GCTrees: Garbage Collecting Snapshots

Chris Dragga and Douglas J. Santry
Advanced Technology Group, NetApp Inc.
chris.dragga@netapp.com, douglas.santry@netapp.com

***Abstract*—File-system snapshots have been a key component of enterprise storage management since their inception. Creating and managing them efficiently, while maintaining flexibility and low overhead, has been a constant struggle. Although the current state-of-the-art mechanism, hierarchical reference counting, performs reasonably well for traditional small-file workloads, these workloads are increasingly vanishing from the enterprise data center, replaced instead with virtual machine and database workloads. These workloads center around a few very large files, violating the assumptions that allow hierarchical reference counting to operate efficiently. To better cope with these workloads, we introduce GCTrees, a novel method of space management that uses concepts of block lineage across snapshots, rather than explicit reference counting. As a proof of concept, we create a prototype file system, gcext4, a modified version of ext4 that uses GCTrees as a basis for snapshots and copy-on-write. In evaluating this prototype analytically, we find that, though they have a somewhat higher overhead for traditional workloads, GCTrees have dramatically lower overhead than hierarchical reference counting for large-file workloads, improving by a factor of 34 or more in some cases. Furthermore, gcext4 performs comparably to ext4 across all workloads, showing that GCTrees impose minor cost for their benefits.**

I. Introduction

Storage usage in modern data centers has changed dramatically over the previous decade. The introduction of virtual machines and the proliferation of database deployments have created new demands on storage systems, making data management more important than ever. This paper describes GCTrees, a file-system-agnostic scheme for implementing snapshots that is optimized for such workloads.

Virtualization has had a profound effect on the modern data center. Applications no longer run on dedicated machines with their root file system on a local disk. Instead, physical servers have been virtualized to support many virtual machines sharing hardware. The root file systems of VMs have also been virtualized. A VM's root file system can be encapsulated in a file, called a disk image, and placed on shared storage. The classic sequential/append workload is directed to the file system inside the disk image, but the workload appears to be random I/O to the disk image. There has also been an explosion of OLTP database deployment in recent years, and this too has been affected by virtualization. It is common practice to place a database's data in a LUN accessed by iSCSI, and store the LUN on a shared storage server (the LUN can be represented by a large file internally on the remote storage server). Access to OLTP LUNs is also random. Due to these trends, the historically important workload for shared storage, a "home directory" style workload, which is characterized by sequential reading and writing or appending, has lost much of its importance.

978-1-4673-7619-8/15/$31.00

There are many advantages to employing shared storage. Shared storage offers location transparency, disaster recovery, and advanced data management features. Snapshots have become an indispensable tool for data management. Snapshots offer a consistent read-only point-in-time view of a file system. Such views are important when a consistent backup is required. Snapshots can also be used to conveniently recover from data loss, potentially without the need for a system administrator's assistance.

A space-efficient implementation of snapshots has to manage block sharing well. As a result, snapshot implementations must be able to efficiently detect which blocks are shared. Consider a new snapshot. It is initially identical to the active file system; all of its blocks are shared. When a shared block is updated in the active file system, it is left intact in the snapshot and the new version of the block is placed in the active file system (copy-on-write, or COW); sharing must be detected efficiently to determine if COW is necessary. The same problem exists when a block is deleted: the system must be able to determine quickly if a block can be released or if it is still in use.

The problem of block sharing is usually addressed on a per data structure (file system) ad hoc basis. Storage systems vary widely in their choice of data structures. Consequently, the implementation of snapshots in NetApp® WAFL® [1] is vastly different from that found in FFS [2].

One potentially unifying approach is that of Rodeh's hierarchical reference counting [3]; this is currently considered to be state of the art. Rodeh's methods make minimal assumptions about the structure of the underlying file system and could be superimposed on many file systems that need to support shared blocks. There are two key pieces to the system. First, block usage accounting is expanded from a binary relation of either used or free in a bitmap to wider entries to support multiple references, that is, more than a bit is devoted to the state of a block; we call the resulting structure an Rmap. Second, the reference counts (refcounts) persisted in the Rmap are actually just the lower bounds on the number of extant references to a block—references can also be inherited from a parent in the file-system tree.

To illustrate how the scheme works, consider the situation depicted for a file tree of indirect blocks in Figure 1; leaf nodes contain data and the arrows are indirect pointers stored in indirect blocks (interior nodes). The top block, *A*, has two references to it and thus has a reference count of 2, but its children only have explicit reference counts of 1; the second reference is implied. If the block *D* in R2 is modified, then COW is performed from *D* up the path to the root, *A*; the result is depicted in the right half of Figure 1. Only now do the implicit reference counts become explicit.

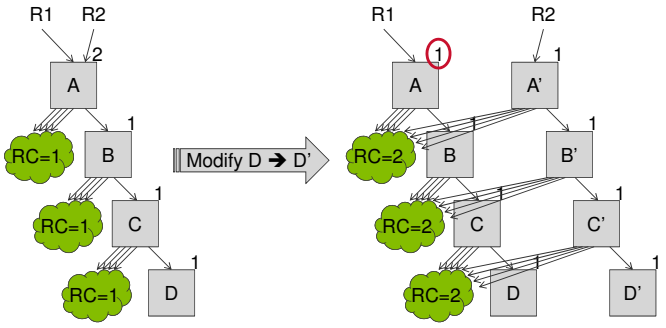


Fig. 1: **Update Storm.** *Behavior of hierarchical reference counting when a single block, D, is modified in a file with a deep metadata tree. The reference count of A changes from 2 to 1, while all other refcounts not directly on the path from A to D change from 1 to 2.*

Hierarchical reference counting is efficient when it can postpone explicitly counting references. A side-effect of the COW is that a number of refcounts must be updated as a result of becoming explicit. The number of refcounts that need to be updated is proportional to the fan-out of the tree multiplied by the height. As the fan-out of such trees can be on the order of hundreds or even thousands, a significant number of refcounts may be updated. These refcount changes need to be propagated to the Rmap, which can generate substantial random I/O. We refer to this I/O as an update storm.

Rodeh conceived a further refinement to hierarchical reference counting to mitigate the update storm: the refcount log [4]. When a refcount change needs to be recorded, it is first logged. The log contains the increments and decrements to each modified refcount—not the absolute counts. When the log is full, the file system processes its entries and propagates them to the Rmap.

Accumulating entries in the log can introduce two mitigating effects (as opposed to simply postponing the update storms). First, opposite refcount deltas may cancel one another out in the log, leaving the net count unchanged and no work to be done. Second, multiple updates to an Rmap block can amortized over a single I/O. It should be noted that in the absence of cancellation in the log, this scheme exacerbates the update storm by unleashing a number of them simultaneously.

A number of artifacts of this scheme are potentially problematic. First, one never truly knows if a block is free. A block with a positive refcount may well be free; we just do not know yet, as the decrements might be waiting in the refcount log. This can be a problem if a file system is desperate for free space. Note, however, that it is impossible for a block to be referenced and have a zero count in the Rmap.

A potential drawback for some workloads is the inherent nature of the algorithm. Consider Figure 1 again: it is clear that the number of refcount updates required is at a maximum when the divergence between two trees is a minimum. The number of refcounts to update is proportional to the number of shared blocks between the trees. For a workload that exhibits a lot of sequentiality, this makes sense (everything changes, or nothing changes). For random I/O, this behavior is the exact opposite of what we want. We would like the amount of work required to maintain the metadata to be proportional to the data that changes, not the inverse. While this might not be a severe problem for the traditional workstation workload, it is not well suited for servers hosting disk images and LUNs.

The contribution of this paper is the introduction of GCTrees, a new system for implementing snapshots in file systems. The target workload is that of large files and random writes, such as disk images and LUNs. The system is suitable for retrofitting to many extant file systems that require snapshots. We describe and evaluate an implementation of GCTrees for Linux ext4. The evaluation shows that GCTrees handle snapshots efficiently while adding little overhead.

