Algorithms and Data Structures for Efficient Free Space Reclamation in WAFL

Ram Kesavan, Rohit Singh, Travis Grusecki
Netapp
ram.kesavan@gmail.com {rh0,travisg}@netapp.com

Yuvraj Patel
University of Wisconsin-Madison
yuvraj@cs.wisc.edu

Abstract

NetApp®WAFL®is a transactional file system that uses the copy-on-write mechanism to support fast write performance and efficient snapshot creation. However, copy-on-write increases the demand on the file system to find free blocks quickly; failure to do so may impede allocations for incoming writes. Efficiency is also important, because the task may consume CPU and other resources. In this paper, we describe the evolution (over more than a decade) of WAFL's algorithms and data structures for reclaiming space with minimal impact on the overall storage appliance performance.

1 Introduction

A file system controls the storage and retrieval of data and metadata. It typically carves up its persistent storage into addressable blocks and then allocates and frees these blocks to process client requests. Efficient free space management is a crucial element of file system performance. Enterprise-class file systems demand consistently high performance for reads and writes. NetApp, Inc. is a storage and data management company that offers software, systems, and services to manage and store data, including its proprietary Data ONTAP® operating system [12]. Data ONTAP implements a proprietary file system called Write Anywhere File Layout (WAFL) [11]. WAFL is a transaction-based file system that employs copy-on-write (COW) mechanisms to achieve fast write performance and efficient snapshot creation.

Like other modern file systems such as FFS [16], XFS [22], ZFS [17], ext3, ext4 [15], and brtfs [19], WAFL tracks allocated and free space for two basic reasons: to enable the file system to find and allocate free blocks to accommodate new writes; and to report space usage to the system administrator to guide purchasing and provisioning decisions. Unlike ext3 and ext4, which write data in place, WAFL, ZFS, and btrfs never overwrite a block containing active data or metadata in place; instead, a free block is allocated and the modified data or metadata is written to it. Because the old data is preserved during a transaction, the process of creating a snapshot is simplified. However, each overwrite also obsoletes the older block (if that block does not belong to any snapshot). Such file systems have to track and reclaim these unneeded blocks at pretty much the same rate as the rate of incoming writes to ensure sufficient availability of free space for new allocations.

The metadata that tracks free space can either be persistent or maintained in memory. Storing this metadata only in memory has the downside that it needs to be rebuilt whenever the machine reboots or the file system is brought online. Therefore, most file systems choose to persist this metadata. Traditional file systems keep their free space metadata up to date. Alternatively, a file system can choose to let its free space metadata go stale, and periodically scan the entire file system to recompute it. Although there is an up-front performance cost to keeping the metadata up to date, this choice provides a convenient pay-as-you-go model for free space reclamation work. An asynchronous recomputation of free space can avoid the up-front performance cost, but file system designs need to cope with potentially stale free space metadata, including how that stale information is reported to the administrator. Additionally, scanning the entire file system to recompute free space only gets more expensive as file system size and activity increases. WAFL has always chosen to keep its free space metadata up to date, except when it comes to snapshots. This requires a free space reclamation infrastructure that can keep up with performance requirements without taking significant CPU and storage I/O away from servicing client operations.

In this paper, we present that infrastructure and show how it satisfies high performance requirements. We describe how it has evolved to keep up with the increase in the file system size while accommodating storage effi-

ciency features like snapshots, clones, deduplication, and compression. We also explain the exception case; that is, how free space is reclaimed asynchronously and efficiently after snapshots are deleted. We do not discuss how free space is reported to the administrator.

We first present the challenges of free space reclamation for a modern enterprise file system with diverse requirements. We then explore the problems and their corresponding solutions as well as their evolution. We analyze these solutions by using performance experiments. Rather than writing a consolidated evaluation section, we chose to include the relevant evaluation in each section; we believe that this approach improves the readability of the paper. Along the way, we present some lessons learned from building and maintaining a feature-rich file system that runs on hundreds of thousands of systems that service a wide range of applications.

2 Background & Motivation

This section presents the challenges of tracking and reclaiming free space to enterprise-quality file systems, which are multi-TiB in size and can process gibibytes per second (GiB/s) worth of operations.

2.1 How Do Blocks Get Freed?

Copy-on-write (COW) file systems never modify data or metadata in-place. Instead, they write out the the new version to a free location, leaving the previous version intact. When combined with a transactional model for writing out a collection of modifications, the file system on the persistent media remains consistent even after a power loss. A COW model means that an overwrite almost always makes the previous version of the block unnecessary unless it belongs to a snapshot. Thus, a 1 GiB/s overwrite workload generates 1 GiB/s worth of potentially free blocks. This means that a COW file system has to (1) find and allocate 1 GiB/s worth of free space for new writes, and (2) modify metadata to record 1 GiB/s worth of freed blocks to keep the file system up to date. The updates to the metadata also generate new allocations and potential frees.

When a file is deleted the file system needs to eventually reclaim all of its blocks. Certain applications create and delete files in bursts. Reclaiming that space in a predictable fashion is a significant challenge.

The ability to create and retain snapshots of a file system is crucial to the life cycle of data management; snapshots are used for data protection, replication, recovery, and so on. When snapshots are deleted, all blocks that uniquely belong to them need to be reclaimed by the file system. Storage efficiency features like compression and deduplication often run in the background to reduce space consumption and free blocks.

A file system must choose a rate for reclaiming free blocks generated from the above activities that satisfies the customer's need for accurate reporting (for provisioning and purchasing decisions) while minimizing performance impact to client operations.

2.2 Lazy Reclamation of Free Space

Traditionally, file systems keep their free space metadata up to date. For instance, if objects in the file system reference block *b*, and if all of these references are dropped in a transaction to the file system, then the free space metadata that is written out by that transaction would indicate that *b* is free. In other words, the metadata persisted by a transaction accurately tracks all the deleted references to blocks. Thus, the update cost increases directly with the number of deletes. This provides a pay-as-you-go model for free space reclamation work.

