A Crash-Safe Key-Value Store Using Chained Copy-on-Write B-trees

by

Bruno Castro-Karney

Supervised by Xi Wang

A senior thesis submitted in partial fulfillment of

the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering

University of Washington

June 2018

Presentation of work given on _____

Thesis and presentation approved by _____

Date _____

A Crash-Safe Key-Value Store Using Chained Copy-on-Write B-trees

Bruno Castro-Karney
University of Washington

Abstract

Key-value stores are the fundamental components for storing and retrieving data. It is essential to persist data even in the case of a power failure, which could happen at any time. Crash-safe key-value stores are difficult to implement; they maintain large, complex, on-disk data structures that atomically update. Modern techniques for automatically verifying software correctness are powerful, but require a finite design.

This thesis presents a design of a crash-safe key-value store amenable to automated verification. The key to the design uses chained copy-on-write b-trees to finitize the free-space map. Chaining bounds the size of each b-tree, which limits the number of updates necessary to terminate an operation. Our experience shows that chained copy-on-write b-trees are simpler to understand and provide performance on par with other implementations.

1 Introduction

Key-value stores are a fundamental piece of many systems. They provide the basic service of storing and retrieving data with varying degrees of reliability and performance. Key-value stores are used in databases [9], file systems [11], and operating systems [14]. Similar to file-systems, most persistent key-value stores are designed to survive machine failures [2, 9] without losing data. It is difficult to build services that can gracefully handle failure at any time, while still providing good performance. Many bugs have been found in popular databases and file-systems that have led to data loss [5, 8, 13].

Previous work in formal verification has successfully proved some systems to be free of bugs [7, 12, 14]. However, writing manual proofs requires a significant development effort [6]. Recent research has therefore focused on automated verification, which uses an SMT solver to verify correctness [7, 12]. Automated verification significantly lowers the proof burden for verified systems, but imposes an additional constraint on the design: operations must be finite, meaning every operation must be expressible as a set of traces of bounded length [7, 12].

An on-disk key-value store must maintain a free space map for the allocation and deallocation of disk blocks. This free space map must be consistent with the key-value store, even between crashes. Finitizing the free space map is difficult. One strategy, used by BTRFS [11], is to keep a b-tree to store reference counts for itself and the original key-value store, and modify the b-tree in memory until reaching a fixed point. This is problematic for automated verification because the interface is not finite.

Our contribution is a finite design of a crash-safe key-value store using chained b-trees: each b-tree stores reference counts for the previous tree. We find this approach allows us to finitize the free space map, is easier to understand, and still provides better performance than a simple write-ahead log.

The rest of this paper is organized as follows. §2 introduces copy-on-write b-trees, the challenge of free space, and automated verification. §3 describes free space management with chained copy-on-write b-trees. §4 presents optimizations. §5 compares chained b-trees to alternative implementations. §6 reports implementation details. §7 evaluates performance. §8 concludes.

2 Background

This section gives an overview of copy-on-write b-trees, automated verification, and the challenges of persisting a crash-safe key-value store on disk.

2.1 Copy-on-Write B-trees

B-trees [1, 10] are space efficient structures good for storing large chunks of data. They have a wide branching factor; each node has hundreds of children. Nodes store keys in sorted order, and enforce this invariant on their sub-trees. Traditionally, for a node n with key k at index i, all of n's children up to i store keys less than k, while all n's children after i store keys greater than or equal to k.

This paper exclusively focuses on b+-trees (which we later refer to as just "b-trees"), a variant which keep all values in the leaf nodes. B+-trees have the advantage of more compact interior node structure, which makes them better for dealing with disks.

Many file systems [3, 11] use copy-on-write (COW) b-trees [10] to provide atomicity, crash recovery, and efficient clones. When the tree would modify a node, it copies the node to a new location on disk, then modifies the copy. Then the parent must be updated to point to the new location. Changes propagate up to the root of the tree. When the operation completes successfully, the

old root can be discarded. If the system crashes, copy-on-write preserves the original tree. Creating multiple clones is easy and efficient with COW, since trees share as many unmodified blocks as possible.

B-trees design nodes to fit exactly into a single disk block (usually 4KB). This minimizes the number of disk accesses when reading or writing a tree.

For example, to look up a key, traverse the tree from the root to a leaf node.

