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

Plan 9 has several general purpose disk file systems available. While they have served us well, all of them leave much to be desired. On power loss, the file systems may get corrupted. Partial disk failure is not caught by the file system, and reads may 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 perform 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]. 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. Why GEFS Is Good Enough

GEFS aims to solve these problems with the above file systems. The data and metadata are copied on write, with atomic commits. This happens by construction, with fewer subtle ordering requirements than soft updates. 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, but no corruption will occur. Furthermore, because of the use of an indexed data structure, directories do not suffer from O(n) lookups, solving a long standing performance issue with large directories.

The file system is based around a relatively novel data structure: the Bϵ tree [1]. The Bϵ tree is a write optimized variant of a B+ tree. In addition to good overall performance, it plays particularly nicely with copy on write semantics. This allows GEFS to

greatly reduce write amplification seen with traditional copy on write B-trees. The reduced write amplification allows GEFS to get away with a nearly trivial implementation of snapshotting.

As a result of the choice of data structure, 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, each block pointer contains a birth generation. Blocks are reclaimed using a deadlist algorithm inspired by ZFS. This algorithm is described later in the paper.

While snapshot consistency is useful to keep data consistent, disks often fail over time. In order to detect corruption, block pointers contain a hash of the data that they point at. If corrupted data is returned by the underlying storage medium, this is detected via block hashes. And if a programmer error causes the file system to write garbage to disk, this can often be caught early. The corruption is reported, and the damaged data may then be recovered from backups, RAID restoration, or some other means.

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.

3. Bε Trees: A Short Summary

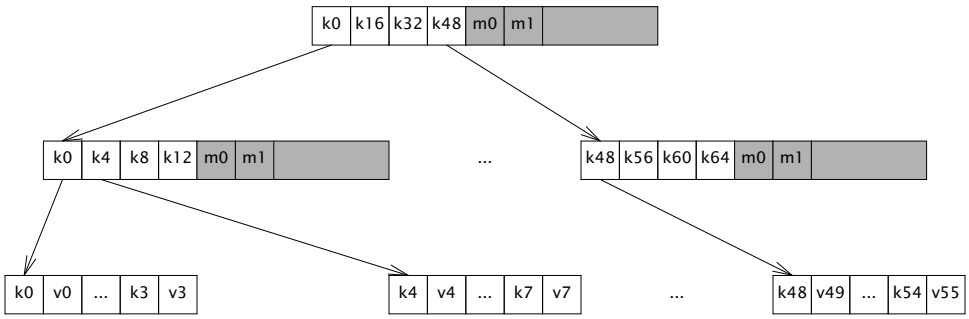
The core data structure used in GEFS is a Bε tree. A Bε tree is a modification of a B+ tree, which optimizes writes by adding a write buffer to the pivot nodes. Like B-trees, Bε trees consist of leaf nodes, which contain keys and values, and pivot nodes. Like B-trees, the pivot nodes contain pointers 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 from a traditional B-tree key value store with 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 into 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. The root's write buffer is flushed into the selected victim. 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.

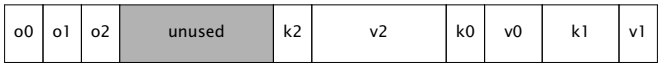
In order to query a value, the tree is walked as normal; however, the path to the leaf node is recorded. When a value is found, the write buffers along the path to the root are inspected, and any messages that have not yet reached the leaves are applied to the final value read back.

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.



For the sake of simplicity, GEFS makes all blocks the same size. This implies that the Bϵ tree blocks are smaller than optimal, and the disk blocks are larger than optimal. The simplifications this allows in the block layer appear to be worthwhile.

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 given an index, or O(log(n)) access while searching, while allowing variable size keys and values.



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 increment, mtime, and muid is performed.

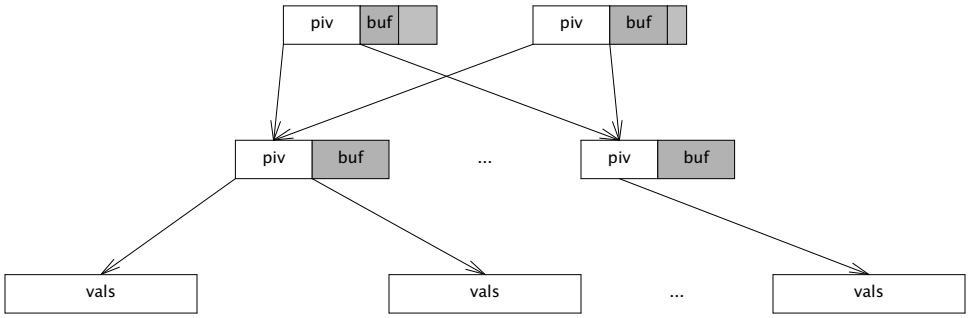
Stats and wstat operations both construct and look up the keys for the directory entries, either upserting modifications or reading the data back directly.