II. Architecture

This section presents Generational Chain Trees (GCTrees). The GCTree system consists of two pieces: a graph overlaid on top of the file-system tree and a background process, the scanner, that garbage collects. The GCTree graph represents the relationships between blocks with respect to COW. The scanner examines block relationships to discover blocks to free. Both pieces are described in this section.

Efficient implementations of snapshots require block sharing. When blocks are shared, their states can no longer be captured by a simple binary state of either free or used. Consequently, writing and freeing blocks is complicated by the file system needing to determine the state of a block before making decisions. The GCTrees scheme addresses these problems with a unique approach to tracking the state of blocks.

GCTrees track relationships between blocks instead of directly counting references. When a shared immutable block is updated in a GCTree system, the COW is performed, but instead of performing any sort of reference counting directly, GCTrees records that the new block is descended[1] from the immutable block. As will be shown, this information is sufficient to perform efficient space management and block sharing. An Rmap is not required; the usual form of space management, such as a bitmap, is all that is needed.

To track the relationships between blocks, GCTrees introduce a small amount of metadata to the objects that are shared. Objects that are not leaves in the file-system tree, such as indirect blocks and inodes, must include the metadata. The leaves of the file-system tree, data blocks, do not need the metadata.

Figure 2 depicts the fields in the GCTree metadata. The fields are used as follows: the head pointer is the head of a doubly linked list of blocks that are descended from the block, that is, those blocks that are COWed from it. The source pointer is used to point to the block from which it is descended, that is, the block from which it was COWed. The previous and next fields are used to implement the doubly linked list of descendants. The list of descendants is required to support data structures such as B+ trees. A B+ tree node may split following COW, producing multiple descendants from a single block. File systems whose tree nodes do not split, such as FFS, can safely omit the previous and next pointers. Finally,

[1] We use descend and descendant exclusively with respect to GCTrees, not file trees (in the latter case, we use child or child block).

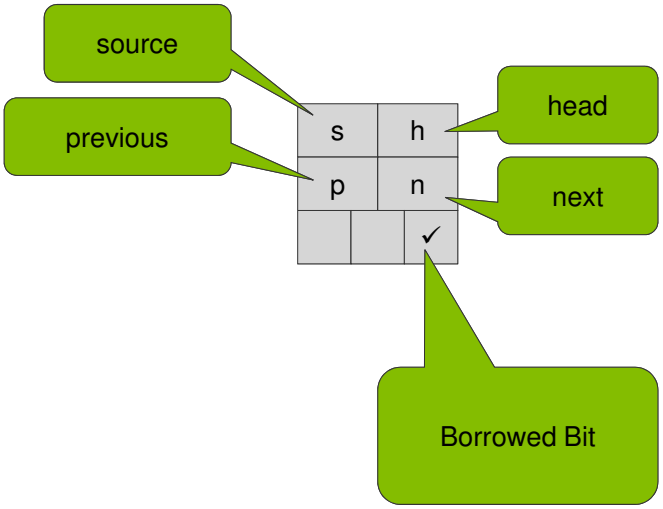


Fig. 2: **GCTree Fields.** *Layout of the fields present in a GCTree metadata item.*

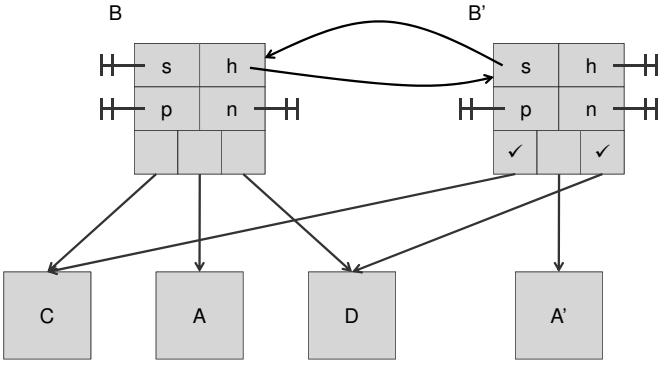


Fig. 3: **GCTree COW Example.** *How GCTrees allow block sharing when COW occurs. A' and B' have been COWed from A and B, respectively; B' still shares blocks C and D with B, however.*

the metadata includes a field of borrowed bits. A bit indicates whether the parent block is the oldest block in the chain to point to a child. As will be seen, this information is used when garbage collecting and deciding to COW.

A. Relating Blocks

To illustrate how these fields are used, Figure 3 presents a simple example. Consider a data block, A, that is a child of the indirect block, B, in a snapshot. A does not include GCTree metadata, because it is a leaf in the file system tree, but B does. Initially all of the GCTree fields in B are null. Now consider the case of updating A in the active file system, so that COW produces a new data block, A'. The indirect block, B, must be updated with the address of A', precipitating a further COW and producing B'. B' is descended from B. The relationships between the blocks are recorded as follows: B' source points to B, B head points to B', and as B' is the only descendant of B, the previous and next pointers remain null.

While B' is in the active file system and points to the new block, A', the remaining children of B' are shared with B. To reflect ownership, the borrowed bits must also be set. Continuing with our example, the pointers that B' inherits have their borrowed bits set as B owns them, but the new block, A', is owned by B', so its borrowed bit is clear. At this point, B' is free to diverge further from B without updating its ancestor again. As new pointers are written to B', their corresponding borrowed bits are cleared. The insight is that the update storm has been replaced with a single I/O to the source COW block when B was first updated in the active file system.

B. Garbage Collecting

In this section we describe how the GCTrees scheme uses the GCTree graph to manage space, one of the most important responsibilities of a file system. At some point, space in the file system will have to be reclaimed. The COW and free operations must efficiently determine the level of sharing before proceeding.

GCTrees use the relationship graph overlaid on the file-system graph to discover free blocks. Files in the active file system and snapshots have the same deletion algorithm applied to determine if blocks are available. Unlink and snapshot deletion remove objects—inodes and root blocks—from the user-visible namespace and pass them to the scanner. The scanner's job is to identify and free blocks inside the objects. Only indirect blocks and inodes are read from disk; data blocks are never read from disk by the scanner. The scanner applies the following algorithm recursively.

Free blocks are identified in two phases. The first phase identifies blocks that are owned by an earlier snapshot; these are then ignored. The second ensures that the remaining blocks are not required by snapshots or the active file system.

The first phase proceeds when the scanner brings in a potential indirect block or inode, or victim, for release. It first identifies a set of candidate blocks for deletion corresponding to the range covered by victim's child block pointers. Then it checks the victim's borrowed bitmap. For each bit set, it removes the child blocks for the corresponding pointer from the candidate set; such blocks are borrowed from an ancestor and are still in use. The candidate set now consists of those blocks that are not required in any past snapshots.

The second phase ensures that a block in the candidate set is not in later snapshots or the active file system, and thus which blocks are safe to delete. The scanner must examine each of the victim's immediate descendants. For each borrowed pointer in a descendant, the scanner checks whether that pointer covers any blocks in the candidate set. If so, the scanner transfers ownership of those blocks to the descendant by clearing the borrowed bit for that pointer and removing the corresponding blocks from its candidate set.

Once the set of free blocks has been finalized, the scanner updates the GCTree pointers of the block's neighbors, setting the head pointer of the victim's ancestor to point to the first descendant and setting the source pointer of the first descendant to point to the ancestor. In addition, if the victim block was part of a descendant list, its descendant blocks must be inserted into this list. Finally, at this point the victim block

can be deallocated. If the remaining blocks in the candidate set are data blocks, these can also be deallocated; otherwise, the scanner must repeat this algorithm on each of these blocks in turn. Note that if a block has no ancestors or descendants, nearly all of these steps can be omitted. In the common case when deleting a snapshot block, the scanner has to perform two reads and writes to maintain the GCTree graph. In the worst case, the length of the descendant list must be traversed; this is rare. Moreover, as will be seen, we have determined empirically that for our target workload this is much cheaper than an update storm.

