GEFS, A Good Enough File System

Ori Bernstein
ori@eigenstate.org

ABSTRACT

GEFS is a new file system built for Plan 9. It aims to be a crash-safe, corruption-detecting, simple, and fast snapshotting file system, in that order. GEFS achieves these goals by building a traditional 9p file system interface on top of a forest of copy-on-write Bε trees. It doesn't try to be optimal on all axes, but good enough for daily use.

1. The Current Situation

Currently, Plan 9 has several general purpose disk file systems available. All of them share a common set of problems. On power loss, the file systems tend to get corrupted, requiring manual recovery steps. Silent disk corruption is not caught by the file system, and reads will silently return incorrect data. They tend to require a large, unshrinkable disk for archival dumps, and behave poorly when the disk fills. Additionally, all of them do O(n) scans to look up files in directories when walking to a file. This causes poor performance in large directories.

CWFS, the default file system on 9front, has proven to be performant and reliable, but is not crash safe. While the root file system can be recovered from the dump, this is inconvenient and can lead to a large amount of lost data. It has no way to reclaim space from the dump. In addition, due to its age, it has a lot of historical baggage and complexity.

HJFS, a new experimental system in 9front, is extremely simple, with fewer lines of code than any of the other on-disk storage options. It has dumps, but does not separate dump storage from cache storage, allowing full use of small disks. However, it is extremely slow, not crash safe, and lacks consistency check and recovery mechanisms.

Finally, fossil, the default file system on 9legacy, is large and complicated. It uses soft-updates for crash safety[7], an approach that has worked poorly in practice for the BSD filesystems[8]. Fossil also has a history (and present) of deadlocks and crashes due its inherent complexity. There are regular reports of deadlocks and crashes when using tools such as clone[9]. While the bugs can be fixed as they're found, simplicity requires a rethink of the on disk data structures. And even after adding all this complexity, the fossil+venti system provides no way to recover space when the disk fills.

2. How GEFS Solves it

GEFS aims to solve the problems with these file systems. The data and metadata is copied on write, with atomic commits. If the file server crashes before the superblocks are updated, then the next mount will see the last commit that was synced to disk. Some data may be lost, by default up to 5 seconds worth, but no corruption will occur. Furthermore, because of the use of an indexed data structures, directories do not suffer from O(n) lookups, solving a long standing performance issue with large directories.

While snapshots are useful to keep data consistent, disks often fail over time. In order to detect corruption and allow space reclamation, block pointers are triples of 64-bit values: The block address, the block hash, and the generation that they were born in. If corrupted data is returned by the underlying storage medium, this is detected via the block hashes. The corruption is reported, and the damaged data may then be recovered from backups, RAID restoration, or some other means. Eventually, the goal is to make GEFS self-healing.

Archival dumps are replaced with snapshots. Snapshots may be deleted at any time, allowing data within a snapshot to be reclaimed for reuse. To enable this, in addition to the address and hash, each block pointer contains a birth generation. Blocks are reclaimed using a deadlist algorithm inspired by ZFS.

Finally, the entire file system is based around a relatively novel data structure. This data structure is known as a Bϵ tree [1]. It's a write optimized variant of a B+ tree, which plays particularly nicely with copy on write semantics. This allows GEFS to greatly reduce write amplification seen with traditional copy on write B-trees.

And as a bonus, it solves these problems with less complexity. By selecting a suitable data structure, a large amount of complexity elsewhere in the file system falls away. The complexity of the core data structure pays dividends. Being able to atomically update multiple attributes in the Bϵ tree, making the core data structure safely traversable without locks, and having a simple, unified set of operations makes everything else simpler. As a result, the total source size of GEFS is currently 8737 lines of code, as compared to CWFS at 15634, and the fossil/venti system at 27762 lines of code (20429 for fossil, with an additional 7333 lines for Venti).

3. Bε Trees: A Short Summary

The core data structure used in GEFS is a Bϵ tree. A Bϵ tree is a modification of a B+ tree, which optimizes writes by adding a write buffer to the pivot nodes. Like B-trees, Bϵ trees consist of leaf nodes, which contain keys and values, and pivot nodes. Like B-trees, the pivot nodes contain pointers to their children, which are either pivot nodes or leaf nodes. Unlike B-trees, the pivot nodes also contain a write buffer.