5. Snapshots

Snapshots are an important feature of GEFS. Each GEFS snapshot is referred to by a unique integer id, and is fully immutable once it is taken. Snapshots are labelled with a human readable string. When marked mutable, the labels move to new snapshots as the file system is written to and synced. A snapshot may only be referred to by 0 or 1 mutable labels, along with as many immutable labels as desired.

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.



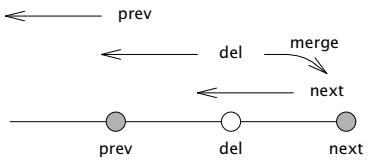
There are a number of options for space reclamation. Some that were considered when implementing GEFS included garbage collection, in the style of HAMMER [3], or optimized reference counting in the style of BTRFS [4], but both of these options have significant downsides. Garbage collection requires that 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 borrowed 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.

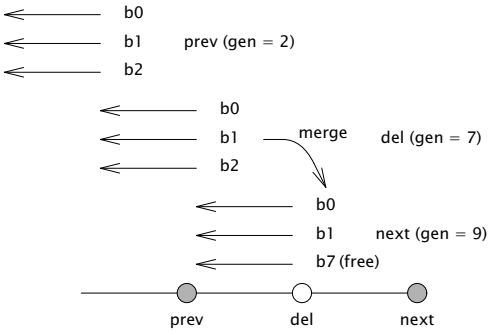
So, in order to do better, we can keep track of blocks that we want to delete from this snapshot as we delete them, instead of trying to reconstruct the list when we delete the snapshot. When we attempt to delete a block, there are two cases: First, the block's birth time may be newer than the previous 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.

All of this assumes that there is a single, linear history of snapshots. However, GEFS allows users to take mutable snapshots off of any label, which breaks the assumption. If the assumption is broken, two different mutable labels may kill the same block, which would lead to double frees. GEFS handles this by adding the concept of a *base* to each snapshot. This base id is the first snapshot in a snapshot timeline. Any blocks born before the base are not considered owned by the snapshot, and no record of their demise will be made in that snapshot. The cleanup is left to the snapshot that was used as the base.



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.

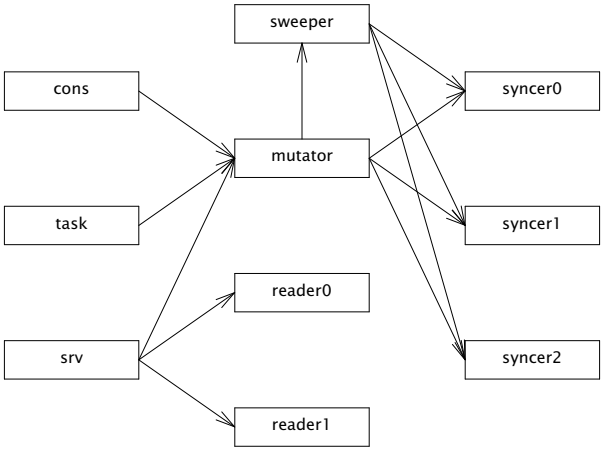
Because of the copy on write structure, it's fairly common for metadata blocks to get allocated and deallocated rapidly. Drives (even solid state drives) care a lot about sequential access, so it's beneficial to make a best effort attempt at keeping data sequential. As a result, GEFS selects the arena 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 are six types of proc that GEFS uses for its operation: The *console*, *dispatch*, *mutation*, *sweeper*, *reader*, and *syncer*. Most of these processes can be replicated, however, there may only be one or *console* at a time. Protocol parsing is handled by one of several dispatch procs. There is one of these per posted service or listener. Each dispatches 9p messages to the appropriate worker, depending on the 9p message type. Read-only messages get dispatched to one of multiple reader procs. Write messages get dispatched to the mutator proc, which modifies the in-memory representation of the file system. The mutator proc generates dirty blocks purely in memory, and sends them to the syncer procs. The job of the syncer proc is simply to write blocks back to disk asynchronously. There are also some tasks that may take a long time, and can be done in the background. These are sent to the sweeper proc. Because the tree is a shared data structure, the sweeper and mutator do not work in parallel. Instead, they must hold the mutator lock to accomplish anything. Finally, the task proc schedules periodic maintenance operations. These include syncing the file system and taking automatic snapshots.

The work of the sweeper could be done by the mutator, and in early versions of the file system, it was. However, some operations such as removing very large files can involve a lot of messages being inserted into the tree, which may block other writers. As a result, the long running operations are better deferred to a background process, which splits them into small chunks, allowing the mutator to make progress between them.