1. If the desired key is less than the minimum key in the current node, stop. The tree does not contain the key.

2. If the current node is a leaf node, find the key within the node, read the corresponding block from disk, and return the value.

3. If the current node is a branch node, find the largest key less than or equal to the desired key, read the corresponding block from disk as the new current node, and go to step 1.

Since each node fills one block, the operation reads at most *depth* blocks. In practice, the depth is usually very small, less than 5.

To insert a key value pair into a COW b-tree, traverse the tree to a leaf node, then insert. On the way down, split any already full nodes.

1. Shadow the current node.

   (a) Increment the reference counts of all children of the current node.

   (b) If the reference count for the current node is greater than 1, decrement it.

2. If the key to insert is less than the minimum key in the current node, replace the minimum key.

3. If the current node is a leaf node, write the value to a new block, then insert the key and block into the node.

4. If the current node is a branch node, choose the child the same way as `get`.

5. If the child node is full, split it into two nodes and add a key to the current node, then continue with the child containing the correct key range after the split.

6. Go back to step 1 with the child as the current node.

7. After the insert, write all modified nodes to new locations on disk.

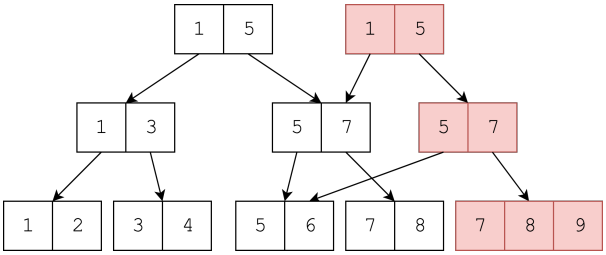


Figure 1: Inserting the key 9 into a COW b-tree. Values in leaves are not shown. Red nodes have been shadowed.

The top-down preemptive split guarantees there is always room in the parent to insert an additional key, in case a child splits. There is a special case when the root node splits. The tree creates a new root with 2 children, and increments the depth [10]. At worst, the tree makes 3 modifications per level - the two split child nodes and the parent.

For example, Figure 1 shows a copy-on-write b-tree after inserting the key 9. First, the tree shadows the root node. Then, because $9 \geq 5$, the tree shadows the right child. Finally, since the current node is a leaf, the tree inserts the key.

`Remove` follows a similar algorithm, except instead of preemptively splitting nodes, the tree preemptively fixes nodes by either borrowing keys from a neighbor or merging two nodes together and removing a key from the parent.

With reference counting, it is simple to deallocate old trees.

1. Decrement the reference count of the root.

2. Then for each child node:

   (a) Decrement the reference count.

   (b) If the reference count is still greater than 0, continue to the next node.

   (c) If the reference count is now 0, free the block and repeat (2) for each of the node's children.

2.2 Free space

In addition to crash recovery, a persistent b-tree needs to manage a free space map of disk. With copy-on-write, and especially with clones, multiple trees may share some blocks. The free space map must then store reference counts of every block on disk. Keeping a finite free-space map consistent with the b-tree in the presence of crashes without leaking resources is a significant challenge. The WAFL file system uses a large bitmap and logical logging [3]. BTRFS uses another copy-on-write b-tree to map block numbers to reference counts [11]. A superblock is atomically updated with the roots of all b-trees.

Problematically, something needs to store reference counts for the free-space tree as well. BTRFS uses the free-space tree to store its own reference counts, and relies on hitting a fixed point. Allocating a block might cause the tree to split, requiring extra allocations from the same tree. Since each block can store many reference counts, eventually the allocations cease. However, a solver would have a difficult time verifying this because of its unbounded nature.

2.3 Automated Verification

Recent work [7, 12] has bypassed manual proofs by formulating verification in satisfiability modulo theories (SMT), and using a solver such as Z3 to check correctness. For a solver to terminate in a reasonable time, the number of execution paths of a program must be limited. Systems like Hyperkernel [7] and Yggdrasil [12] approach this by designing a *finite interface*, one in which all operations avoid unbounded loops and recursion. We take a similar approach to the design of our b-tree.

3 Design

The primary purpose of our design is to provide persistence and crash safety in a form amenable to automated verification. As a result, a major requirement is bounding all operations.