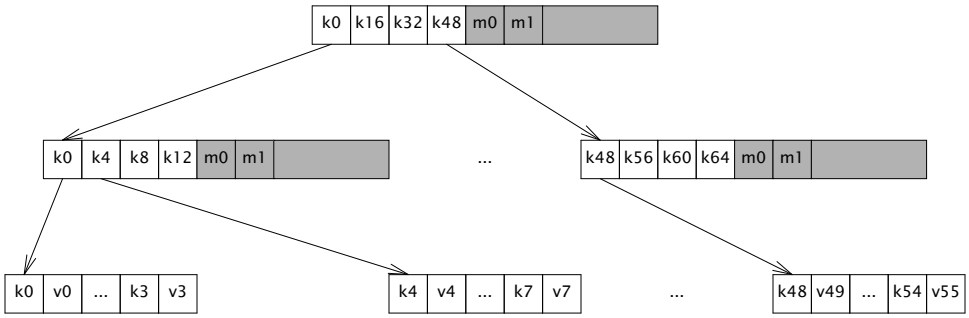
The Bϵ tree implements a simple key-value API, with point queries and range scans. It diverges form a traditional B-tree key value store by the addition of an upsert operation. Upsert operations are operations that insert a modification message into the tree. These modifications are addressed to a key.

To insert to the tree, the root node is copied, and the new message is inserted into its write buffer. When the write buffer is full, it is inspected, and the number of messages directed to each child is counted up. The child with the largest number of pending writes is picked as the victim, and the root's write buffer is flushed towards it. This proceeds recursively down the tree, until either an intermediate node has sufficient space in its write buffer, or the messages reach a leaf node, at which point the value in the leaf is updated.

In order to query a value, the tree is walked as normal, however the path to the leaf node is recorded. When a value is found, the write buffers along the path to the root are inspected, and any messages that have not yet reached the leaves are applied to the final value read back.

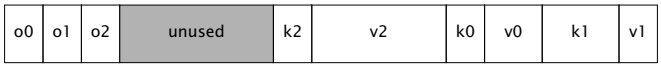
Because mutations to the leaf nodes are messages that describe a mutation, updates to data may be performed without inspecting the data at all. For example, when writing to a file, the modification time and QID version of the file may be incremented without inspecting the current QID; a 'new version' message may be upserted instead. This allows skipping read-modify-write cycles that access distant regions of the tree, in favor of a simple insertion into the root nodes write buffer. Additionally, because all upserts go into the root node, a number of operations may be upserted in a

single update. As long as we ensure that there is sufficient space in the root node's write buffer, the batch insert is atomic. Inserts and deletions are upserts, but so are mutations to existing data.



In GEFS, for the sake of simplicity all blocks are the same size. Unfortunately, this implies that the Bε tree blocks are smaller than optimal, and the disk blocks are larger than optimal.

Within a single block, the pivot keys are stored as offsets to variable width data. The data itself is unsorted, but the offsets pointing to it are sorted. This allows O(1) access to the keys and values, while allowing variable sizes.



In order to allow for efficient copy on write operation, the Bε tree in GEFS relaxes several of the balance properties of B-trees [5]. It allows for a smaller amount of fill than would normally be required, and merges nodes with their siblings opportunistically. In order to prevent sideways pointers between sibling nodes that would need copy on write updates, the fill levels are stored in the parent blocks, and updated when updating the child pointers.

4. Mapping Files to Bε Operations

With a description of the core data structure completed, we now need to describe how a file system is mapped on to Bε trees.

A GEFS file system consists of a snapshot tree, which points to a number of file system trees. The snapshot tree exists to track snapshots, and will be covered later. Each snapshot points to a single GEFS metadata tree, which contains all file system state for a single version of the file system. GEFS is somewhat unique in that all file system data is recorded within a single flat key value store. There are no directory structures, no indirect blocks, and no other traditional structures. Instead, GEFS has the following key-value pairs:

`Kdat(qid, offset)` → `(ptr)`

Data keys store pointers to data blocks. The key is the file qid, concatenated to the block-aligned file offset. The value is the pointer to the data block that is being looked up.

`Kent(pqid, name)` → `(stat)`

Entry keys contain file metadata. The key is the qid of the containing directory, concatenated to the name of the file within the directory. The value is a stat struct, containing the file metadata, including the qid of the directory entry.