Alternatively, file systems can choose a consistency model in which the free space metadata can become stale but in a conservative fashion. In this model, dropping a reference becomes very simple; at the end of the transaction from the previous example, the file system metadata indicates that *b* is still allocated even though no objects refer to it. In the background, a periodic scan walks every object in the file system to rebuild a map of blocks referenced by all its objects. This idea has been explored in a research setting [9] and commercially [24]. Using standard hard disks with a typical user workload, scanning these metadata blocks can take 2 to 4 seconds per GiB of user data [9]. Under these assumptions, it would take more than a day to scan a 50 TiB collection of user data before any free space could be reclaimed. Modern storage efficiency techniques like compression, deduplication, cloning, etc., pack more user data into usable storage, and allow user data blocks to be pointed to by different metadata blocks or files. This results in more metadata for the same amount of used storage. Thus, even when hard disks are replaced by faster media like SSDs, the reclamation scan of a multi-TiB file system would still take hours to complete. While this may be acceptable with relatively static datasets, it quickly becomes untenable under heavy write workloads and with increasing file system sizes. The file system would need to hold aside a large reserve of free space to hedge against unpredictable reclamation times. For example, a workload with an overwrite rate of 200 MiB/s would produce more than 17.2 TiB of writes over the course of a day. A file system with 50 TiB of user data would need to keep around 34% free space just to ensure that overwrites suc-

ceed, even if the background scanner is running as fast as possible.

Therefore, WAFL chooses to pay the continual cost of keeping its metadata up to date in all cases, except for snapshots. Section 5 describes how space is reclaimed after snapshot deletion.

2.3 Cost Of Freeing Blocks In WAFL

This section is a brief introduction to WAFL. The Write Anywhere File Layout (WAFL) is a UNIX style file system with a collection of inodes that represent its files [11, 8]. The file system is written out as a tree of blocks rooted at a superblock. Every file system object in WAFL, including metadata, is a file. A file in WAFL is a symmetric n-ary tree of 4 KiB blocks, where the level of the tree is $\lceil log_n \frac{filesize}{4KiB} \rceil$. The leaf node ($L_0$) holds file data, the next level node (L_1) is an indirect block that refers to L_0s, and so on. WAFL is a COW file system, where every modified block of a file is written to a new location on storage. Only the file system superblock is ever written in place. One advantage of this method is that new allocations can be collected together and written out efficiently.

As buffers and inodes are modified (or *dirtied*) by client operations, they are batched together for performance and crash consistency. Every mutable client operation is also recorded to a log in nonvolatile memory before it is acknowledged; the operations in the log are replayed to recover data in the event of a crash. WAFL collects the resultant dirty buffers and inodes from hundreds of thousands of logged operations, and uses a checkpoint mechanism called a *consistency point* (*CP*) to flush them to persistent media as one single and very large transaction. Each CP is an atomic transaction that succeeds only if all of its state is successfully written to persistent storage. Updates to in-memory data structures are isolated and targeted for a specific CP to ensure that each CP represents a consistent and complete state of the file system. Blocks that are allocated and freed for a CP are also captured as modifications to the allocation metadata files that are written out in that same CP. Once the entire set of new blocks that belong to a CP is persisted to storage, a new file system superblock is atomically written in-place that references this new file system tree [8, 11].

File systems use different data structures like linked-lists, bitmaps, B(+) trees, etc., to track free space information, and incorporate the cost of maintaining such structures in their designs. Their block allocators either use these structures directly or build secondary structures to help find space efficiently. WAFL uses bitmap files as the primary data structure to track the allocated or free state of a block. For example, the i^{th} block of the file system is free if the i^{th} bit in the *activemap* file is 0; the block is allocated if the bit is 1. WAFL uses a 4 KiB block size, and therefore, a 1 GiB WAFL file system needs a 32 KiB activemap file, a 1 TiB WAFL file system needs a 32 MiB one, and so on. Clearly, the metadata for multi-TiB sized file systems is too large to fit in memory - the WAFL buffer cache [7] - and needs to be read on-demand from persistent media. WAFL uses auxiliary structures based on the bitmaps to speed up block allocation, but that is outside the scope of this paper.

WAFL allocates blocks that are colocated in the block number space, which minimize updates to the activemap. However, frees may be distributed randomly over the number space, and all such updates to the activemap have to be written out in the same transaction. The more random the updates, the larger the number of dirty activemap buffers for that CP to process. This prolongs the transaction, which negatively affects the write throughput of the file system.

In the rest of this paper, we discuss the techniques that let WAFL handle the nondeterministic nature of block free processing in a way that ensures smooth and deterministic system performance.

2.4 Free Space Defragmentation

Reclaiming free space is not necessarily the same as generating contiguous free space. As a file system ages, the free space in that file system gets fragmented as random blocks are freed while others stay allocated or trapped in snapshots. Contiguous free space is important for write performance and subsequent sequential read performance.

Various ways exist to defragment free space. File systems like LFS [20] use techniques like *segment cleaning* to reclaim contiguous free space. Some file systems put aside a large reserve of free space or recommend holding the space usage below a certain percentage in order to provide acceptable levels of free space contiguity and performance. Some file systems provide tools to defragment free space. WAFL implements both background and inline free space defragmentation. However, techniques for free space defragmentation are outside the scope of this paper.

3 Free Space: The Basics

Let us first define the consistency model of the activemap. As explained in earlier sections, the superblock points to a self-consistent tree of blocks that define a WAFL file system. This means that the metadata in the tree accurately describes the allocated blocks of the tree.

Definition 1. Every tree of blocks contains an activemap of bits; its i^{th} bit (referred to as a_i) is set iff the i^{th} block b_i of the file system is in use by that tree.

3.1 Reuse of Freed Blocks

Lemma 1. *In a COW file system, a block freed during transaction CP_n can only be allocated for use in a subsequent transaction. In other words, it cannot be allocated to a new block written out by transaction CP_n.*

Proof. Transaction CP_n gets persisted only when its new superblock is written in-place; a disruption discards all intermediate state. Recovery is accomplished by starting with the last persistent state, CP_{n-1}, and reapplying changes computed by replaying the operations in the nonvolatile log. Therefore, CP_{n-1} must remain intact until the CP_n is complete. Any free and reuse of a block by CP_n violates this invariant. ■

Thus, when bit a_i is cleared in CP_n, the WAFL block allocator cannot use b_i until the superblock of CP_n has been written out. This is implemented quite simply by creating a second copy of the corresponding activemap 4 KiB buffer when the first bit in it is cleared. The two copies are called *safe* and *current*; the block allocator consults the former and bits are cleared only in the latter. The block allocator sets bits in both copies when recording allocations. The current copy is eventually written out by the CP and the safe copy is discarded when the CP completes. Therefore, any activemap buffer dirtied due to a block free needs twice the memory, and this gets expensive if a large number of random blocks get freed in one CP. Although this state could be maintained in a more memory-efficient manner, it would result in longer codepaths for consulting the activemap. Section 4 describes a more elegant solution to the random update problem.

3.2 A Cyclic Dependency

The activemap file is a flat self-referential file. Like all files, it is composed of blocks and those blocks are by definition also tracked by the activemap. An activemap block written to b_j *covers* activemap block b_i if bit a_i resides in b_j.