We limit the maximum depth of the tree to 4 (a tree consisting of only the root has a depth of 0), enough to store hundreds of billions of keys. Our nodes fit exactly into 4KB blocks. With 128-bit keys, each node can store up to 170 keys.

3.1 API

The b-tree supports a minimal API:

- `put(key, value)`
- `get(key)`
- `remove(key)`

Keys and values are configurable sizes. For our testing, we used 128-bit integer keys, and 4KB values. The algorithms for `put`, `get`, and `remove` are identical to those in §2.1, except for changes to shadowing and reference counts described in §3.3.

3.2 Free Space

To store reference counts, we unroll the single BTRFS free-space tree into multiple trees, each smaller than the previous. When the first free-space tree needs to allocate a block, it uses a second, much smaller b-tree. That tree uses a third, even smaller b-tree. The final b-tree is guaranteed to be so small that an array of its reference counts can fit into a single 4KB disk block. Chaining four trees together in this way is enough to support a maximum depth of four for the top-level b-tree.

This approach offers two main advantages. First, it is finite. We bound the depth of each tree, which bounds the number of modifications for a single operation, which bounds the number of allocations, which bounds the modifications to the next tree, and so on. Second, we can reuse the same b-tree implementation with different key and value sizes to manage free space. Instead of 128-bit keys and 4KB values, we use 64-bit keys and values. This means we can reuse the verification of the top-level b-tree to show the free-space management is correct.

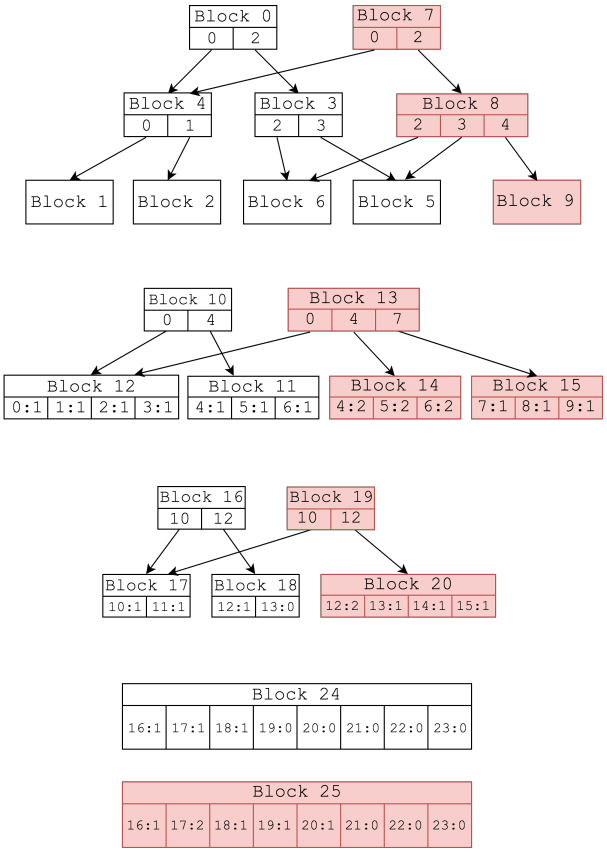


Figure 2: Inserting the key 4 into a COW b-tree with two chained b-tree allocators. Each tree stores reference counts in the tree below it. The smallest tree stores reference counts in a single block array.

3.3 Chaining Implementation

Figure 2 shows a simplified example of inserting a new key into a b-tree with two chained tree allocators. The top-level b-tree has a row of leaf nodes which store keys and block numbers of the corresponding values. The value blocks are shown below the leaf nodes; they are reference counted the same as regular tree nodes. The allocator b-trees simply store reference counts instead of block numbers in the leaf nodes. When the top-level b-tree allocates new blocks for its shadowed nodes, the first allocator tree runs out of space in one of the leaf nodes, so the node splits. The second allocator tree does not require a split. The array allocator shadows the single block and writes the new reference counts for the second

tree allocator. A small number of blocks equal to the maximum number of clones are reserved to store the array allocators.

The allocators dramatically decrease in size. In practice, one leaf node can store up to 255 reference counts.