`Kup(qid)` → `Kent(pqid, name)`

Up keys are maintained so that '..' walks can find their parent directory. The key is the qid of the directory. The value is the key for the parent directory.

Walking a path is done by starting at the root, which has a parent qid of ~0, and a name of "/". The QID of the root is looked up, and the key for the next step on the walk is constructed by concatenating the walk element with the root qid. This produces the key for the next walk element, which is then looked up, and the next key for the walk path is constructed. This continues until the full walk has completed. If one of the path elements is '..' instead of a name, then the super key is inspected instead to find the parent link of the directory.

If we had a file hierarchy containing the paths 'foo/bar', 'foo/baz/meh', 'quux', 'blorp', with 'blorp' containing the text 'hello world', this file system may be represented with the following set of keys and values:

```
Kdat(qid=3, off=0) → Bptr(off=0x712000, hash=04a73, gen=712)
Kent(pqid=1, name='blorp') → Dir(qid=3, mode=0644, ...)
Kent(pqid=1, name='foo') → Dir(qid=2, mode=DMDIR|0755, ...)
Kent(pqid=1, name='quux') → Dir(qid=4, mode=0644, ...)
Kent(pqid=2, name='bar') → Dir(qid=6, mode=DMDIR|0755, ...)
Kent(pqid=2, name='baz') → Dir(qid=5, mode=DMDIR|0755, ...)
Kent(pqid=5, name='meh') → Dir(qid=5, mode=0600, ...)
Kent(pqid=-1, name='') → Dir(qid=1, mode=DMDIR|0755, ...)
Kup(qid=2) → Kent(pqid=-1, name='')
Kup(qid=5) → Kent(pqid=2, name='foo')
```

Note that all of the keys for a single directory are grouped because they sort together, and that if we were to read a file sequentially, all of the data keys for the file would be similarly grouped.

If we were to walk `foo/bar` then we would begin by constructing the key `Kent(-1, '')` to get the root directory entry. The directory entry contains the qid. For this example, let's assume that the root qid is 123. The key for `foo` is then constructed by concatenating the root qid to the first walk name, giving the key `Kent(123, foo)` This is then looked up, giving the directory entry for `foo`. If the directory entry contains the qid 234, then the key `Kent(234, bar)` is then constructed and looked up. The walk is then done.

Because a Bε tree is a sorted data structure, range scans are efficient. As a result, listing a directory is done by doing a range scan of all keys that start with the qid of the directory entry.

Reading from a file proceeds in a similar way, though with less iteration: When writing to a file, the qid is known, so the block key is created by concatenating the file qid with the read offset. This is then looked up, and the address of the block containing the data is found. The block is then read, and the data is returned.

Writing proceeds in a similar manner to reading, and in the general case begins by looking up the existing block containing the data so that it can be modified and updated. If a write happens to fully cover a data block, then a blind upsert of the data is done instead. Atomically along with the upsert of the new data, a blind write of the version number incremnt, mtime, and muid is performed.

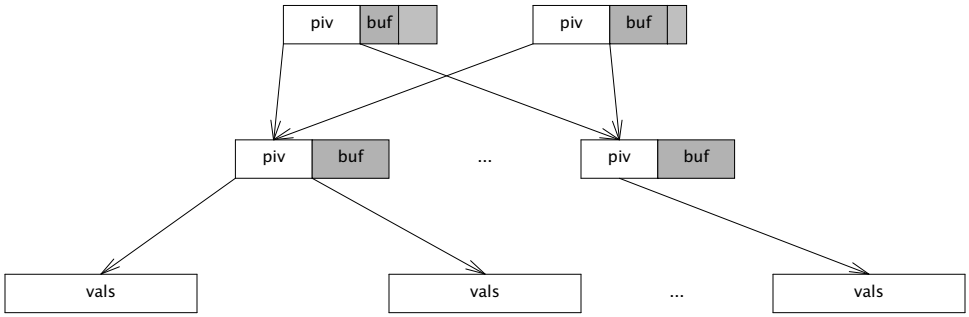
Stats and wstat operations both construct and look up the keys for the directory entries, either upserting modifications or reading the data back directly.