III. Implementation

Fully validating GCTrees requires testing them in a real file system. To accomplish this, we add GCTrees to Linux's ext4 file system, implementing both copy-on-write and snapshot capabilities in the process, creating a new file system, gcext4. This section begins with a basic overview of ext4's features and then describes our implementation process, highlighting some of the pitfalls that we encountered along the way.

A. Ext4 Background

Ext4 is the fourth and most recent iteration of Linux's ext file system. As it is the default file system of several major distributions, including Ubuntu and Fedora, it has a broad user base, spanning both enterprise and personal use, and is known for its robustness and stability. For these reasons, we choose to implement GCTrees in this file system, even though it lacks support for snapshots and copy-on-write. Before we delve into our changes, we first provide a brief overview of its most relevant details to our work.

In many ways, ext4 bears a strong resemblance to UNIX's original FFS [5]. As in FFS, ext4 divides the disk into fixed-size block groups, each of which contains allocation metadata and a fixed number of inodes and general-purpose blocks. Each file is rooted in an inode, which is uniquely identified by its inode number; file contents are stored in data blocks. Although the file system treats files and directories differently, they use identical data structures on disk and in memory.

Despite these similarities, ext4 diverges in a number of critical ways, adding features like delayed allocation, directory indexing, and journaling. Most importantly, it maps blocks to files using extents, ranges of contiguous blocks as large as 128MB. As in the FFS model, inodes can either point directly to their extents, or they can point to intermediate index blocks; unlike indirect blocks in FFS, these index blocks are arranged in a B+ tree. Although ext4's implementation of most B+ tree operations is minimal—nodes are seldom merged, and rebalancing happens only in a limited fashion—extents provide substantial performance and efficiency gains [6].

As our main focus is to provide a proof-of-concept implementation, we omit some of ext4's more advanced features. In particular, we do not support direct I/O, delayed allocation, or directory indexing. However, because of their importance in file-system robustness and efficiency, we do support both journaling and extents. Although support for journaling is mostly trivial, thanks to the flexibility of ext4's journal, accommodating extents adds noticeable complications.

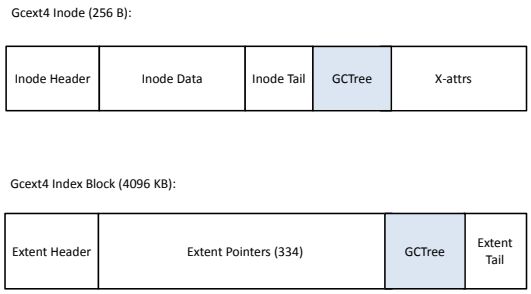


Fig. 4: **GCTree Placement.** *Placement of GCTree metadata in gcext4's on-disk structures. The unshaded areas are those present in unmodified ext4 (note that inode header and tail aggregate a number of separate fields), and the shaded area indicates the location of the GCTree metadata. For inodes, GCTree metadata consumes some of the space for extended attributes, and for index blocks, GCTree metadata consumes several extent pointers.*

B. Implementing Snapshots in ext4

Because ext4 is a file system that only overwrites data in place, adding snapshot support requires implementing COW. Doing so involves several steps: adding GCTree metadata to the appropriate structures, creating and maintaining snapshots, and correctly performing COW. In each of these, we seek to minimize the impact of our changes to ext4's existing functionality, both to simplify our task and to take better advantage of ext4's robust codebase. This section describes our implementation of each of these steps in turn and some of the more serious issues we encountered in this process.

1) Adding GCTree Metadata: Supporting COW snapshots by using GCTrees requires adding metadata to all file-system structures that contain disk pointers. In ext4, these consist of inodes, index blocks, and directory entries, which contain the inode numbers of the files to which they point. For inodes and indirect blocks, we simply inline the GCTree metadata with the existing structures. For directory entries, however, this approach would add prohibitive overhead to each COW, as the file system would have to modify every directory on the path from the newly copied file to the root. Therefore, we employ a different method for directory entries, using a special file, the ifile, to add a layer of indirection between directory entries and inodes.

Embedding GCTree metadata. We represent GCTree metadata as a simple structure containing the source, head, next, and previous pointers and a bitmap indicating which file system pointers are borrowed from an ancestor (see Figure 2). Since disk pointers in gcext4 are 48-bit, the four GCTree pointers occupy 24 bytes. The size of the bitmap varies depending on the number of file system pointers; thus inodes, which have four pointers, use only a byte for their bitmap, whereas index blocks require 42 bytes.

In order to avoid additional random I/O upon copy-on-write, we inline the GCTree metadata in its corresponding structure, as shown in Figure 4. This proves simple for inodes, since ext4 leaves substantial amounts of additional space in its inodes for extended attributes. As these are optional, we appropriate some of this space for GCTree metadata. In

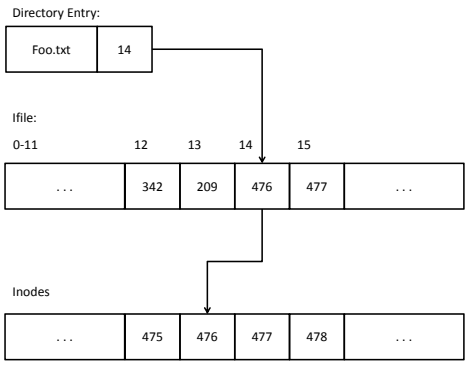


Fig. 5: **Ifile Usage.** *Lookup of the file foo.txt. Its directory entry stores logical inode number 14; this is used to index into entry 14 in the ifile, which points to physical inode 476.*

contrast, index blocks are tightly packed; thus we have to sacrifice several file system pointers, reducing the total from 340 to 334.

Ifile. Handling the addition of GCTree metadata to directory entries is not as simple. COWing a file assigns it a new inode; as inode numbers in ext4 translate directly to physical disk locations, the file thus receives a new inode number. A straightforward implementation would then update the directory entries pointing to the inode with the new inode number, using GCTree metadata to keep track of lineage. This approach, however, ultimately requires that all inode COWs proceed upward through the entire namespace, which does not scale as the file system grows; it also necessitates reverse lookup of directory entries, which is nontrivial in Linux.

We solve this problem by removing physical inode numbers from directory entries, replacing them with logical inode numbers. These logical inode numbers are then used to look up the physical inode numbers in the ifile, a special file that stores the logical-to-physical mappings, as shown in in Figure 5. Logical inode numbers remain unchanged across snapshots; when an inode is COWed, the file system simply changes the physical portion of the mapping. While this may require COWing portions of the ifile, the activity generated is generally substantially smaller and more predictable than COWing the namespace hierarchy.

Because physical inode pointers represent disk pointers, each mapping requires a borrowed bit in order to track its lineage. To simplify our implementation, we use the high bit of each physical mapping for this purpose. We do not require GCTree pointers for each ifile data block, however, because those contained in the parent metadata suffice.

2) Snapshot Management: Although GCTrees provide a foundation for tracking sharing across snapshots, they provide no mechanisms for managing the snapshots themselves. Thus, before implementing copy-on-write, the file system requires a means of creating, accessing, and deleting snapshots. We discuss the first two here, deferring snapshot deletion until Section III-C, where we discuss deletion more generally.

As described in the previous section, the ifile contains pointers to all inodes currently in use by the file system. Thus the ifile, along with the corresponding root directory, uniquely identifies a snapshot and the inodes associated with it, a fact that gcext4 exploits for snapshot creation. Specifically, when the file system creates a snapshot, it immediately COWs both the root inode of the active file system and the current ifile. The original root inode then becomes the root of the new snapshot. To link the root inode with its ifile, the file system stores the physical inode number of the original ifile in an on-disk field of the original root inode. Finally, the file system adds an entry for the new snapshot root to a special snapshot directory, completing the snapshotting process.