With chaining, updating reference counts causes updates to another tree. The relevant algorithms are roughly implemented as follows.

```
inc_ref(btree, block):
    ref = get(btree, block)
    put(btree, ref + 1)

dec_ref(btree, block):
    ref = get(btree, block)
    put(btree, block, ref + 1)

alloc_block(btree):
    for block = 0 to MAX_BLOCK
        if !containsKey(btree, block) or
           get(btree, block) == 0:
            inc_ref(btree, block)
            return block


shadow(allocator, node):
    for child_block in node.children:
        inc_ref(allocator, child_block)

    if get(allocator, node.block) > 1:
        dec_ref(allocator, node.block)

    new_block = alloc_block(allocator)
    write(new_block, node)


get(btree, key):
    node = read(btree.root)
    while !node.leaf:
        node = get_child(node, key)
        if node is None:
            return None

    return node_get_value(node, key)


put(btree, key, value):
    for each node on path from root to leaf:
        shadow(btree.allocator, node)
        split_if_full(node)

    node_insert_key(node, key, value)

remove(btree, key):
    for each node on path from root to leaf:
        shadow(btree.allocator, node)
        fix_if_almost_empty(node)

    node_delete_key(node, key)
```

In practice, there is a slight problem with the above algorithms. After shadowing a node, the parent must be updated with the new location. This is solved as a byproduct of caching modified nodes before writing to disk, as described in §3.5.

3.4 Crash Safety

The b-tree must persist through a system failure that could happen at any point. It may not store the current operation, but should retain all previous data. The b-tree reserves the first block of the disk as a superblock. This block contains the block numbers of the root nodes of the top-level b-tree and each allocator tree. Once an operation completes successfully, the superblock is atomically updated with the new roots. If the system crashes before updating the superblock, the trees remain. The next operation after the crash will read the superblock and use the most recent stable trees.

A single atomic put works as follows:

1. Read the superblock to locate each b-tree.
2. `put` the key value pair into the top-level b-tree.
3. Flush the cache of the top-level b-tree (explained further in §3.5). This step simply ensures all shadowed nodes are written to new locations on disk.
4. Deallocate the old b-tree.
5. Repeat steps 3-4 on the first allocator b-tree, then on the second, and so on.
6. Update the superblock.

Deallocation is not strictly necessary; the old b-tree can be kept unmodified as a snapshot and restored later.

3.5 Mark dirty

There is a big problem with the number of allocations needed for chained COW b-trees. If each reference count update requires a `put` which shadows an entire path through the tree, then each `put` requires several allocations. Each of those allocations is another `put` on a smaller tree, which requires even more `puts` the next level down. The trees supposed to be kept small grow even bigger than the large trees, overflowing the array allocator. Most of these allocations are unnecessary.

Mark dirty [10] is an important optimization for COW b-trees, and essential for chaining. When shadowing a

node, set a dirty bit in the node instead of immediately writing it to a new location on disk. Keep the dirty node in memory and use the cached copy for subsequent modifications. When the b-tree is ready to be committed, flush all the dirty nodes to disk.

With *mark dirty*, when a tree allocator first inserts a new reference count into a leaf, it can insert as many more reference counts as that leaf can hold, without requiring additional allocations from the next tree. This is key to bounding the number of updates to complete an operation, as well as minimizing disk accesses.

4 Optimizations

This section aims to improve the performance of chained b-trees by reducing the number of disk accesses. We do this two ways, batching writes and reducing fragmentation.

4.1 Write Batching

Persistent key-value stores commonly use write batching to further reduce disk accesses [9]. We extend *mark dirty* to hold nodes in memory beyond just one operation. After some number of updates, determined by the batch size, the tree flushes its cache. If the tree modifies a node multiple times in a single batch, it only needs to write the node to disk once. The greater the batch size, the longer the tree holds nodes in memory, and the less it writes to disk. However, a larger batch size increases memory usage. We use a batch size of around 100, which reduces the number of disk accesses by two orders of magnitude.

4.2 Fragmentation

One of the biggest causes of disk writes is incrementing childrens' reference counts when marking dirty a clean node. Figure 3a shows what happens in the worst case when shadowing a node with children fragmented across many different blocks. The allocator tree needs to modify every path, which causes unnecessary disk writes. Figure 3b shows what happens in the best case when shadowing a node with children stored in contiguous blocks. The allocator tree only modifies one path.