5. Snapshots

GEFS supports snapshots. Each snapshot is referred to by a unique integer id, and is fully immutable once it is taken. For human use, a snapshot may be labeled. The labels may move to new snapshots, either automatically if they point to a tip of a snapshot, or via user intervention. For the sake of space reclamation, snapshots are reference counted. Each snapshot takes a reference to the snapshot it descends from. Each label also takes a reference to the snapshot that it descends from. When a snapshot's

only reference is its descendant, then it is deleted, and any space it uses exclusively is reclaimed. The only structure GEFS keeps on disk are snapshots, which are taken every 5 seconds. This means that in the case of sudden termination, GEFS may lose up to 5 seconds of data, but the data on disk will always be consistent.

If there was no space reclamation in gefs, then snapshots would be trivial. The tree is copy on write. Therefore, as long as blocks are never reclaimed, it would be sufficient to save the current root of the tree once all blocks in it were synced to disk. However, because snapshots are taken every 5 seconds, disk space would get used uncomfortably quickly. As a result, space needs to be reclaimed. An advantage of Bε trees here is that often, only the root block will be copied, and updates will be inserted into its write buffer.



There are a number of options for space reclamation. Some that were considered when implementing GEFS included garbage collection, in the style of HAMMER [3], or optimized reference counting in the style of BTRFS [4], but both of these options have significant downsides. Garbage collection requires that the entire disk get scanned to find unreferenced blocks. This means that there are scheduled performance degradations, and in the limit of throughput, the bandwidth spent scanning must approach the bandwidth spent on metadata updates, as each block must be scanned and then reclaimed. Reference counting implies a large number of scattered writes to maintain the reference counts of blocks.

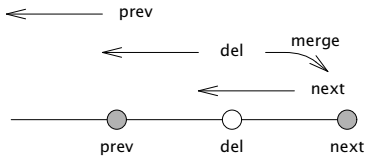
As a result, the algorithm for space reclamation is lifted directly from ZFS [6]. It is based on the idea of using deadlists to track blocks that became free within a snapshot. If snapshots are immutable, then a block may not be freed as long as a snapshot exists. This implies that block lifetimes are contiguous. A block may not exist in a snapshot and be available for reallocation. Thus, when freeing a block, there are 2 cases: Either a block was born within the pending snapshot, and died within it, or it was born in a previous snapshot and was killed by the pending snapshot.

To build intuition, let's start by imagining the crudest possible implementation of snapshot space reclamation. Assuming that block pointers contain their birth generation, we can walk the entire tree. When a block's birth time is <= the previous snapshot, it is referred to by an older snapshot. We may not reclaim it. If the subsequent snapshot refers to this block, then it was born in this snapshot but is still in use. We may not reclaim it. Otherwise, the block is free, and we can reclaim it.

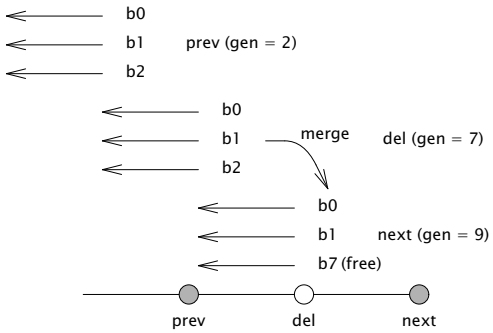
Obviously, this is slow: It involves full tree walks of multiple snapshots. It may walk large numbers of blocks that are not freed.

So, in order to do better, we can keep track of blocks that we want to delete from this snapshot as we delete them, instead of trying to reconstruct the list when we delete the snapshot. When we attempt to delete a block, there are two cases: First, block's birth time may be newer than the previous snapshot, in which case it may be freed immediately. And second, the block may have been born in the previous snapshot or earlier, in which case we need to put it on the current snapshot's deadlist. When the current snapshot is deleted, the current snapshot's deadlist is merged with the next

snapshot's deadlist. All blocks on the deadlist that were born after the previous snapshot are freed.



There's one further optimization we can do on top of this to make deletions extremely fast. The deadlists may be sharded by birth generation. When a snapshot is deleted, all deadlists within the snapshot are appended to the descendant snapshot, and any deadlists with a birth time after the deleted snapshot in the descendant may be reclaimed. With this approach, the only lists that need to be scanned are the ones consisting wholly of blocks that must be freed.



The disadvantage of this approach is that appending to the deadlists may need more random writes. This is because in the worst case, blocks deleted may be scattered across a large number of generations. It seems likely that in practice, most bulk deletions will touch files that were written in a small number of generations, and not scattered across the whole history of the disk.