Definition 2. An *activemap chain* that starts with a specific activemap block is defined as the longest list of activemap blocks where the $(i+1)^{th}$ block in the list covers the i^{th} block.

By definition, each activemap block can belong to only one unique activemap chain. Thus, assuming that no snapshots are present, when an activemap buffer is dirtied and the subsequent CP (allocates a new block and) frees the old block corresponding to that activemap block, it must dirty the next element in the chain. If that buffer is not yet dirty, this step is repeated. This continues until a previously dirty activemap buffer is encountered. Thus, when any block is freed and its activemap bit is cleared, the entire chain for that activemap block gets updated. In theory, all blocks of the activemap could belong to a single chain; in the worst case, a CP might dirty and write out the entire activemap file! As mentioned earlier, the activemap of a multi-TiB file system might not fit into main memory.

It should be noted that long chains are not easily formed because WAFL often allocates blocks that are colocated in the activemap; they are only formed by very unlikely sequences of allocations and frees. Three solutions have been built in WAFL to solve this problem.

1. **Prefetch the entire chain**. WAFL prefetches the entire chain when an activemap buffer is dirtied. Ideally, the prefetches complete by the time the CP starts allocating new blocks for the activemap file of the file system; the CP processes metadata towards its end, so this works in the right circumstances.

2. **Preemptively break chains**. A background task preemptively finds each moderately long chain and dirties its first block; the subsequent CP breaks that chain.

3. **Postpone the free of the old block**. When a CP processes a dirty activemap buffer, it simply moves the reference to the old block to a new metafile called the *overflow file*. The L_1s of the overflow file serve as an appendlog for free activemap blocks, and the CP avoids dirtying the next element of the chain. Once the CP completes, the blocks in the overflow file are now really freed; their corresponding activemap bits are cleared. Thus, an activemap chain of n elements gets broken down completely in at most n consecutive CPs. This ensures consistent CP length without violating Definition 1.

Section 3.4 discusses the evolution of these solutions in WAFL.

3.3 Processing File Deletion

To process a file deletion operation, WAFL moves the file from the directory namespace into a hidden namespace. However, the space used by the hidden file is not reclaimed; this does not violate Definition 1. Then, potentially across several CPs, the file gradually shrinks as WAFL frees the blocks of this file incrementally. Eventually, the entire tree of blocks is freed and the inode for the hidden file reclaimed. The updates to the activemap are more random if the file has been randomly overwritten, because its tree will refer to random blocks; we call these *aged files*.

3.4 Evaluation and Some History

It should be clear by now that if a very large number of random blocks are freed in a single CP it will need to process a lot of dirty activemap buffers. Long activemap chains can make a bad situation worse.

WAFL was originally designed in 1994 for file sharing and user directory workloads over NFS [21] and CIFS [10], which is characterized by the sequential read and write of many small files. The maximum file system size was 28 GiB at the time; the activemaps for these file systems spanned approximately 200 blocks. The problems described in this section were not applicable for these particular workloads with such small quantities of metadata.

More than two decades later, WAFL hosts a very wide range of applications, from file-intensive applications to virtual machine images and databases over both file-based protocols like NFS/CIFS and block-based SCSI protocols [18], which do random I/O to fewer large files. During this timespan, the sizes of individual disks (hard drive and solid state) and file systems have exploded. The maximum file system size supported by WAFL has increased from 28 GiB in 1994 to 16 TiB in 2004, and to 400 TiB in 2014. Today, the activemaps for the largest WAFL file systems span millions of blocks.

As a result, the problem of random updates to metadata became acute. Larger and larger fractions of the available CPU cycles and I/O resources were being consumed for processing frees. Some applications, like the ones used for electronic design automation, needed to delete large numbers of files in bursts without compromising the performance of other workloads on the system. Clearly, this was not ideal. Section 4 describes the solutions to this problem.

In 2000, WAFL used two mechanisms in tandem – prefetching the activemap chain and preemptively breaking activemap chains – to avoid the cyclic dependency problem. By 2010, WAFL supported file systems that were 100 TiB in size. So, these mechanisms were replaced by the overflow file, which works well for file systems of any size. The overflow file mechanism provided an interesting technique for postponing deletion without changing the consistency semantics of the activemap. Section 4 presents designs that use this technique to convert many random updates to the activemap into fewer and predictable bunched updates.

4 Batching Updates to Metadata

As explained earlier, when a large number of random blocks are freed in a CP, they can generate many metadata updates. If these updates can be accumulated, they can be sorted into more efficient metadata updates. And, if this work can be done over the course of multiple CPs, the cost can be spread out over time in a predictable fashion.

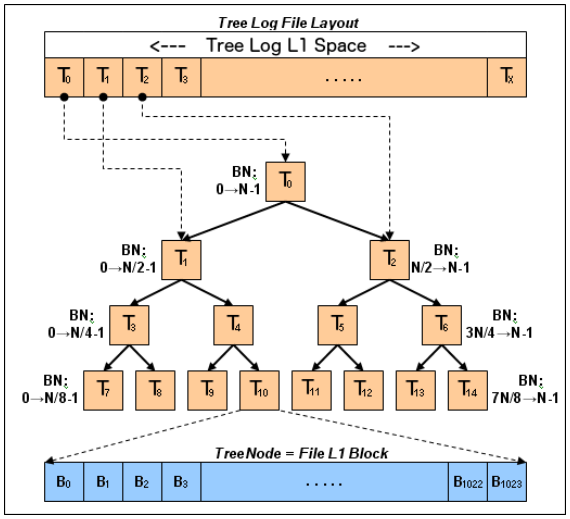

Figure 1: Structure of the TLog.

Much like the overflow file, the two structures described in this section postpone the freeing of blocks by taking ownership of them; the blocks are referenced by the L_1s of these files, which preserves Definition 1. We call this *delete logging*. File system consistency checking treats these structures like regular files. Sorting occurs by (partially) ordering the references by block number across the L_1s of these files. Once they have been sufficiently sorted, the references to the blocks are punched out from the L_1s, and the corresponding activemap bits cleared in a batched fashion.

4.1 The Tree Log

The first manifestation of a delete logging mechanism in WAFL was the *Tree Log* (or *TLog*). As Figure 1 shows, the L_1s (which reference the blocks that the TLog owns) of the TLog form the nodes of a binary tree, where each node contains block references lying in a fixed range of the block number space. A left child covers the left half of its parent's range, and a right child covers the right half.