Designing an allocator to minimize fragmentation is beyond the scope of this paper. A basic approach could examine all dirty nodes at the time of the flush, and then allocate contiguous blocks for siblings. Another strategy could periodically reorganize nodes so siblings blocks lie close together. However, this could become expensive as the tree increases in size.

5 Discussion

This section presents two alternative implementations to chained copy-on-write b-trees, a write-ahead log and a fixed-point b-tree. We compare chaining to both alternatives in terms performance and scalability.

5.1 Write-Ahead Log Allocator

For comparison purposes, we implemented another simpler free space manager, the log allocator. It uses a large bitmap of reference counts, and keeps a write-ahead log for crash recovery. To allocate a block, the log allocator finds an entry in the bitmap with a reference count of zero, appends a new entry to the log, and increments the entry in the bitmap to one. Updating the superblock proceeds as follows.

1. Ensure both the b-tree and log are written to disk, then flush.
2. Update the superblock to point to the new b-tree root and the new root of the log. Since the log may span multiple blocks, the superblock only stores the first block of the log and its length. Then flush the disk.
3. Write the new array of reference counts to disk, then flush.
4. Clear the number of entries in the log, and flush one last time.

The log allocator requires four disk flushes, whereas the chained tree allocators require two. However, for updating a single reference count, the log allocator has less overhead, because it does not need to modify an entire tree. With write batching, both allocators use more memory. In our experiments, the log grew several times faster than the tree allocators' cache, though the cache took more memory initially.

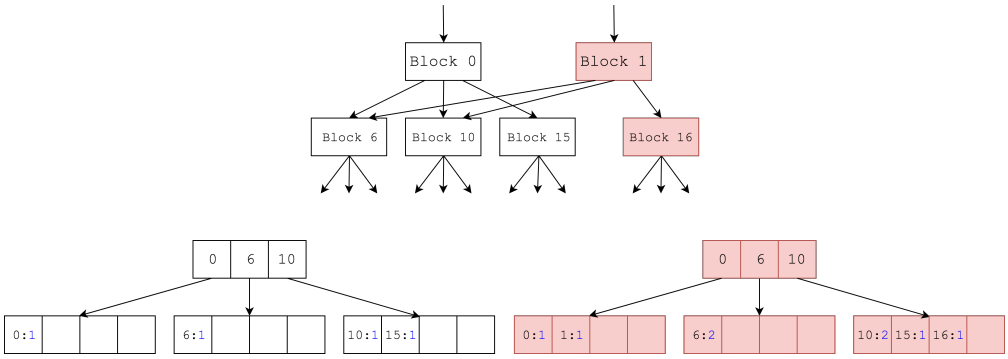
5.2 Fixed Point

A single fixed-point b-tree can allocate blocks for itself using cached dirty nodes, incurring no additional writes in the best case. Chained b-trees thus have a higher overhead compared to a single b-tree. The first time a chained b-tree allocates a block, the next tree must shadow a new path. In the worst case, updating reference counts of a fixed-point tree allocator requires shadowing many paths, same as with a chained tree allocator. However, a chained tree allocator shadows paths in another shallower tree, causing fewer modifications than a deeper fixed-point tree which shadows its own paths. A fixed-point tree requires the same disk flushes as chained trees.

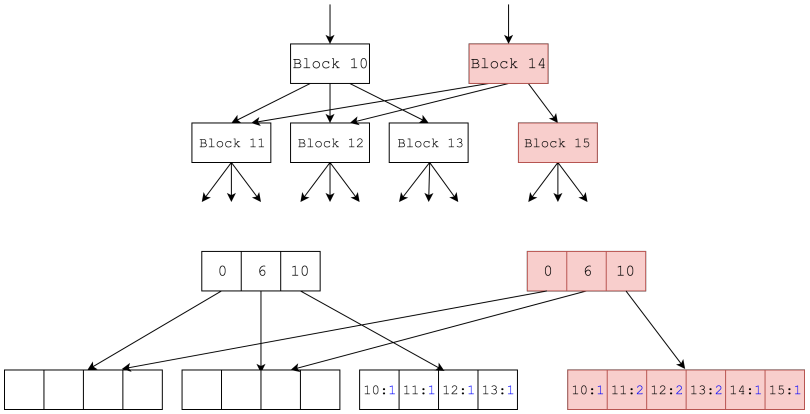
Chained b-trees are slightly less flexible, because the array allocator must fit all reference counts into a single block, and the number of b-trees must be fixed at setup time.