Accessing snapshots takes similar advantage of the ifile. By default, `namei` uses the ifile associated with the active file system to perform logical-to-physical inode number translations. However, if it encounters a directory inode with a valid ifile pointer, it recognizes that the directory represents a snapshot root and uses the stored ifile to perform all further translations, switching seamlessly into the snapshot's namespace.

3) Implementing Copy-On-Write: To preserve the data in each snapshot, we add COW capabilities to gcext4's file data and metadata items, using shadow-paging techniques like those in WAFL and btrfs [7]. Unlike these systems, we COW only once per object per snapshot, to avoid disrupting ext4's attempts at preserving spatial locality. Even though shadow-paging is a known technique, we nonetheless encounter several challenges in implementing it in ext4.

The first difficulty in implementing COW in ext4 is correctly determining which items need to be COWed. For this, we rely principally on GCTree metadata: if a block has its borrowed bit set in one of its parents, then it must be COWed. Similarly, if the borrowed bit for a given inode is set in its ifile mapping or in one of the mapping's parent metadata blocks, that inode needs to be COWed. Conversely, if all borrowed bits are clear, it is safe to overwrite.

Although correct, this strategy performs poorly with Linux's page cache, which allows data blocks to be retrieved without checking their metadata upon access. As it is necessary to verify the COW status of a block every time it is written, a naive implementation would simply traverse the metadata tree upon each block write, which would likely add unacceptable CPU and disk overhead. To avoid this, the file system creates a global generation counter that it increments upon each snapshot, as well as individual generation counters for each inode and data block. When a block or inode is brought into memory, the file system checks its metadata to determine whether it needs to be COWed. If so, its counter is initialized to zero; otherwise, its counter is initialized to the global generation number. Upon writing that object, if its counter equals the global generation counter, the file system performs the write in place; otherwise, it COWs the object, updating its counter to the global value when finished.

The other challenge in implementing COW in ext4 is dealing correctly with extents. Because extents can address up to 128 MB, COWing full extents on single block writes would prove prohibitively expensive. Instead, we COW only those blocks that are needed, splitting the extent into multiple parts. The unmodified extents have their borrowed bits set, and the newly written extents do not. Additionally, because ext4 manages extents in a B+ tree, we have to establish GCTree relations across node splits and to reset GCTree pointers during tree growth (we move the parent's GCTree metadata into

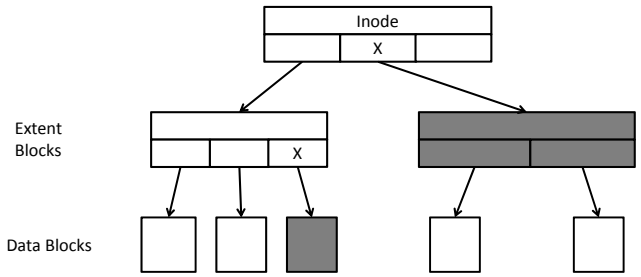


Fig. 6: **Truncate Example.** *Truncation of a file with five data blocks pointed to by two separate index blocks. Shaded blocks represent those blocks to be deleted, and Xs indicate the pointers to be removed. The shaded extent block will be enqueued for deletion, as it completely covers the last two data blocks and it may require GCTree processing; the shaded data block, however, will be directly deleted, since it is not involved in GCTree relationships by definition.*

the new child node and change pointers accordingly). For simplicity, however, we disable movement of pointers into neighboring nodes while performing B+ tree rebalancing if the nodes lack a GCTree relationship.

C. Deletion Scanner

Although GCTrees strive to add minimal overhead to ordinary file-system operations, they do impose additional I/O during file and snapshot deletion, as described in Section II-B. Thus, rather than perform deletions synchronously, we execute them in a background task, the deletion scanner. In this section, we first explain how the scanner handles the deletion of ordinary objects and then describe how it handles the more complex task of snapshot deletion.

1) Active File-System Deletion: Deletion of files and blocks in the active file system is straightforward. The scanner itself consists of a group of kernel threads that monitor a shared message queue. When the user unlinks or truncates a file, the file system places a deletion message onto the queue. A scanner thread then wakes up and performs the deletion, following the algorithm in Section II-B.

Although most of the work of deletion occurs in the background, some foreground actions must still occur. Unlinking a directory entry requires removing it from the parent directory and adjusting the link count of the target inode, both of which may require COWs. Truncating a file is more involved, as the truncated file remains fully accessible in the namespace, allowing the user to reuse the truncated space before the scanner has processed it. To handle this without resorting to superfluous data COWs, truncation searches the file's metadata tree for the highest-level blocks that completely cover a range of bytes to be truncated. If these blocks are metadata, it enqueues them for deletion; otherwise, if they are data, it directly frees them. Finally, it removes the pointers to the blocks to be deleted from the parent. Figure 6 illustrates this process.

Because deletion happens after blocks become inaccessible, special measures are necessary to ensure that deletions are not lost during crashes. For inodes, we take advantage of ext4's orphan list, an on-disk, singly linked list that contains all inodes that are in the process of being deleted. We add a similar list for deleted metadata blocks, repurposing two fields in the extent header after deletion to point to the next block in the list. Maintaining these lists imposes three additional I/Os per enqueued block, but the total impact should be minimal, since we enqueue blocks at the highest possible levels of the file's metadata tree.

2) Deleting Snapshots: Deleting snapshots requires more effort than deleting inodes, although it proceeds similarly. When the user wants to delete a snapshot, he or she simply deletes the root directory for the snapshot, which enqueues a normal deletion message for its inode. When a scanner thread dequeues the message, however, it recognizes that the inode to be deleted is a snapshot root and deletes its ifile before deleting the snapshot root itself.

Ifile deletion follows a special procedure similar to the deletion of ordinary files, but with several additional steps:

1) For each inode mapping in the ifile:
   a) Determine whether the mapping is borrowed by the ifile's successor.
      i) If so, transfer ownership to the successor ifile.
      ii) Otherwise, enqueue a deletion message for the inode and invalidate any existing in-memory structures for the inode[2].
   b) Reduce the size of the ifile by the size of the mapping to prevent the scanner from processing the mapping again if it resumes following a crash.
2) Delete the ifile.
3) Delete the snapshot root inode that points to the ifile.

As described, this procedure risks deleting snapshot roots more than once, since they are assigned logical inodes, and, thus, ifile mappings. We avoid this by marking all snapshot roots borrowed in every ifile, regardless of when they were created. Thus the scan of the ifile skips over every snapshot root, since the ifile does not appear to own that inode. Furthermore, as snapshots are immutable, snapshot roots never lose their borrowed status, ensuring that no erroneous deallocation occurs.

IV. Evaluation

Now that we have outlined the implementation of a file system with snapshots built upon GCTrees, we evaluate how well GCTrees function in practice. In this section, we first examine our prototype's performance against unmodified ext4, demonstrating that our addition of snapshots comes at an acceptable performance cost and that deletion in the background does not harm performance. Then, more importantly, we compare the work that our file system does to that of btrfs and that of a hypothetical hierarchical reference counting system. We find that, although it produces more overhead in NFS-like workloads, our GCTree system requires dramatically fewer writes per operation—in some cases, up to 34 times fewer—for the enterprise-centric workloads that we target.

[2]We do this for simplicity, though it does technically violate POSIX semantics, as existing snapshot file descriptors will return stale error codes when accessed.