A delete-logged block is inserted into the root node. If the root node fills up, its contents are pushed to the children nodes based on the range they cover. If either child fills up, the process cascades down the tree. If a leaf node of the tree fills up, all the blocks in it are punched out and the activemap is updated. The size of the TLog determines the range covered by a leaf node. For ex-

ample, if a file system has n blocks and each activemap block covers 32K blocks, a TLog sized to have $n/32K$ leaf nodes ensures that each leaf node covers a single activemap block. In typical deployments, a TLog sized to approximately 1.5% of the file system size provides significant performance improvement; each leaf node covers 4 to 6 activemap blocks.

The TLog structure had one significant problem: there was no easy way to tell how many nodes would be updated due to an insertion. That was a function of which part of the binary tree was affected by an insertion and the fullness of the nodes in that part of the tree. In extreme cases, a small number of insertions would suddenly cause a large number of nodes to be updated, which would result in the CP needing to process many more dirty TLog buffers than if the frees had not been delete-logged. To alleviate these worst-case events, the infrastructure would prefetch the activemap buffers (as it would if delete logging were off), and when it detected one of these bad situations it would stop delete-logging and free directly instead. In practice, this situation was very rare and not reliably created in our testing. However, after it was seen in the field we decided to gradually move away from the TLog structure to a new *Batched Free Log* (or *BFLog*) solution.

4.2 The Batched Free Log

Instead of embedding a sorting data structure in its L_1s, the BFLog adopts a different approach. It accumulates delete-logged blocks and, after some threshold, it sorts and frees them.

This can be achieved in several ways. In WAFL, it is accomplished by using three files: (a) the *active log*, (b) the *inactive log*, and (c) the *sorted log*. A block is delete-logged by appending to an L_1 of the active log. Once the active log fills to a certain threshold, it becomes the inactive log and it is replaced by an empty active log. The delete-logged blocks in the inactive log are sorted across all of its L_1s. This is accomplished in two steps: sorting the blocks within a fixed number of L_1s at a time, followed by a merge-sort, which also moves the blocks to the sorted log. Once the sorted log is ready, its blocks are punched out and the activemap is updated in a bunched fashion. It should be noted that all processing of BFLog data structures is localized or sequential in nature, which is important for its performance.

Once the BFLog reaches steady-state, it becomes possible to pace its operations such that, on the average, each append of a new block results in one block being sorted and/or one block being merge-sorted and/or one block being punched out of the sorted log. This is in stark contrast to the unpredictability of the TLog, where the sorting was strictly controlled by the contents of the binary tree. Rigorous experimentation showed that sizing the BFLog (all three files together) to 0.5% of the file system provided sufficient performance boost.

4.3 Logging in a Reference Counted File System

Several storage efficiency features of WAFL depend on the property that a block can be shared by multiple files or within the same file. The extra references to a block are tracked by a *refcount* file, which is a flat file of integers, one per block. When a multiply referenced block is freed, the refcount file is updated, and because the file is several times less dense than the activemap, the problem caused by random frees of blocks increases manifold. The now ubiquitous deduplication feature results in highly reference counted file systems, which makes delete-logging even more critical. The delete-logging of a multiply referenced block is no different from that of a regular block; when punching out the block the refcount file is updated for all but the last reference.

4.4 Evaluation

Although we have anecdotal data from a few customer systems that show the unpredictable behavior of the TLog, we do not have reproducible experiments to demonstrate it. Because the TLog has now been replaced by the BFLog, we present BFLog data only to show the merits of delete-logging.

We first studied the benefits to random overwrite workloads. A set of LUNs [23] was configured and several clients were used to generate heavy random 8 KiB overwrite traffic to the LUNs; this simulates database/OLTP writes. Although the benefit of the BFLog was a bit muted on high-end platforms (we observed a roughly 5% to 10% improvement in throughput), they were higher on mid-range platforms. Figure 2 shows results on a mid-range system with 12 cores (Intel Westmere) and 98 GiB of DRAM.

Without delete logging, our throughput plateaued at approximately 60k IOPs, whereas with delete logging we were able to continue to about 70k IOPs; this represents approximately a 17% improvement in throughput. In addition to a throughput improvement, we also observed anywhere from a 34% to 48% improvement in latency across most load points. These improvements were achieved via a 65% reduction in our metadata overhead, because the BFLog was able to batch and coalesce a large number of updates to metadata blocks.

The SPEC SFS [4] is not capable of generating the sort of file deletion load that some of our customers do. We

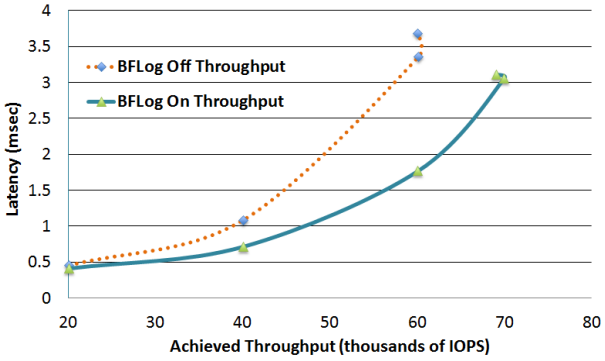

Figure 2: The benefits of delete logging for a random overwrite workload on a mid-range system.

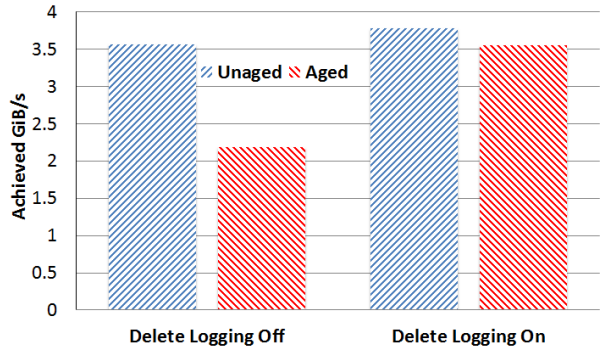

Figure 3: The benefits of delete logging on a high-end all-SSD system for the file deletion throughput workload.

fashioned a custom benchmark that measures file deletion throughput as the rate at which an otherwise idle system reclaims space after a number of large files totalling 1 TiB get deleted in short order.

Figure 3 shows the results of the delete throughput test with and without BFLog on an all-SSD high-end system with 20 cores (Intel Ivy Bridge) and 128 GiB of DRAM. This was repeated with files that had been significantly aged by randomly overwriting them for two hours. The blocks freed from an aged file are more randomly distributed over the number space. The baseline runs (without the BFLog) show that 2 hours of aging results in a 40% drop in delete throughput.

