual.

[23] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 115–128, 2004.

[24] Prashanth Nayak and Robert Ricci. Detailed study on linux logical volume manager. *Flux Research Group University of Utah*, 2013.

[25] Patrick O'Neil, Edward Cheng, Dieter Gawlic, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[26] Zachary Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[27] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. In *In Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990.

[28] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage*, 3(4):2:1–2:27, 2008.

[29] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage*, 9(3):9:1–9:32, 2013.

[30] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 110–123, 1999.

[31] Mike Schroeder, David K. Gifford, and Roger M. Needham. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Opeating Systems Principles*. Association for Computing Machinery, Inc., November 1985.

[32] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, 2003.

[33] Richard P. Spillane, Wenguang Wang, Luke Lu, Maxime Austruy, Rawlinson Rivera, and Christos Karamanolis. Exo-clones: Better container runtime image management across the clouds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.

[34] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Wenji Li, Raju Rangaswami, and Ming Zhao. Evaluating docker storage performance: from workloads to graph drivers. *Cluster Computing*, pages 1–14, 2019.

[35] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. In search of the ideal storage configuration for docker containers. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 199–206. IEEE, 2017.

[36] Veritas. Veritas system recovery. https://www.veritas.com/product/backup-and-recovery/system-recovery, 2019.

[37] Xingbo Wu, Wenguang Wang, and Song Jiang. Totalcow: Unleash the power of copy-on-write for thin-provisioned containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 15. ACM, 2015.

[38] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496, 2017.

[39] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Optimizing every operation in a write-optimized file system. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, pages 1–14, 2016.

[40] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Writes wrought right, and other adventures in file system optimization. *ACM Transactions on Storage*, 13(1):3:1–3:26, 2017.

[41] ZFS. http://zfsonlinux.org/. Accessed: 2018-07-05.

[42] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The full path to full-path indexing. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, pages 123–138, 2018.

[43] Yang Zhan, Yizheng Jiao, Donald E. Porter, Alex Conway, Eric Knorr, Martin Farach-Colton, Michael A. Bender, Jun Yuan, William Jannen, and Rob Johnson. Efficient directory mutations in a full-path-indexed file system. *ACM Transactions on Storage*, 14(3):22:1–22:27, 2018.

[44] Frank Zhao, Kevin Xu, and Randy Shain. Improving copy-on-write performance in container storage drivers. Storage Developer's Conference, 2016.