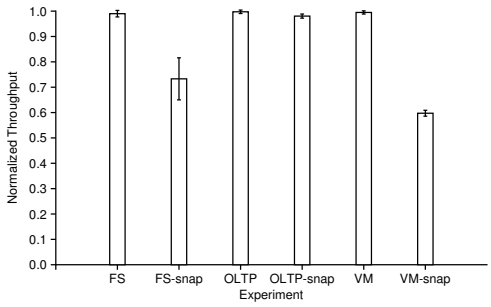


Fig. 7: **Normalized Benchmark Throughput.** *Mean gcext4 throughput for the Filebench fileserver (FS), OLTP, and VM benchmarks, normalized to mean ext4 throughput (7.94, 0.144, and 7.70 MB/s, respectively). Bars labeled "-snap" have snapshots taken during the run. Error bars show normalized standard deviation.*

A. Overall Performance of Gcext4

Before we can begin to analyze the effectiveness of GCTrees themselves, we must show that they can be implemented efficiently in a file system. To do so, we run a series of macrobenchmarks against gcext4 and compare its performance, both with and without actively creating snapshots, to that of its source file system, ext4, showing that gcext4 imposes acceptable overhead. Our test machine consists of a desktop with a 3GHz Intel Core 2 Duo processor, 6GB of RAM, and a 7200 rpm 160 GB Hitachi Deskstar P7K500 hard drive, running the Linux 3.9.4 kernel.

We focus on two benchmarks from the Filebench suite [8], each exercising a different type of workload. OLTP reproduces a typical database workload, performing a combination of small reads and writes to several large files, as well as writes to a log; all writes are synchronous. Fileserver emulates the file-system activity of an NFS server and contains an even mixture of metadata operations, appends, and whole-file reads and writes. To adequately stress the disk, we configure each benchmark to have a footprint at least twice the size of memory (ten 1332MB files for OLTP and 10,000 files for fileserver); we leave the other parameters at their default values. To examine gcext4's performance in virtualized environments, we also configure the fileserver benchmark to run in an Oracle VirtualBox VM, running stock Ubuntu 13.04 with ext4. The disk image is 17GB, providing 12GB of usable space, is fully preallocated, and uses the host buffer cache. We run the benchmark with 9,000 files and refer to this as the VM benchmark.

To test the effect that snapshots have on performance, we choose intervals that reflect those commonly used by enterprises. OLTP workloads are likely to be mission critical, so we snapshot every five minutes during that benchmark. In contrast, we snapshot the fileserver and VM benchmarks once per hour, since they emulate less important workloads.

Figure 7 shows the average throughput of gcext4 and gcext4 with snapshots over five three-hour runs for the fileserver, OLTP, and VM benchmarks. We normalize the values shown to the mean performance of ext4 for each benchmark and indicate the standard deviation of the normalized value with error bars.

Across all benchmarks, performance without snapshots shows no statistically significant difference from ext4, indicating that the indirection added by the ifile comes at little cost. Adding snapshots does hurt performance in some cases: the fileserver and VM benchmarks degrade by 30% and 40%, respectively. In both of these cases, the performance penalty that we observe is unrelated to GCTrees.

The performance degradation that we observe in the fileserver benchmark occurs because we currently synchronize files to disk when their inodes are COWed to ensure that any reads from the snapshotted version contain all data present in the file at snapshot creation time. The fileserver benchmark ordinarily performs no synchronous writes, so the additional synchronizations cause performance to suffer. More sophisticated sharing of pages between inodes and their COWed descendants could likely remove the need for this flush to disk and reduce or eliminate the slowdown we observe. To verify this hypothesis, we remove the synchronization step and repeat the benchmarks with snapshots. We find that performance is on par with gcext4 without snapshots, confirming that the slowdown is an artifact of our implementation and not caused by GCTrees.

In contrast, the VM benchmark is relatively unaffected by additional synchronization. The VM's disk image is the only file in this benchmark, so only one synchronization occurs per snapshot. However, the image file suffers from significant fragmentation over the course of the benchmark due to COW. We verify that this is the case by reexecuting the benchmark on the fragmented VM image with COW disabled. When we do so, the performance remains degraded, indicating that the degradation is unrelated to GCTree I/O. This is a problem for any file system that supports COW—we observe a performance degradation of similar magnitude when executing the same benchmarks on btrfs with snapshots[3]—and can be addressed with defragmentation techniques and online cleaning. As our file system is a prototype, we have implemented neither of these and leave them for future work.

Finally, we observe virtually no performance penalty in the OLTP benchmark. All writes are synchronous in this benchmark and reads are small and random, causing disk I/O from the benchmark to dominate in all cases.

B. Comparison to Hierarchical Refcounts

Although direct performance numbers show that our implementation performs comparably to unmodified ext4, they indicate little about the efficacy of GCTrees themselves compared to the current state-of-the-art method, hierarchical reference counting. To evaluate this, we instrument both btrfs, a modern file system that uses hierarchical refcounts, and gcext4 to measure the number of block writes that each system requires to maintain its snapshot metadata. In addition, as btrfs differs from ext4 substantially (for instance, ext4 writes in place, whereas btrfs never overwrites), we also simulate, within gcext4, a hypothetical hierarchical reference counting system and measure the blocks it would write.

Our comparison focuses on blocks written because this is the factor most likely to affect performance. Under most

[3]See Section IV-B3 for our experimental setup; notably, we set it to COW data once per snapshot, as in gcext4.

conditions, metadata in the working set will be small enough to be cached in memory, whereas metadata writes must always reach the disk to ensure persistence. Even when metadata is not cached, the worst case number of additional blocks read for both systems will be roughly commensurate with the number of blocks written. The only exception to this can occur with snapshot deletion for GCTrees: B+ tree node splits may lead to chains of descendants that have to be read in, even if they are not written. In practice, we have found this latter case to be rare, occupying no more than 0.6% of the total block overhead. Thus, as the number of reads in practice is difficult to measure correctly and because the worst case reads are comparable, we omit these numbers from our totals and report only writes.

In tracking block writes, we are careful to account for the effects of the page cache, which may absorb redundant writes and thus reduce the total write burden. To do so, we only measure blocks that actually reach disk, during either page cache writeback or explicit synchronization. As we use the default writeback settings in our experiments, our results represent a reasonable compromise between durability and write reduction and are likely to reflect overhead observed in practice.

Using this metric, we confirm our hypothesis that, although hierarchical reference counting performs well for traditional small-file workloads, update storms present a problem for modern large-file workloads, dramatically inflating the write overhead. In contrast, although GCTrees add some overhead relative to hierarchical refcounts for traditional workloads, they minimize overhead for modern workloads.

1) Btrfs Background: Btrfs is a new, currently in-development Linux file system based entirely around B-trees. Unlike ext4 and gcext4, which use inodes in fixed locations and a separate metadata tree for each inode, btrfs organizes all of its file metadata per snapshot into a single, reference-counted B-tree. By default, btrfs never overwrites data in place and instead COWs for every write, although this behavior can be disabled for data blocks on a per-file or file-system-wide basis (the `nodatacow` option).

To our knowledge, btrfs provides the only freely available implementation of hierarchical reference counting at this time, using it to support snapshots and clones. We measure the overhead for hierarchical refcounts by tracking all disk writes generated exclusively by increments and decrements to reference counts[4], other than those used for allocation and deallocation.

2) Simulation Rationale and Methodology: While informative, direct comparison to btrfs may be misleading. As mentioned previously, the structure of the two file systems differs dramatically, and thus observed differences may reflect factors other than the characteristics of the block-sharing methods. In addition, its lack of an on-disk log is likely to reduce its ability to cancel refcount changes, potentially inflating the number of writes we observe. Therefore, we examine how hierarchical reference counts would work on top of ext4.