6 Implementation

Our implementation is about 2,000 lines of C code, measured with CLOC. Testing and benchmarking is about 500 lines of C code.



(a) Worst case shadowing of a path. The free-space tree touches every node to update reference counts.



(b) Best case shadowing of a path. The free-space tree only needs to touch a single path to a leaf.

Figure 3: Best case vs. worst case shadowing of a path in a COW b-tree with another COW b-tree allocator.

7 Evaluation

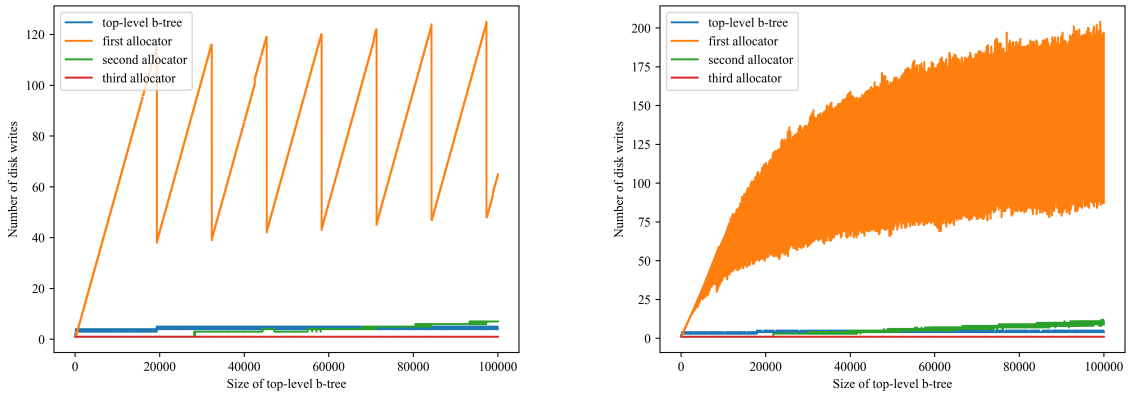
This section aims to answer the following questions:

- What is the performance of the b-tree in terms of disk operations (reads, writes, and flushes)?
- What is the performance impact of the various optimizations?
- How does the performance of chained b-tree allocators compare to a simpler array allocator?

7.1 Performance of Chained B-tree Allocators

Figure 4a shows the number of disk writes each tree makes for a single atomic `put`, as a function of the size of the tree. By far, the first b-tree allocator dominates the number of writes. This is because it makes many more modifications than any other tree. Recall the algorithm for shadowing a node. When the top-level b-tree shadows a node for the first time, it increments the reference count of all its children. With 100,000 keys, the tree has a depth of two. To insert a new key, it must shadow three nodes on the path from the root to the leaf. For each shadowed node, it increments around 128 reference counts. This means the first b-tree allocator needs to update hundreds of keys. The tree then has to shadow several different paths from the root to different leaves. A b-tree needs to write each shadowed node once to disk. While the top-level b-tree only shadows three nodes, the first allocator b-tree shadows many more.

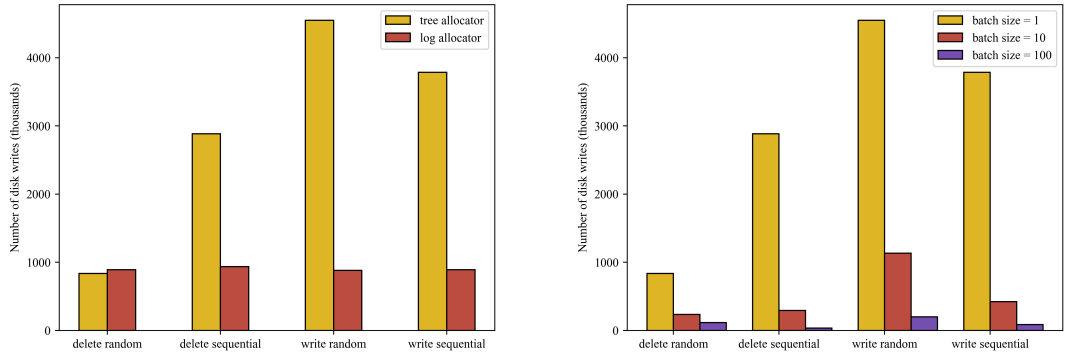
Interesting behavior occurs with sequential puts. Periodically, the number of writes from the first allocator tree drops sharply. This happens when a node high in the top-level b-tree, possibly the root or one of its direct children, splits. Before the split, shadowing the full node increments 170 reference counts. After the split, shadowing the new node only increments half of the reference counts. The splits occur at regular intervals because the puts are sequential. Figure 4b shows writes per put per tree with random keys.