Data flow through these processes is unidirectional, and any block that has made it out of the mutating processes is immutable. This makes it reasonably easy to reason about consistency.



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, a naive implementation would allow the reader to observe a corrupt block. As a result, some additional cleverness is needed: block reclamation needs to be deferred until all readers are done reading a block. The algorithm selected for this is epoch based reclamation.

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 current 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])
```

If the old epoch is not empty, then the blocks are not freed, and the cleanup is deferred. If a reader stalls out for a very long time, this can lead to a large accumulation of garbage, and as a result, GEFS starts to apply back pressure to the writers if the limbo list begins to get too large.

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.

8. Appendix A: Data Formats

The formats used for GEFS on-disk storage are described below. There are several data structures that are described: Superblocks, arena headers, tree nodes, and tree values.

All blocks except raw data blocks begin with a 2 byte header. The superblock header is chosen such that it coincides with the ascii representation of 'ge'.

All numbers in GEFS are big-endian integers, byte packed.

The headers are listed below:

Value	Description
0	Unused
1	Pivot node
2	Leaf node
3	Allocation log
4	Deadlist log
5	Arena Header
0x6765	Superblock header

8.1. Superblocks

Superblocks are the root of the file system, containing all information needed to load it. There is one superblock at offset 0, and one superblock at the last block of the file system. These two superblocks are duplicates, and only one intact superblock is needed to successfully load GEFS. Because the superblock fits into a single block, all the arenas must also fit into it. This imposes an upper bound on the arena count. With 16k blocks, this natural limit is approximately 1000 arenas. Gefs imposes a smaller limit internally, limiting to 256 arenas by default.

header[8] = "gefs9.00"
blksz[4]: the block size for this file system
bufspc[4]: the buffer space for this file system
snap.ht[4]: the height of the snapshot tree
snap.addr[8]: the root block of the snapshot tree
snap.hash[8]: the hash of the snapshot tree root
snapdl.hd.addr: the address of the snap deadlist head
snapdl.hd.hash: the hash of the snap deadlist head
snapdl.tl.addr: the address of the snap deadlist tail
snapdl.tl.hash: the hash of the snap deadlist tail
narena[4]: the number of arenas
flags[8]: flags for future expansion
nextqid[8]: the next qid that will be allocated
nextgen[8]: the next generation number that will be written
qgen[8]: the last queue generation synced to disk
arena0.addr[8], arena0.hash[8]: the location of the 0th arena
arena1.addr[8], arena1.hash[8]: the location of the 1st arena
...
arenaN.addr[8], arenaN.hash[8]: the location of the N'th arena
sbhash[8]: hash of superblock contents up to the last arena

8.2. Arenas

An arena header contains the freelist, the arena size, and (for debugging) the amount of space used within the arena.

type[2] = Tarena
free.addr[8]: the address of the start of the freelist
free.hash[8]: the hash of the start of the freelist
size[8]: the size of the arena
used[8]: the amount of used space in the arena

8.3. Logs

Logs are used to track allocations. They are the only structure that is mutated in place, and therefore is not fully merkelized. There are two types of log in gefs: Allocation logs and deadlists. They share a common structure, but contain slightly different data.

All logs share a common header:

type[2] = Tlog or Tdlist
logsz[2]: the amount of log space used
loghash[8]: the hash of all data after the log header
chainp[24]: the block pointer this log block chains to

8.3.1. Allocation Logs

When the type of a log block is Tlog, the contents of the block are formatted as an allocation log. In an allocation log, each entry is either a single u64int, recording an allocation or free of a single block, or a pair of u64ints, representing an operation on a range of blocks.

The operations are listed below:

Value	Description
1	Allocate 1 block
2	Free 1 block
3	Sync barrier
4	Alloc block range
5	Free block range

Operations are packed with the operation in the low order byte. The rest of the value is packed in the upper bits. For multi-block operations, the range length is packed in the second byte.