Implementing a counterpart to gcext4 that uses hierarchical refcounts in lieu of GCTrees would require prohibitive effort, so we opt instead to simulate the write activity that such a system would produce, following the full deferred reference counting system outlined by Rodeh [4]. We build our simulation directly within gcext4, adding an in-memory log that stores refcount changes and simulates a durable log on disk. We then instrument each point where gcext4 copies blocks on write, recording the refcount changes that such an operation would incur. To make these changes durable, we assume that the system logs them with a single sequential write of one or more blocks when it commits the current journal transaction. Once the log fills, we sort it and sum the refcount changes for each block. We then record the number of blocks that must be dirtied when writing the nonzero refcounts to their fixed locations in the Rmap, assuming that refcounts are 16-bit. Note that, even though ext4 maps files in extents, we count references at the granularity of individual blocks, because tracking refcounts on a per-extent basis would represent a significant change from ext4's current block-based allocation.

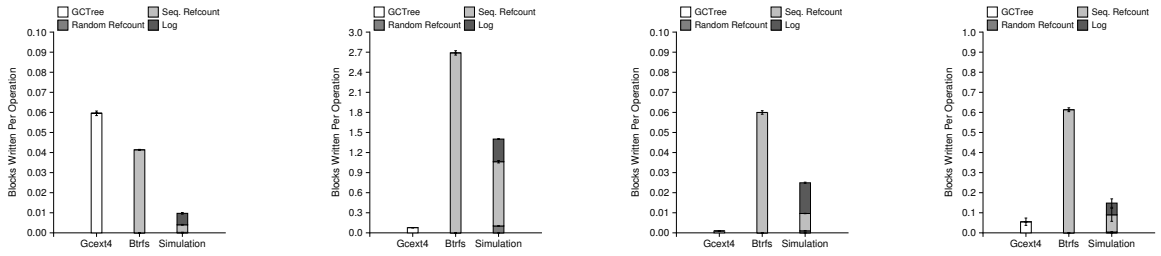
In addition to block refcounts, we also record refcount changes and log writes for inode numbers in the ifile, as inodes need to be reference counted. However, as with btrfs, we do not record refcount changes for the initial allocation or final deallocation of blocks or inodes, because these are necessary for both GCTrees and hierarchical reference counting. Finally, we track sequentiality of writes to the Rmap. We assume that the Rmap is entirely colocated on disk (unlike allocation bitmaps in ext4, which are divided by block group) and that updates are applied in ascending order. Thus we count as random any access that skips at least one block.

3) Basic Results: To compare the two space management techniques, we run gcext4 and btrfs version 0.20-rc1 on the same machine used in Section IV-A. We mount both file systems with access times disabled, and, to bring its behavior closer to that of gcext4, we mount btrfs with the `nodatacow` option. Our simulation uses 15MB for its log, as that yields a reasonable trade-off between the time required to clean the log and the ability to both amortize and cancel refcount updates. We represent refcount changes as 64-bit integers (allowing for both the block address and a flag indicating whether the write is an increment or a decrement), permitting the log to store up to 1,966,080 of them. We assume that the file system processes increments and decrements in the log only when cleaning; although we have experimented with preprocessing log entries during transaction commit, this appears to have little effect in practice.

We use four benchmarks in our experiments: the three described in Section IV-A, using identical parameters, and an open source implementation of the SPC-1 benchmark created by Daniel and Faith [9]. The SPC-1 benchmark uses a 12GB, sparsely allocated file and reproduces the behavior of a LUN hosting a database. This benchmark is time-consuming, so we omit reads, which do not affect our results.

As with our performance measurements, we run each benchmark for three hours and take snapshots at a rate appropriate to the workload being simulated: every hour for the fileserver and VM workloads and every five minutes for the OLTP and SPC-1 benchmarks (measured in simulated time for SPC-1). In addition to write overhead in the active file system, we also measure the additional blocks written when deleting one snapshot produced by each benchmark. To do so,

[4]We treat btrfs's back references interchangeably with refcounts, as they serve the same purposes and are treated similarly in the code.



(a) Fileserver (b) OLTP (c) VM (d) SPC-1

Fig. 8: **Block Write Overhead.** *These plots show the mean block write overhead for GCTrees and hierarchical refcounts in both btrfs and our simulation, normalized by benchmark operations performed. Error bars indicate standard deviation.*

we remount the file system after each experiment and delete the middle snapshot. We repeat each experiment five times.

Figures 8 and 9 depict the block-write overhead in our experiments. Because throughput differs between btrfs and gcext4, we normalize the results from each benchmark by the total operations performed; however, as each snapshot contains roughly the same number of logical blocks, we do not normalize the blocks for snapshot deletion. In addition, we separate the simulation overhead into three parts—random writes to the Rmap (labeled "Random Refcount"), sequential writes to the Rmap ("Seq. Refcount"), and sequential writes to the log ("Log")—as sequential writes are always cheaper in practice than random writes. Note that GCTree writes are always random and, due to the write-anywhere nature of the file system, writes in btrfs are effectively always sequential.

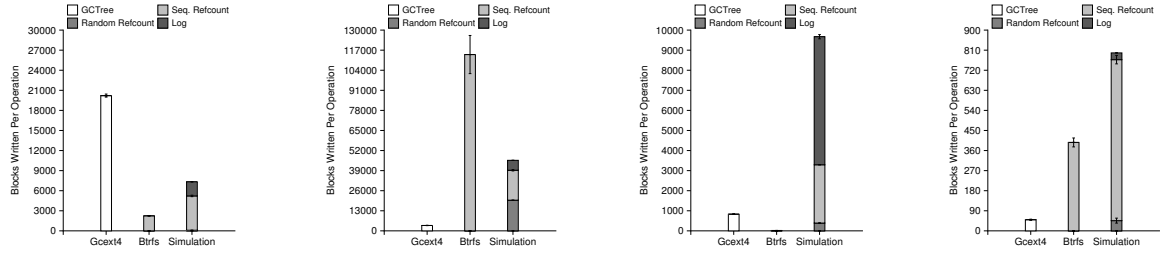
As we anticipated, the degree of overhead we observe is highly workload dependent. Hierarchical reference counting in both btrfs and the simulation is more efficient for the fileserver workload, whereas GCTrees require fewer writes for the VM, OLTP, and SPC-1 benchmarks.

The difference between GCTrees and hierarchical reference counting on the fileserver workload is sizable: GCTrees require, on average, 1.4 times more writes per operation than btrfs and 6.2 times more than the simulation for the active workload. During snapshot deletion, GCTrees require 9.1 times more total writes than btrfs and 2.8 times more than the simulation. Moreover, the vast majority of the I/O in the simulation is sequential, in contrast to the random I/O caused by GCTrees. On examining detailed statistics for these runs, it becomes clear that the low refcount overhead primarily results not from refcount cancellation, but from amortization of nonzero refcounts across blocks when committing them. Even within the simulation, which employs a large log, an average of only 1.5 million refcounts out of 6.2 million total changes cancel; however, the remaining refcount changes require only an average of about 14,000 block writes to persist. Btrfs fares worse, with only about 47,000 cancellations out of 840,000 changes on average, resulting in 220,000 block writes. We observe a similar, though more pronounced, phenomenon with snapshot deletion, in which our simulation sees no refcount cancellation at all and btrfs sees cancellations only from allocation and deallocation.

In contrast, the VM, SPC-1, and OLTP workloads heavily favor GCTrees, as these three are dominated by many random writes to a few very large files. Thus these workloads represent a worst case for hierarchical reference counting: the fan-out per metadata block is generally high, requiring many writes to the refcount log, and refcount changes seldom sum to zero, necessitating a large number of writes when committing them. Conversely, GCTrees require writes proportional to the file's depth, so they are largely unaffected by the high degree of fan-out. While GCTrees do incur additional writes from B+ tree operations, like node splits, these writes are also proportional to file depth and constitute only a small fraction (less than 1%) of the total GCTree-caused writes. As a result, btrfs and our refcount simulation both require more writes per operation for these workloads, spanning a range of 2.7 times more for SPC-1 in the simulation to 34 times for OLTP using btrfs. Snapshot deletion generally follows a similar pattern, although btrfs requires only 18 writes on average for the VM workload. This occurs primarily because a large number of the blocks written are used for both refcounts and allocation and thus are ignored in our tally; if we include these blocks, they exceed those that gcext4 writes.