(a) Keys inserted in ascending order.

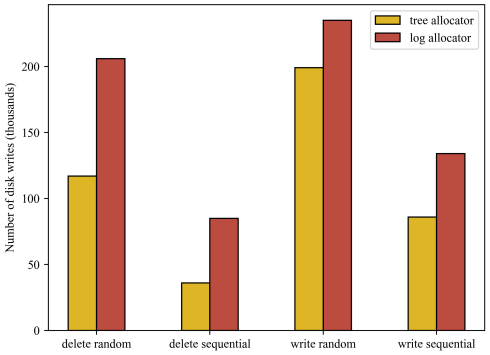
(b) Keys inserted in random order.

Figure 4: Number of disk writes to complete a single atomic put, as a function of the size of the tree.



(a) Chained tree allocators vs. write-ahead log allocator.

(b) Chained tree allocators of different batch sizes.



(c) Chained tree allocators vs. write-ahead log allocator, both using a batch size of 100.

Figure 5: Comparing Disk I/O on `db_bench` after inserting 50,000 128-bit keys with 4KB values (lower is better).

7.2 Micro-benchmarks

db_bench is a set of microbenchmarks provided with LevelDB [4]. We selected four benchmarks to evaluate performance. Each commits 50,000 operations, using 128-bit keys and 4K values.

- *Write Sequential* inserts keys in ascending order.
- *Write Random* inserts random keys.
- *Delete Sequential* takes a full b-tree of 50,000 keys and deletes all keys in ascending order.
- *Delete Random* takes a full b-tree keys and tries to delete randomly selected keys between 0 and 50,000.

Figure 5a compares the number of writes of chained b-tree allocators versus a write-ahead log allocator. This configuration uses a batch size of one; each put is atomic. This means the chained allocators flush their caches after every operation, deallocate all old trees, and then have to shadow everything again. The log allocator has no extra trees to shadow, so it incurs less overhead.

Figure 5b compares the number of writes of chained b-tree allocators using different batch sizes. With a larger batch size, the b-trees cache nodes for longer. The smaller trees can cache every node, allocate blocks using the in-memory tree, and only write to disk after every batch.

Figure 5c compares the number of writes of chained b-tree allocators versus a write-ahead log allocator, using a batch size of 100 for both. While both allocators benefit from batching, the tree allocator scales better than the log allocator. This is because write batching minimizes the overhead of chaining, and tree allocators better handle sparse keys. Since the log allocator uses a bitmap, keys that are far apart in value are stored in separate blocks on disk. Updating those keys requires writing to different blocks. A b-tree can fit those keys in the same node if there are only a few keys in between.

8 Conclusion

Chained copy-on-write b-trees finitize the free-space map of crash-safe key-value stores. They provide crash-safety, and good performance with optimization. The finite design allows for automated verification in the future, and is simpler to understand and implement. We believe this offers a promising direction for future design of low level crash-safe key-value stores, by designing the system for proof automation.

References

[1] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[2] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a database system. Foundations and Trends in Databases, 2007.

[3] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. *USENIX*, 1994.

[4] LevelDB. LevelDB db_bench benchmark. https://github.com/google/leveldb/blob/master/db/db_bench.cc, September 2014.

[5] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–44, San Jose, CA, February 2013.

[6] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*, Madrid, Spain, January 2011.

[7] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 252–269, Shanghai, China, October 2017.

[8] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. Faults in Linux: Ten years later. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–318, Newport Beach, CA, March 2011.

[9] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 497–514, Shanghai, China, October 2017.

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

[11] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage*, 9(3):9:1–32, August 2013.

[12] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016.

[13] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. eXplode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, November 2006.

[14] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, Seattle, WA, November 2006.