Although the BFLog improved delete throughput by a modest 6% in the unaged data set, it improved delete throughput by over 60% in the aged one. It shrunk the difference between the aged and unaged results from 40% to 6%, which shows that it offset the randomness in the blocks caused by aging. It should be noted that using solid state undersells the improvement. The benefits are much higher on a system with hard drives, because random IOs to load the activemap blocks result in a much bigger difference without delete-logging. Additionally, if this were a highly deduplicated dataset (which it wasn't) the improvement would have been much larger because the BFLog would batch the updates to the less dense refcount file. This result shows the primary benefit of delete logging, i.e., reordering random metadata found in aged file systems so that it appears to be sequential, as found in unaged file systems.

Without delete-logging, increasing delete throughput requires forcing deletes to run faster, which in turn results in more random reads of activemap blocks and more work for the CP to write them out. This takes valuable I/O and CPU resources away from client workloads. Delete-logging allows us to increase delete throughtput without hurting client workloads.

5 Snapshots and Free Space

Snapshots are a crucial building block for many data management features. A file system snapshot is a point-in-time version of the entire file system. In WAFL, it is created quite simply by storing an extra copy of a CP's superblock. None of the blocks in the snapshot can be freed until the snapshot is deleted. This section explains how free space reclamation protects and reclaims the blocks held in snapshots.

We introduce a new term, *volume*, which includes the current file system and all of its snapshots. The current file system is the active part of the volume that services all client operations whose modifications are being written out by the CP mechanism. Snapshots are denoted by S_i, and we use S_0 to denote the current file system.

Clearly, the activemap in S_0 cannot really tell us which blocks are trapped in a given snapshot. Initially the activemap of S_0 is literally the same tree of blocks as the activemap of a new snapshot, S_1, but it quickly diverges due to changes in S_0. The blocks of S_1 are already recorded in its activemap (called *snapmap*), which remains unchanged due to the COW nature of the file system. Therefore, a block used by S_1 can get freed in S_0, and then be marked as free in the activemap.

Lemma 2. *The set of blocks pointed to by a volume can be represented by the bit-OR of its activemap together with the snapmaps of all its snapshots.*

Proof. Since a snapshot is simply a tree of blocks pointed to by a CP's superblock, based on Definition 1, each snapmap accurately captures the set of blocks pointed to by that tree. Therefore, the bit-OR of the activemap with all the snapmaps accurately represents the set of blocks pointed to by the volume. ■

5.1 The Summary Map

A block in the volume is free iff it is free in the activemap and in all the snapmaps. The latter condition needs more IO and CPU to ascertain. There are several options for solving this problem: a file that stores a bitmask per block with one bit per snapshot, a B+tree that captures free/allocated blocks in each snapshot in some searchable but condensed fashion, etc. WAFL uses a bitmap file called the *summary map*, which is a bit-OR of the snapmaps of all snapshots in the volume. This simple choice allows for inexpensive space tracking.

Space reclamation after a snapshot deletion happens lazily but, as this section shows, efficiently. When a snapshot is deleted, some bits in the summary need to be cleared, and doing that atomically (as part of a single CP) is impractical. There are other reasons for the summary to become instantaneously stale, but in a conservative way. WAFL supports a feature called *volume snaprestore*, in which a volume can jump backwards in time to an older snapshot, say S_i. This is accomplished rather trivially by taking the superblock of S_i and writing it out as the volume's superblock in the next CP. The summary map at the time of S_i's creation had been frozen, and after the snaprestore, it becomes the summary of S_0. However, this summary can include blocks that belong to snapshots older than S_i that have since been deleted.

5.2 Snapshot Creation

When a snapshot is created, the summary map has not yet been updated with the new snapshot's blocks. A background *snap create scan* now walks and bit-ORs the snapmap into the summary. It should be noted that this scan is idempotent, and so it can safely restart from the beginning after a reboot.

Theorem 1. *Assuming no deleted snapshots, the set of blocks pointed to by a volume is always guaranteed to be equal to the bit-OR of the activemap and summary even while the snap create scan is in progress.*

Proof. Although the summary does not necessarily include the blocks of a newly created snapshot, S_1, this does not violate Lemma 2, because all blocks that belong to S_1 are also recorded in the activemap at the instant of S_1's creation. To maintain the invariant, if a block is to be freed in the activemap and if it is also shared by S_1, it must be recorded in the summary. In other words, if the snap create scan hasn't gotten to that summary bit yet, it is done on demand. This stops once the snap create scan has completed its task. ■

There are few more interesting cases here. The newly created snapshot S_1 could get deleted before the scan is done, but let us delay that discussion till Section 5.4. A new snapshot S_2 could get created before S_1's scan is complete. In that case, the scan restarts but switches to bit-OR'ing S_2's snapmap into the summary. There is no longer a need to process S_1's snapmap because all blocks that belong to S_1 either belong to S_2 or were freed before S_2 was created. Section 5.5 shows how this consultation as well as the entire snap create scan can be eliminated!

5.3 Another Cyclic Dependency

Another cyclic dependency comes into play while the snap create scan is in progress. Suppose that snapshot S_1 gets created and so the activemap is identical to S_1's snapmap. Now, b_i gets freed and so a_i is cleared and the i^{th} bit in the summary is set by the snap create on-demand work. This results in a dirty summary buffer, which means that the old version of that summary block b_j has to be freed. So a_j gets cleared, which in turn sets the j^{th} summary bit, and so on. Thus, although unlikely, long activemap-summary chains can get created, which is twice as bad as the activemap chain problem. One customer system hit this problem on an aged volume of size 80 TiB. The CP uses the overflow file to postpone the free of the old summary block and thereby avoids dirtying the next activemap in the chain.

5.4 Snapshot Deletion

When a snapshot is removed from the namespace its blocks may or may not be reflected in the summary map. A background *snap delete scan* walks the summary to remove blocks that belong exclusively to that snapshot. This scan is also idempotent and can be safely restarted at any time.

Theorem 2. *Independent of the status of the snap create and delete scans, the set of blocks pointed to by a volume is always equal to or a subset of the bit-OR of the activemap and summary map.*

Proof. Theorem 1 proved that the bit-OR is equal to the set of blocks pointed to by a volume independent of the status of the create scan. While the snap delete scan is in progress, the summary is guaranteed to include all the bits that are set in the deleting snapshot's snapmap unless the snapshot got deleted before its corresponding snap create scan was able to completely bit-OR its snapmap into the summary. Then, either the block still lives in S_0, in which case its activemap bit would be set, or it has since been freed, in which case, there's no need for the summary to protect it if it was uniquely owned by the deleted snapshot. ■

Lemma 3. *Let all snapshots in a volume (including S_0) be sorted in temporal order of creation. For any block b_i in the volume, the bit string created by concatenating the i^{th} bit of all the snapmaps – in the case of S_0, the activemap – in that temporal order will yield a $0^*1^*0^*$ pattern. This is called the 010 snapshot property.*

Proof. Once allocated in S_0, a block can belong to subsequent snapshots until it gets freed in S_0. Once freed in S_0, it is unavailable for allocation until the activemap and all snapmaps say it is free. So, it cannot belong to subsequent snapshots until all snapshots that it belongs to have been deleted. Hence, the bit pattern. ■

There are two ways for the delete scan to update the summary, one of which uses the 010 property.