```
PACK64(logent, addr|op);
if(op == 4 || op == 5)
        PACK64(logent+8, len);
```

8.3.2. Deadlist Logs

Deadlist logs are simpler than allocation logs. They only contain a flat list of blocks that have been killed.

8.4. The Tree

The tree is composed of two types of block: Pivot blocks, and leaf blocks. The block types were

8.4.1. Pivot Blocks

Pivot blocks contain the inner nodes of the tree. They have the following header. The layout is as described earlier in the paper.

type[2] = Tpivot
nval[2]: the count of values
valsz[2]: the number of bytes of value data
nbuf[2]: the count of buffered messages
bufsz[2]: the number of bytes of buffered messages

8.4.2. Pivot leaves

Within the block, the first half of the space after the header contains a key/pointer set. The head of the space contains an array of 2-byte offsets to keys, and the tail of the space contains a packed set of keys and block pointers.

The offset table is simple:

*off[2*nval]*: the offset table

the keys/pointers are slightly more complicated. They contain a length prefixed key, and a pointer to the child block for that key.

nkey[2]: the length of the key
key[nkey]: the key data
addr: the address of the pointed to block
hash: the hash of the pointed to block
gen: the generation number of the pointed to block

The second half of the space consists of messages directed to a value in the leaf. This is formatted similarly to the key/pointer set, but instead of offsets to key/pointer pairs, the offsets point to messages.

The array of offsets grows towards the end of the block, and the array of values or messages grows towards the start of the block.

The offset table is the same, however, instead of having *nval* entries, it has *nbuf* entries.

*off[2*nbuf]*

The messages contain a single byte opcode, a key, and a message that contains an incremental update to the value.

op[1]: the message operation
nkey[2]: the length of the target key
key[nkey]: the contents of the target key
nval[2]: the length of the message
val[nval]: the contents of the message

8.4.3. Leaf Blocks

Leaf blocks contain the leaf nodes of the tree. They have the following header. The layout is as described earlier in the paper.

type[2]: the block type Tleaf *nval[2]*: the number of key value pairs
valsz[2]: the size of the key value pairs

Within a leaf, the layout is very similar to a pivot. There is a table of key-value offsets, and an array of packed messages. As before, the array of offsets grows towards the end of the block, and the array of values grows towards the start of the block.

*off[2*nval]*: the offset table

Each key value pair is encoded as below:

nkey[2]: the length of the key
key[nkey]: the contents of the key
nval[2]: the length of the value
val[nval]: the contents of the value

8.5. Keys and Values.

In GEFS, keys begin with a single type byte, and are followed by a set of data in a known format. Here are the types of known keys:

Kdat qid[8] off[8] describes pointer to a data block. The value for this data key must be a block pointer. Block pointers are encoded as *addr[8] hash[8] gen[8]*. This entry is only valid in file system trees.

Kent pqid[8] name[n] describes a pointer to a file entry (stat structure). The value must be the body of a dir structure. This entry is only valid in file system trees. The dir structure is structured as below:

flag[8]: flags for future expansion
qid.path[8]: the qid path
qid.vers[4]: the qid version
qid.type[1]: the qid type
mode[4]: the permission bits
atime[8]: the access time
mtime[8]: the modification time
length[8]: the file size
uid[4]: the owning user id
gid[4]: the owning group id
muid[4]: the last user that modified the file

Kup qid[8] describes a pointer to a parent directory. The value is the *Kent* formatted key. This key is the entry of the containing directory. It's only present for directories. This entry is only valid in file system trees.

Klabel name[] describes a label for a snapshot. The value is a *snapid[8]*, referring to a snapid indexed by Ksnap. This key is only valid in snapshot trees.

Ksnap snapid[8] describes a key referring to a snapshot tree. The value is a tree entry. The tree is formatted as:

nref[4]: the number of references from other trees
nlbl[4]: the number of references from labels
ht[4]: the height of the tree
flag[4]: flags for future expansion
gen[8]: the tree generation number
pred[8]: the predecessor snapshot
succ[8]: the successor snapshot
base[8]: the base snapshot
bp.addr[8]: the address of the root block
bp.hash[8]: the hash of the root block
bp.gen[8]: the generation of the root block

Kdlist snap[8] gen[8] describes a key referring to a deadlist. The *snap* field refers to the snapshot that the deadlist belongs to, and the *bgen* field refers to the birth generation of the blocks on the deadlist. The value of the deadlist entry is a pair of block pointers, pointing to the head and tail of the block list.

8.6. Messages

Oinsert and *Odelete* can have any key/value pair as an operand. They replace or remove a key/value pair respectively.

Oclearb inserts a deferred free of a block, without reading it first. It has no value, but the key must be a *Kdat* key.

Oclobber is similar to *Oclearb*, but its operand must be a *Kent* key. *Owstat* updates an existing file entry. The key of an *Owstat* message must be a *Kent*, and the value is a bit field of fields to update, along with the new values. The first byte is a set of wstat flags, and the remaining data is the packed value associated with each flag. It can contain the following updates:

Owsize fsize[8]: update file size
Owmode mode[4]: update file mode
Owmtime mtime[8]: update mtime, in nsec
Owatime atime[8]: update atime, in nsec
Owuid uid[4]: set uid
Owgid uid[4]: set gid
Omuid uid[4]: set muid

Orelink and *Oreprev* rechain snapshots. The key of either of these messages is a *Ksnap*, and the operand is the ID of a new predecessor or successor snap.

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