The information about the snapshots, deadlists, and labels are stored in a separate snapshot tree. The snapshot tree, of course, can never be snapshotted itself. However, it's also a copy on write Bε tree where blocks are reclaimed immediately. It's kept consistent by syncing both the root of the snapshot tree and the freelists at the same time. If any blocks in the snapshot tree are freed, this freeing is only reflected after the snapshot tree is synced to disk fully.

The key-value pairs in the snapshot tree are stored as follows

`Ksnap(id)` → `(tree)`

Snapshot keys take a unique numeric snapshot id. The value contains the tree root. This includes the block pointer for the tree, the snapshot generation of the tree, the previous snapshot of the tree, its reference count, and its height.

`Klabel(name)` → `(snapid)`

Label keys contain a human-readable string identifying a snapshot. The value is a snapshot id. Labels regularly move between snapshots. When mounting a mutable snapshot, the label is updated to point at the latest snapshot every time the tree is synced to disk.

`Kslink(snap, next)` → `()`

A snap link key contains a snapshot id, and the id of one of its successors. Ideally, the successor would be a value, but our Bε tree requires unique keys, so we hack around it by putting both values into the key. When we have exactly one next link, and no labels that point at this snapshot, we merge with our successor.

`Kdead(snap, gen) → (headptr, tailptr)`

A dead key contains a pair of snapshot id and deadlist generation. The value contains a head and tail pointer for a deadlist. These are used to quickly look up and merge deadlists, as described earlier in this paper.

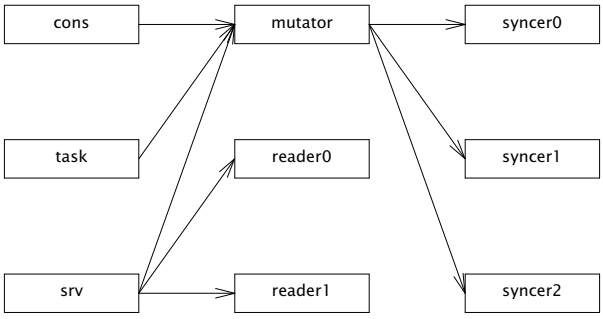
6. Block Allocation

In GEFS, blocks are allocated from arenas. Within an arena, allocations are stored in a linked list of blocks, which is read at file system initialization. The blocks contain a journal of free or allocate operations, which free or allocate regions of disk. When the file system starts, it replays this log of allocations and frees, storing the available regions of blocks in an in-memory AVL tree. As the file system runs, it appends to the free space log, and occasionally compresses this log, collapsing adjacent free or used blocks into larger regions.

Because of the copy on write structure, it's fairly common for metadata blocks to get allocated and deallocated rapidly. Drives (even solid state drives) care a lot about sequential access, so it's beneficial to make a best effort attempt at keeping data sequential. As a result, GEFS selects the arena to allocate from via round robin, offsetting by the type of block. If the round robin counter is 10, and we have 7 arenas, then data blocks (type 0) are allocated from arena 3 ((10+0)%7), pivot blocks (type 1) are allocated from arena 4 ((10+1)%7), and leaf blocks (type 2) are allocated from arena 5 ((10+2)%7). The round robin counter is incremented after every few thousand block writes, in order to balance writes across arenas. Since all arenas are the same, if an arena is full, we simply advance to the next arena.

7. Process Structure

GEFS is implemented in a multiprocess manner. There is one protocol proc per posted service or listener, which dispatches 9p messages to the appropriate worker. Read-only messages get dispatched to one of many reader procs. Write messages get dispatched to the mutator proc, which modifies the in-memory representation of the file system. The mutator proc sends dirty blocks to the syncer procs. There is also a task proc which messages the mutator proc to do periodic maintenance such as syncing. The console proc also sends messages to the mutator proc to update snapshots and do other file system maintenance tasks.



Because the file system is copy on write, as long as the blocks aren't reclaimed while a reader is accessing the tree, writes need not block reads. However, if a block is freed within the same snapshot, it's possible that a reader might observe a block with an inconsistent state. This is handled by using epoch based reclamation to free blocks.