Accounting for sequentiality mitigates some of the overhead we observe for hierarchical reference counts, as the bulk of the refcount overhead, from both the log and the Rmap, is sequential. This is particularly noticeable in SPC-1; because it uses a sparsely allocated file, gcext4 is able to pack distant logical addresses together on disk and thus experiences few random writes. In contrast, the OLTP and VM-fileserver workloads, which do not use preallocated files, as well as SPC-1 deletion, see similar amounts of random writes using both hierarchical refcounts and GCTrees. Although random writes are more expensive to process, the additional sequential writes are not free, and their sheer volume represents sizable overhead.

4) Varying Reference Counting Parameters: Our results so far indicate a marked dichotomy between the performance of GCTrees and hierarchical reference counting on different workloads. Although hierarchical reference counting requires substantially fewer writes for traditional fileserver and home directory workloads, GCTrees appear to outperform hierarchical reference counting by an even larger margin on large-file workloads. That said, hierarchical reference counting has



(a) Fileserver (b) OLTP (c) VM (d) SPC-1

Fig. 9: **Snapshot Delete Overhead.** *These plots show the mean blocks of write overhead for GCTrees and hierarchical refcounts when deleting a snapshot for each benchmark. Error bars indicate standard deviation.*

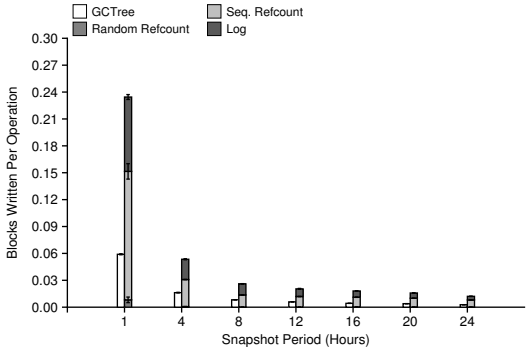


Fig. 10: **Varying Snapshot Period.** *Additional write activity caused by both GCTrees and hierarchical reference counting for the SPC-1 benchmark as the time between snapshots increases. Error bars indicate standard deviation.*

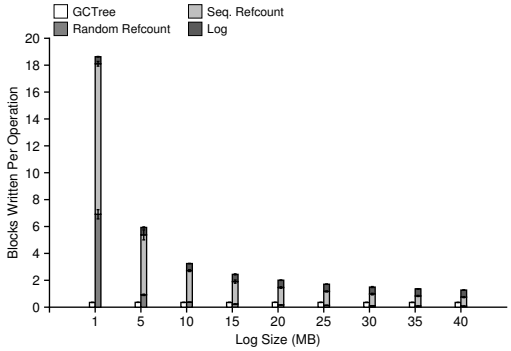


Fig. 11: **Varying Log Size.** *Additional write activity caused by both GCTrees and hierarchical reference counting for the SPC-1 benchmark as the size of the refcount log changes. Error bars indicate standard deviation.*

several parameters—principally, the snapshot period and the size of the log—which, if configured properly, may be able to reduce its overhead to levels comparable to that of GCTrees for those workloads in which it suffers. To analyze this, we execute two experiments varying these parameters on the SPC-1 benchmark, as the gap in overhead for this workload was much smaller than that for the OLTP benchmark. We only compare gcext4 to our simulation, since it exhibits lower write overhead than btrfs, and we increase the execution time of the benchmark to 72 simulated hours, both to permit a broader array of parameters and to better assess the impact of any changes; otherwise, we retain all parameters. Again, we run each experiment five times and report the average and standard deviation of each set of runs.

We first examine how hierarchical refcounts respond to changes in the snapshot period. Hypothetically, a longer interval between snapshots should provide more opportunity for refcount cancellation; thus we would expect the overhead of GCTrees and hierarchical reference counting to converge as the snapshot period grows larger. Figure 10 shows our results for snapshot periods ranging from 1 to 24 hours. As the snapshot period increases, the number of blocks written per operation by both GCTrees and hierarchical refcounts decreases; this occurs because the number of snapshots taken decreases. However, the ratio fails to narrow, hovering between 3.3x and 6.5x, generally increasing as the period lengthens. Similarly, although hierarchical reference counting produces fewer random writes with a longer snapshot period, the overall ratio of random to sequential writes remains roughly unchanged. Thus, even accounting for experimental noise, increasing the snapshot period is unlikely to bring the two systems to parity for this workload.

Increasing the log size may be more likely to reduce write overhead: a larger log may allow more refcounts to cancel and will increase amortization when checkpointing the log. To evaluate this, we run the SPC-1 benchmark with log sizes ranging from 1MB to 40MB and display our results in Figure 11[5]. Increasing the log size yields rapid improvements when the log is small; however, doing so provides increasingly diminishing returns as the log grows larger than 15MB. Random I/O does decrease substantially, but even with a 40MB log, which would require significant effort to checkpoint, hierarchical refcounts still require nearly four times as much total I/O. Further

[5]Blocks per operation are higher here than in Section IV-B3 because the benchmark runs longer, allocating more of the underlying sparse file and thus triggering additional metadata writes.

increases in size are unlikely to provide much additional benefit, indicating that large log sizes cannot eliminate the write gap for SPC-1 here.

V. Related Work

Although we have primarily compared GCTrees to hierarchical reference counting, a plethora of other techniques exist for tracking file history, ranging from simple bitmaps to complex, space-optimized tree structures. In this section, we provide an overview of some of the more notable of these methods.

One of the earliest and most straightforward means of tracking snapshots originates with WAFL [1], which employs per-block bitmaps. Each bit corresponds to a snapshot; if the bit is set, the block is present in that snapshot. The simplicity of this method makes it easy to understand, but it also proves restricting: WAFL supports only a limited number of snapshots and requires a layer of indirection to support clones [10].

The original Plan 9 file system takes a different approach, allowing an effectively unlimited number of snapshots to be stored on write-once media (specifically, optical disks) [11], [12]. As with GCTrees, the system uses flags on blocks in the active file system to determine whether the blocks need to be COWed for a given snapshot; however, because the underlying media is write-once, the system does not concern itself with deletion or space management. Similarly, BlobSeer, a distributed object store using incremental snapshots and copy-on-write that seeks to maximize concurrency, avoids questions of space management by assuming that metadata and data are immutable once written [13]. Although both approaches are convenient, the inability to remove snapshots is prohibitively limiting.

Plan 9 later allowed snapshots to be freed with its Fossil file system, which assigns an epoch number to each block in the file system once it is COWed and thus is no longer part of the active file system [14]. The system then maintains a minimum epoch number that it periodically increments; it deallocates any blocks whose epoch number is below this minimum. This allows space reclamation with relatively low overhead, but at a high cost: only recent snapshots can be kept, requiring an auxiliary system for long-term backups.

In addition, Plan 9 later went on to include content-addressable storage in the Venti system [15]. Content-addressable storage systems ensure that only one copy of a datum exists on a system and it is never over-written. Thus, in this context, snapshots, and hence space reclamation, lie in a different problem domain from copy-on-write storage systems.

Yet another approach can be found in versioning file systems, such as Elephant [16], which tie history to individual file operations and thus tend to have data structures within files that refer back to previous versions. Elephant, for instance, stores the inodes for a versioned file in an "inode log," allowing earlier versions to be read by scanning the log; it is unclear how it handles indirect metadata, however. CVFS [17] employs a similar mechanism for metadata but stores previous versions as a log of deltas to be applied to the current version, improving storage efficiency at the cost of performance. For directories, it employs multiversion B-trees, which store all versions of each directory entry in the tree. Again, this structure was chosen for the sake of minimizing metadata size at the expense of performance in some cases, and may not be appropriate as a general-purpose file system structure.