1. **Deletion by Subtraction (or dbys)**. This scan uses the 010 property to process the deletion of S_i. If a block is used by S_i but not by either of its temporal neighbor snapshots, then it exclusively belongs to S_i. In other words, the scan clears a summary bit if the bit in S_i's snapmap is set but the bits in the neighbor snapshots' snapmaps are not. The youngest snapshot has no younger neighbor and the oldest snapshot has no older neighbor in this scheme. Suppose that a second snapshot S_j is deleted while the dbys scan is in progress. A separate scan can process that S_j, and because the result is idempotent, both scans can work independently as long as they never concurrently update the same bit. If S_j happens to be S_i's neighbor, then the same scan replaces S_j with the next neighbor and continues onwards. Now the scan is updating the summary on behalf of both deletions. When the scan hits the last bit it completes S_i's deletion, but it wraps around and continues processing S_j's deletion until it gets back to where it was when S_j was deleted.

2. **Deletion by Addition (or dbya)**. This scan rebuilds the summary by simply bit-OR'ing the snapmaps of all remaining snapshots. If a second snapshot gets deleted while the scan is in progress, the scan can be restarted for the remaining snapshots. In practice, it is better for the scan to continue down the bitmap space to leverage the readaheads of the snapmaps and summary that have already been issued. When the scan gets to the last bit in the summary, it then wraps around and continues until it gets back to where the second deletion occurred.

As mentioned earlier, both modes of the delete scan can restart after a reboot without affecting correctness. Section 5.2 mentioned one interesting case. What happens if the youngest snapshot S_1 is deleted before its snap create scan is complete? Suppose that S_2 is the next youngest snapshot. It is possible that S_1 got created while the create scan for S_2 was still in progress, in which case, the scan had switched to using S_1's snapmap. Therefore, after the deletion of S_1, the create scan needs to redo S_2.

5.5 Evaluation

The CPU and I/O requirement for all these scans is somewhat straightforward; they need to load the necessary snapmap and summary blocks, and bit manipulation is not too expensive. Since the scans walk the block number space linearly, the necessary blocks can be easily prefetched in time. In theory, a given scan can be parallelized at any granularity because each bit of the summary is independent of the other. In practice, our MP programming model [6] allows for ranges of the summary to be updated concurrently. In most cases, the delete scans can be throttled to run at slow speeds so they don't interfere with client traffic; even on loaded systems, a scan of a 1 TiB sized volume (with 8K bitmap blocks) usually completes in a few minutes. In rare cases, snapshots are deleted to create space in the volume, and the scans need to run fast. We do not present the infrastructure for pacing and throttling scans; that is a larger topic.

5.5.1 Snapshot Creation

This scan can be completely eliminated thanks to Theorem 1. As explained earlier, a snapmap block of the youngest snapshot S_1 is the same exact block as the activemap block until the activemap diverges from it; and that happens when it first gets dirtied after the CP in which S_1 was created. Section 5.2 describes how the scan's work is done on-demand if an activemap bit is cleared before the scan gets to it. If this work is done instead on the first modification of the activemap block after S_1 is created, then the scan is unnecessary. This requires a last-modify timestamp on each activemap block that can be compared with S_1's creation time. WAFL increments a CP count and stores it in the superblock, and also stores it as a last-modify timestamp in the header information of each metadata block. This CP count serves as the timestamp needed to eliminate the scan.

It should be noted that the on-demand create scan work cannot be paced or throttled. It is a function of when and how bits getting cleared in the activemap. Section 4 describes how the BFLog paces the clearing of bits in the activemap in a batched manner, which indirectly also paces the on-demand create scan work.

5.5.2 Snapshot Deletion

The performance trade-off between dbys and dbya modes of snapshot deletion is obvious. The dbys scan loads and processes blocks from three snapmaps and the

summary for a deletion. When multiple snapshots are deleted, their number and temporal pattern defines the number of neighbor snapmaps loaded by the scan. On the other hand, the dbya scan loads the snapmaps of all remaining snapshots. The more efficient mode can be chosen based on the number of snapmaps needed by the scan. If more snapshots are deleted while the dbys scan is in progress, it can be aborted and a dbya scan can be kicked off. This switch is safe because the scans are idempotent. It is not possible to switch from dbya to dbys mode for reasons explained below.

The dbys mode clearly needs the deleted snapshot to be preserved (in a hidden namespace) to the extent that its snapmap can be loaded and consulted until the scan is complete. In constrast, the dbya mode does not need the deleted snapshots. Usually, a block is considered free if it is marked free in the activemap and summary. However, this invariant is violated by the dbys mode. Suppose that S_1 is deleted and the dbys scan clears the i^{th} bit of the summary block. It is possible that the block b_i in the volume happens to be a snapmap block of S_1, or an ancestor of a snapmap block in S_1's tree of blocks. If the block allocator assumes b_i to be free and writes new content to it, then the dbys scan may be rendered infeasible, or worse, corrupt the summary map. Unless the file system has some convenient way to tell whether any given block of S_1 is needed for the dbys scan, it is easier to protect all of S_1's blocks until the scan is done. This leads to:

Theorem 3. *While a dbys scan is in progress to delete one or more snapshots, if the i^{th} bit of both the activemap and summary is 0, then block b_i is free iff the i^{th} bit in the snapmap of those snapshots is also 0.*

This also means that the computation for available space in the volume has to incorporate blocks that belong to deleting snapshots while the dbys scan is in progress. This additional complexity requires more persistent data structures to solve. Therefore, dbya is chosen sometimes even when the dbys mode looks attractive from the stand point of performance. One such situation is when the volume is running low on space. Another is when the 010 snapshot property is not true; for example, WAFL allows for a file (and its blocks) to be directly restored into S_0 from a not-so-recent snapshot. This is done by simply pointing the file directly to the blocks in the snapshot instead of copying them, which violates the 010 property. Such events are timestamped so that the system can deduce when the property is restored.