When a proc starts to operate on the tree, it enters an epoch. This is done by atomically taking the current global epoch, and setting the proc's local epoch to that, with an additional bit set to indicate that the proc is active:

```
epoch[pid] = atomic_load(globalepoch) | Active
```

As the mutator frees blocks, instead of immediately making them reusable, it puts the blocks on the limbo list for its generation:

```
limbo[gen] = append(limbo[gen], b)
```

When the proc finishes operating on the tree, it leaves the epoch by clearing the active bit. When the mutator leaves the current epoch, it also attempts to advance the global epoch. This is done by looping over all worker epochs, and checking if any of them are active in an old epoch. If the old epoch is empty, then it's safe to advance the current epoch and clear the old epoch's limbo list.

```
ge = atomic_load(globalepoch);
for(w in workers){
        e = atomic_load(epoch[w]);
        if((e & Active) && e != (ge | Active))
                return;
}
globalepoch = globalepoch+1
freeblks(limbo[globalepoch - 2])
```

This epoch based approach allows GEFS to avoid contention between writes and reads. A writer may freely mutate the tree as multiple readers traverse it, with no locking between the processes, beyond what is required for the 9p implementation. There is still contention on the FID table, the block cache, and a number of other in-memory data structures.

The block cache is currently a simple LRU cache with a set of preallocated blocks.

8. Future Work

Currently, GEFS is buggy, and the disk format is still subject to change. In its current state, it would be a bad idea to trust your data to it. Testing and debugging is underway, including simulating disk failures for every block written. In addition, disk inspection and repair tools would need to be written. Write performance is also acceptable, but not as good as I would like it to be. We top out at several hundred megabytes per second. A large part of this is that for simplicity, every upsert to the tree copies the root block. A small sorted write buffer in an AVL tree that gets flushed when the root block would reach disk would greatly improve write performance.

On top of this, I have been convinced that fs(3) is not the right choice for RAID. Therefore, I would like to implement RAID directly in GEFS. When implementing RAID5 naïvely, there is a window of inconsistency where a block has been replicated to only some devices, but not all of them. This is known as the "write hole". Implementing mirroring natively would allow the file system to mirror the blocks fully before they appear in the data structures. Having native RAID would also allow automatic recovery based on block pointer hashes. This is not possible with fs(3), because if one copy of a block is corrupt, fs(3) would not know which one had the incorrect data.

Furthermore, growing the file system would be extremely useful for taking flashed disk images and growing them to the full size of the underlying storage. The way that arenas work allows for this to happen fairly easily, but right now there's no way to change the list of arenas. Additionally, the naive round robin allocation across arenas works doesn't allow for balancing.

There are a number of disk-format changing optimizations that should be done. For example, small writes should be done via blind writes to the tree. This would allow small writes to be blindingly fast, and give us the ability to keep small files directly in the values of the tree, without allocating a disk block for them.

Deletion of files could also be done via a range deletion. Currently, each block deleted requires an upsert to the tree, which makes file deletion O(n) with a relatively high cost. Moving to a range deletion makes deletion blindingly fast, at the cost of a large amount of complexity: The deletion message would need to split as it gets flushed down the tree. It's an open question whether the benefit is worth the complexity.

Finally, it would be good to find a method of supporting narrow snapshots, where only a range of the file hierarchy is retained.

9. References

[1] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan,

''An Introduction to Bε Trees and Write-Optimization,'' *;login:*,
October 2015, Vol. 40, No. 5" ,

[2] 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,'' *Proceedings of the 13th USENIX Conference on File and Storage Technologies,* 2015

[3] Matthew Dillon, "The HAMMER Filesystem," June 2008.

[4] Ohad Rodeh, Josef Bacik, Chris Mason, "BTRFS: The Linux B-Tree Filesystem" *ACM Transactions on Storage, Volume 9, Issue 3, Article No 9, pp 1–32,* August 2013

[5] Ohad Rodeh, "B-trees, Shadowing, and Clones",

H–0245(H0611-006) November 12, 2006

[6] Matt Ahrens, '' How ZFS Snapshots Really Work,'' *BSDCan,* 2019

[7] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. ''Soft Updates: A Solution to the Metadata Update Problem in File Systems,'' *ACM Transactions on Computer Systems*, Vol 18., No. 2, May 2000, pp. 127–153.

[8] Valerie Aurora, ''Soft updates, hard problems'' *Linux Weekly News*, July 1, 2009, https://lwn.net/Articles/339337/

[9] kvik, *Clone*, https://shithub.us/kvik/clone/HEAD/info.html