Hierarchical reference counting for the COW-friendly B-trees described by Rodeh [3] originated as a response to the shortcomings of the prior work described, providing a unified, efficient mechanism for space and version management that allows both full-system snapshots and fine-grained version tracking. Nonetheless, as we have noted, the method has shortcomings of its own, and ours is not the only work to have explored other techniques. Btrfs augments its use of hierarchical refcounts with back-references to facilitate defragmentation and deletion [7]. Twigg et al. [18] note that COW B-tree performance suffers in the presence of updates and present stratified B-trees, a data structure involving a hierarchy of arrays linked with forward references and optimized for updates and sequential access. These bear some semblance to GCTrees, but are much more concerned with optimal access, losing generality.

Finally, it is worth noting that ours is not the first effort to add copy-on-write and snapshot functionality to an ext file system. Next3's approach [19] differs noticeably from ours; it stores snapshots in a special file, which then must be mounted to read. When COWing blocks, Next3 explicitly moves the original contents into the snapshot file and then performs an in-place overwrite, rather than using the shadow-paging approach taken by gcext4.

In contrast, ext3cow [20] uses techniques similar to ours, and thus warrants closer examination. Ext3cow is a file system designed for regulatory compliance, where the evolution of file data over time must be tracked. Thus it exposes a time-shifting interface, where the user accesses snapshotted versions of files by appending earlier timestamps to directory entries. It implements this interface by storing in each inode a pointer to the previous version of the inode, creating a chain which the file system walks to retrieve any prior version of the file. To track which blocks have yet to be COWed in the current version, ext3cow employs a bitmap stored in each inode and indirect block. Together, these features strongly resemble those that gcext4 uses to track lineage; however, the two systems employ them for different purposes. Rather than use its lineage tree to access snapshots, as ext3cow does, gcext4 retrieves them using snapshot root inodes that point to previous ifiles. Conversely, gcext4 uses lineage as a means of deleting snapshots and reclaiming their space, an operation that ext3cow does not allow and that would not work well with ext3cow's time-shifting semantics. Other, less important differences exist as well: ext3cow does not use an ifile, instead relocating old versions of inodes before overwriting them, and, because it is based in ext3, it lacks facilities to handle B+ trees and extents.

VI. Conclusion

Snapshots constitute a crucial component of any storage-management solution, offering rapid recovery from user error and a stable platform for data backups. However, managing space in snapshots is a challenging task, and it is unclear whether a single solution exists that can provide flexibility and

efficiency for all file-system workloads. As our analysis shows, although logged hierarchical reference counts create relatively little overhead for traditional workloads, they collapse under the large-file workloads that are coming to dominate data centers.

To better accommodate these kinds of workloads, we present GCTrees, a file-system-agnostic alternative to hierarchical reference counts. Rather than track references directly, GCTrees track the lineage of pointers across snapshots on a block-by-block basis. Using this information, the file system can quickly determine whether a given block is shared by multiple snapshots, facilitating deallocation and COW decisions. As a proof of concept, we implement GCTrees within ext4, creating a prototype file system with dramatically lower write overhead for database and VM-style workloads than comparable hierarchical reference counting implementations, including that used in btrfs.

Despite the efficacy of our prototype for these workloads, our implementation is still nascent, with many possible improvements worth exploring. Support for clones—that is, writable snapshots—should be possible using the next and previous pointers, and GCTrees should have little trouble supporting fine-grained snapshots of file-system objects, such as directories or individual files. We have yet to investigate either of these features in detail, however, and they may require additional machinery to manage properly.

Ultimately, we expect GCTrees to become a full-fledged alternative to hierarchical reference counts. At this point, the choice of technique will be decided by the anticipated workload; we doubt that any algorithmic tweaks will ever bridge the substantial dichotomy between the two. Our results underscore the need for file-system designers to carefully consider their target use-cases; at least in the case of snapshot management, a universal solution may be untenable. Increasingly, as enterprise file systems move away from traditional workloads, we anticipate that GCTrees will be the more appealing choice.

VII. Acknowledgments

We would like to thank John Strunk for his technical advice and oversight, as well as our anonymous reviewers for their comments.

References

[1] "File system design for an NFS file server appliance." in *Proceedings of the USENIX Winter Technical Conference*, ser. USENIX Winter '94.

[2] M. K. McKusick and G. R. Ganger, "Soft updates: A technique for eliminating most synchronous writes in the fast filesystem," ser. USENIX ATC '99. Berkeley, CA, USA: USENIX Association, 1999.

[3] O. Rodeh, "B-trees, shadowing, and clones," *Trans. Storage*, vol. 3, no. 4, pp. 2:1–2:27, Feb. 2008.

[4] ——, "Deferred reference counters for copy-on-write b-trees," IBM, Tech. Rep. rj10464, 2010. [Online]. Available: http://domino.watson.ibm.com/library/Cyberdig.nsf/papers/B7C80D4AF7CB08DF85257712004C5228/$File/rj10464.pdf

[5] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for unix," *ACM Trans. Comput. Syst.*, vol. 2, no. 3, pp. 181–197, Aug. 1984.

[6] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux Symposium*, vol. 2, 2007, pp. 21–33.

[7] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *Trans. Storage*, vol. 9, no. 3, pp. 9:1–9:32, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2501620.2501623

[8] "Filebench." [Online]. Available: http://filebench.sourceforge.net/wiki/index.php/Main_Page

[9] S. Daniel and R. Faith, "A portable, open-source implementation of the spc-1 workload," in *Proceedings of the IEEE International Workload Characterization Symposium, 2005*, ser. IISWC-2005, Oct 2005, pp. 174–177.

[10] J. K. Edwards, D. Ellard, C. Everhart, R. Fair, E. Hamilton, A. Kahn, A. Kanevsky, J. Lentini, A. Prakash, and K. A. Smith, "FlexVol: flexible, efficient file volume virtualization in WAFL," in *USENIX 2008 Annual Technical Conference*, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 129–142.

[11] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs," in *Proceedings of the Summer 1990 UKUUG Conference*, 1990, pp. 1–9.

[12] S. Quinlan, "A cached WORM file system," *Software: Practice and Experience*, vol. 21, no. 12, pp. 1289–1299, Dec. 1991.

[13] B. Nicolae, G. Antoniu, L. Bougè, D. Moise, and A. Carpen-Amarie, "BlobSeer: next-generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, pp. 169–184, Feb. 2011.

[14] S. Quinlan, J. McKie, and R. Cox, "Fossil, an archival file server." [Online]. Available: http://www.cs.bell-labs.com/sys/doc/fossil.pdf

[15] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," 2002.

[16] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir, "Deciding when to forget in the elephant file system," *SIGOPS Oper. Syst. Rev.*, vol. 34, no. 2, pp. 18–19, Apr. 2000.

[17] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger, "Metadata efficiency in versioning file systems," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, ser. FAST '03. Berkeley, CA, USA: USENIX Association, 2003, pp. 43–58.

[18] A. Twigg, A. Byde, G. Miłoś, T. Moreton, J. Wilkes, and T. Wilkie, "Stratified b-trees and versioned dictionaries," in *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, ser. HotStorage'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 1–5.

[19] Amir G., "NEXT3 snapshot design," Jul. 2011. [Online]. Available: http://sourceforge.net/projects/next3/files/Next3_Snapshots.pdf/download

[20] Z. Peterson and R. Burns, "Ext3cow: A time-shifting file system for regulatory compliance," *Trans. Storage*, vol. 1, no. 2, pp. 190–212, May 2005.