6 FlexVol Virtualization Layer

WAFL uses a virtualization technique that allows hosting hundreds or more volumes (*NetApp FlexVol*®) within the same collection of physical storage media called an *aggregate*. This has been key to providing several new capabilities such as FlexVol cloning, replication, thin provisioning, etc. Section 3 of [8] explains this layering: Each aggregate is a WAFL volume with blocks called *physical volume block numbers* (or *pvbns*) that map directly to blocks on persistent storage, and each FlexVol is a WAFL volume with blocks called *virtual volume block numbers* (or *vvbns*) that are actually just blocks in a file in the aggregate WAFL volume; the file is called the *container file* for the FlexVol. In other words, the vvbn of a block in a FlexVol is the same as its block offset in the corresponding container file. The interested reader is strongly encouraged to read [8] for diagrams and details.

Each layer maintains bitmaps to track its free blocks and each layer supports snapshots. In other words, a FlexVol uses its activemap, summary map, BFLog, etc., to track allocated and free vvbns, and an aggregate uses another set of this metadata to track pvbns. After a vvbn is completely freed in the FlexVol, it is necessary to effectively punch out the physical block at the appropriate L_1 offset of the container file before the corresponding pvbn can be marked free in the aggregate's activemap. It is crucial to do this soon after the vvbn gets freed, so that all FlexVols can share the available free space in the aggregate without restriction. However, this means that freeing a random vvbn requires loading and updating a random container file L_1; this operation is expensive because the L_1 is several times less dense than a bitmap. This motivated an important performance optimization in WAFL even before the structures in Section 4 were implemented. WAFL delays the free from the container in order to batch them. WAFL maintains a count of the number of *delayed-free* blocks in a container file per 32K consecutive vvbns, which is written to a per FlexVol file in the aggregate.

There are soft limits on the total number of delayed frees so that the file system can absorb large bursts of delayed frees getting created. Deletion of snapshots that exclusively own many blocks, a volume snaprestore (described in Section 5.1), or other events can generate such bursts of delayed-frees. When these limits are exceeded, the system tries to process that backlog with some urgency. Because the BFLog batches frees in the activemap, it can choose to do the "container-punch" work if there are sufficient delayed-frees in that range.

6.1 Evaluation

We first show the importance of the delayed-free optimization. A benchmark was built in house with the read/write mix that models the query and update operations of an OLTP/benchmark application. It was built

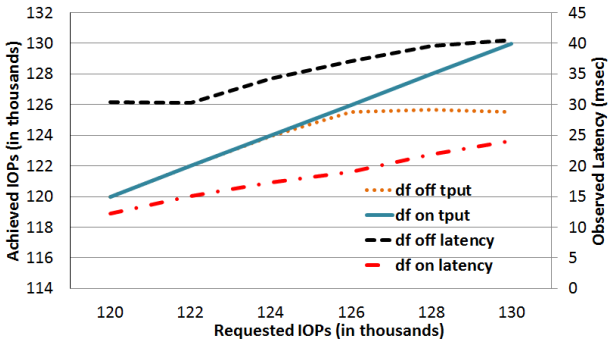

Figure 4: Benefit of delaying container frees: IOPs (left side y-axis) and latency (right side y-axis) versus input load with and without delayed frees (df) using an OLTP/database workload on a high-end all-SSD system with 20 cores and 128 GiB DRAM.

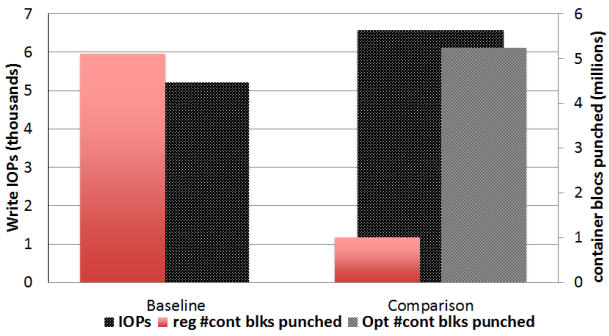

Figure 5: Benefit of the BFLog punching out the container blocks based on delayed-free counts on a low-end system.

to be very similar to the Storage Performance Council-1 (SPC-1) benchmark [5]. The benchmark was run on a high-end all-SSD system with 20 Intel Ivy Bridge cores and 128 GiB of DRAM, with and without the delayed-free optimization. Figure 4 shows the achieved IOPS (on the left y-axis) and latency (on the right y-axis). Only a small range of the input load is shown near the data points where the achieved load hits saturation and flattens out on the system without the optimization. The corresponding normalized latencies are 60% lower with delayed-frees. This shows that random container L_1 reads and updates hurt performance even on a high-end system with SSDs.

The primary source of benefit in the latencies we observed came from the greater than 60% reduction of metadata overhead with delayed-frees enabled. As a result, the CPs were able to give up more CPU cycles that could be used to service user requests at a higher rate.

Next, we study the aforementioned BFLog optimization. When the BFLog frees vvbns in the activemap, it consults the delayed-free count for that range. These delayed-frees could have accumulated from previously run BFLog or snap delete scan activity. If the total count is higher than an empirically derived threshold, the BFLog does the "container-punch" work, and appends the corresponding pvbns in the aggregate's BFLog active log. This is an incremental cost to the BFLog since it already had the FlexVol bitmaps loaded for freeing vvbns. Without the optimization, a separate background task searches for and processes these delayed-frees by loading all the FlexVol and container metadata, and that is more expensive and intrusive to client workloads.

To measure this, we ran test that randomly overwrote the data set with 32 KiB sized I/Os while FlexVol snapshots that exclusively owned a large number of blocks were deleted. This was done on a low-end system with 4 Intel Wolfdale cores and 20 GiB DRAM, which is more susceptible to this problem. Figure 5 shows the results. The client write load saw a 26% increase in throughput because, as the right-hand bar graphs show, the BFLog was able to perform the "container-punch" work much more efficiently; about 84% of it was done in the optimized mode.

7 Related Work

Originally FFS [16] started by using a linked list to maintain free blocks in the file system, which simplified the process of freeing blocks. The allocator picked up entries from this free list for allocation. However, once multiple files were created and removed, this unsorted free list grew in size, which led to random blocks being allocated by the allocator. This resulted in slower sequential reads. The designers then replaced the free list with a bitmap that identifies free blocks in a group, which resulted in better allocation but increased the cost of updating bitmaps for processing random frees. Similarly ext4 [3] also uses bitmaps for tracking free space. To make the allocation faster, it builds an in-memory buddy cache that maintains the status of clusters of 2^n blocks as free or allocated. Thus, ext4 also suffers from the problem of random frees like FFS.

The XFS [22] file system tracks free space by using two B+ trees. One B+ tree tracks space by block number and the second one by the size of the free space block. This scheme allows XFS to quickly find free space near a given block and therefore an extent of free blocks. When searching large regions of free space, a B+ tree performs better than a linear scan of a bitmap, and the second B+ tree can readily provide that information. Unfortunately, B+ trees are expensive to maintain in the face of random frees. Moreover, an insert or delete into a B+ tree can end up in a split or merge operation, which increases the cost

of a free operation. Btrfs [19] also maintains a B-tree in the form of an extent allocation tree to track allocated and free extents that serves as a persistent free space map for allocation. This looks similar to what XFS does for free space management and thus it also suffers when performing random sub-extent sized frees.

ZFS [2] tries to handle the problem of random frees by maintaining a log of allocations and frees per metaslab called a Space Map. As the random frees are appended to the log, it becomes extremely efficient because it is not necessary to load random portions of the bitmaps for updates. For allocation, the Space Map is loaded and the log entries are replayed to build an AVL tree of free space sorted by offset. ZFS compacts the persistent Space Map as needed by looking at the in-memory AVL tree. On a fairly aged file system with very little free space, the allocator has to load a lot of Space Map metadata to build this AVL tree. ZFS tries to amortize the cost of frees by logging them, but this forces the allocator to pay the cost of constructing the AVL tree.

There is clearly a trade-off that a file system can make in terms of when and how it processes a free that has been logged. Additionally, for the purposes of speeding up block allocation, file systems can choose to build complex data structures for tracking free space. However, that comes with the cost of maintaining those structures when blocks are freed, especially when they are random in their number space. Production-level file systems are usually built with specific goals in terms of workloads and features, and design principles are chosen to further those goals. They typically use a mix of persistent and in-memory free space tracking data structures that facilitate fast allocation assuming a certain buffer of free space, which lets the free space reclamation infrastructure play catch-up while the allocator looks for free blocks.

The WAFL block reclamation infrastructure stands apart because it maintains high and consistent performance while still supporting the various features of WAFL : snapshots, compression, deduplication (inline and background), FlexVol cloning, thin provisioning, file cloning, volume snaprestore, replication, single file restore, single file move on demand, etc. Unfortunately, the free space infrastructure interactions with many of these features are too lengthy to be presented here. WAFL has built-in data integrity mechanisms that protect memory-resident file system structures against scribbles and logic bugs [14]. The batching efficiency of the BFLog plays an important role in ensuring that these mechanisms work with negligible performance overhead.

The ability to absorb a high rate of reference count increments is critical to features like inline deduplication and rapid file cloning. One approach to batching refcount updates is an increment-decrement log; instead of just batching refcount decrements due to deletes, this log also collects increments created by new block sharing. This approach is especially useful in hierarchically reference counted file systems [19] since a single overwrite can generate hundreds or thousands of reference count updates [9]. This approach was explored in WAFL to support instantaneous cloning. In a hierarchically reference counted file system, where reference count updates are batched by an increment-decrement log, both the creation of and writes to cloned files can be performed efficiently [13]. By batching these updates, the decrements to block reference counts from writes can be canceled out by the increments of a clone create in the increment-decrement log without ever updating the persistent refcount file. However, there is also a need to query the log for pending increments or decrements to a particular block. Two implementations of B-trees that satisfy fast insertions and queries are presented in [1]. The queries can be satisfied in logarithmic time, and the insertions can be accomplished with guaranteed amortized updates to the B-trees.

8 Conclusion

The WAFL file system has evolved over more than two decades as technology trends have resulted in bigger and faster hardware, and larger file systems. It has also transformed from handling user directory style NFS/CIFS workloads to servicing a very wide range of SAN and NAS applications. It has been deployed to run in different configurations: purpose built platforms with all combinations of hard and solid state disks, software-defined solutions, and even NetApp Cloud Ontap®instances on AWS. The free space reclamation infrastructure has evolved to work well across all these permutations, providing a rich set of data management features and consistently high performance. Even though the pattern and rate of frees can be very random, we show that the proposed techniques allow the free space reclamation to keep up and behave deterministically without impacting the rest of the system. We describe elegant and efficient methods to track the space allocated to snapshots. Finally, we show how the infrastructure works across the extra layer of FlexVol virtualization.

9 Acknowledgement

The WAFL engineers who contributed to the designs presented in this paper are too many to list. We thank James Pitcairn-Hill, our reviewers, and our shepherd Marshall Kirk Mckusick for their invaluable feedback.

References

[1] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceeding of ACM Symposium on Parallel algorithms and architectures*, pages 81–92, 2007.

[2] Jeff Bonwick. Space maps. `https://blogs.oracle.com/bonwick/en/entry/space_maps`, 2007.

[3] Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Linux Symposium*, page 263, 2008.

[4] Standard Performance Evaluation Corporation. Spec sfs 2014. `https://www.spec.org/sfs2014/`.

[5] Storage Performance Council. Storage performance council-1 benchmark. `www.storageperformance.org/results/#spc1_overview`.

[6] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceeding of Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[7] Peter R. Denz, Matthew Curtis-Maury, and Vinay Devadas. Think global, act local: A buffer cache design for global ordering and parallel processing in the WAFL file system. In *Proceeding of 45th International Conference on Parallel Processing (ICPP)*, 2016.

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

[9] Travis R. Grusecki. Improving block sharing in the write anywhere file layout file system. `http://hdl.handle.net/1721.1/76818`, 2012.

[10] Christopher Hertel. *Implementing CIFS: The Common Internet File System*. Prentice Hall Professional Technical Reference, 2003.

[11] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter 1994 Technical Conference*, pages 235–246, Jan 1994.

[12] NetApp Inc. Data ONTAP 8. `http://www.netapp.com/us/products/platform-os/ontap/`, 2010.

[13] Ram Kesavan, Sriram Venketaraman, Mohit Gupta, and Subramaniam Periyagaram. Systems and methods for instantaneous cloning. Patent US8812450, 2014.

[14] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2017.

[15] Avantika Mathur, Mingming Cao, and Andreas Dilger. ext4: the next generation of the ext3 file system. *Usenix Association*, 2007.

[16] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, August 1984.

[17] Sun Microsystems. ZFS at OpenSolaris community. `http://opensolaris.org/os/community/zfs/`.

[18] Peter M. Ridge and David Deming. *The Book of SCSI*. No Starch Press, San Francisco, CA, USA, 1995.

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

[20] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10:1–15, 1992.

[21] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation or the sun network filesystem, 1985.

[22] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *USENIX Annual Technical Conference*, 1996.

[23] Ralph H. Thornburgh and Barry Schoenborn. *Storage Area Networks*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[24] David D. Wright. Data deletion in a distributed data storage system. Patent US8819208, 2014.