Efficient Search for Free Blocks in the WAFL File System

Ram Kesavan
NetApp, Inc
Sunnyvale, California
ram.kesavan@gmail.com

Matthew Curtis-Maury
NetApp, Inc
RTP, North Carolina
mcm@netapp.com

Mrinal K. Bhattacharjee
NetApp, Inc
Bangalore, India
mrinalb@netapp.com

ABSTRACT

The WAFL® write allocator is responsible for assigning blocks on persistent storage to data in a way that maximizes both write throughput to the storage media and subsequent read performance of data. The ability to quickly and efficiently guide the write allocator toward desirable regions of available free space is critical to achieving that goal. This ability is influenced by several factors, such as any underlying RAID geometry, media-specific attributes such as erase-block size of solid state drives or zone size of shingled magnetic hard drives, and free space fragmentation. This paper presents and evaluates the techniques used by the WAFL write allocator to efficiently find regions of free space.

CCS CONCEPTS

• **Hardware** → **External storage**; • **General and reference** → *Performance*;

KEYWORDS

Storage Management, File Systems Management, Operating Systems, Performance

ACM Reference Format:
Ram Kesavan, Matthew Curtis-Maury, and Mrinal K. Bhattacharjee. 2018. Efficient Search for Free Blocks in the WAFL File System. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3225058.3225072

1 INTRODUCTION

NetApp, Inc. is a data-management company that offers a proprietary operating system called ONTAP®, which implements the Write Anywhere File Layout (WAFL®) [15] file system. WAFL is a transaction-based file system that employs copy-on-write (COW) mechanisms to achieve fast write performance and efficient snapshot creation. A file system's *write allocator* is responsible for determining the location on persistent storage to which data and metadata are written in order to maximize write throughput and subsequent read performance. Because WAFL is a COW file system, each mutation is written to a new persistent location. Thus, it is crucial to quickly and efficiently find regions of free space that allow the write allocator to achieve its layout objectives for any given media type from within large block number spaces.

The ONTAP operating system nests two layers of the WAFL file system. In particular, multiple *virtualized* volumes called FlexVol® volumes are hosted on a shared pool of *physical* storage called an *aggregate* [11], with each layer managed by an instance of a WAFL file system. Therefore, when allocating persistent storage for data and metadata, the WAFL write allocator [10] must find free blocks in the block number spaces of both the aggregate and the FlexVol volume. That is, it must allocate both a physical block number and a virtual block number. The physical storage of an aggregate can be composed of various permutations of hard drives (HDDs), solid state drives (SSDs), and on-premises or remote object stores. To maximize performance, the search for free space needs to incorporate knowledge of any underlying device geometry (such as RAID [9, 24]) and any device-specific read and write performance characteristics. Data layout objectives within the virtualized block number space of a FlexVol are different, but they still require information about free space. For example, due to file system aging, free space fragmentation can interfere with the system's ability to achieve its data layout objectives [2, 28]. The problem of free space fragmentation is particularly acute in COW file systems such as WAFL because client overwrites of data free the previously consumed block. It has also been observed that providing large regions of free space is one of the key challenges in the design of a log-structured file system [26].

In this paper, we present the techniques used by WAFL for efficiently narrowing the search for free space in the aggregate and FlexVol block number spaces. We show how the solutions incorporate RAID geometry as well as an awareness of the underlying media, such as solid state drives (SSDs) or shingled magnetic recording (SMR) drives. We also present a novel data structure, *histogram-based partial sort* (HBPS), for efficiently tracking available free space in cases where RAID configuration is not a factor in data layout, such as within FlexVols and physical storage with built-in redundancy. Finally we evaluate these techniques, which have been running reliably on more than 250K customer systems, with different media types and workloads. Although our techniques are discussed in the context of the WAFL file system, the lessons learned apply to other COW file systems, such as LFS [26], ZFS [7], Btrfs [25], and more.

The layout of the paper is as follows. Section 2 gives background material on WAFL and write allocation. Section 3 presents the key algorithms and data structures for identifying free space in WAFL. We evaluate the performance of the approaches in section 4, present some related work in section 5, and conclude in section 6.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225072

2 BACKGROUND & MOTIVATION

Write allocation is the process by which the storage system chooses where on the persistent media to write data and metadata. File systems can be broadly categorized as either write-in-place or copy-on-write (COW). In the former category, write allocation happens a priori, and modifications to most of the file system data and metadata are written back to the same original location in persistent media. However, a COW file system, such as WAFL, allocates new blocks for all mutations. For example, the WAFL write allocator has to find and allocate at least 1GiB/s worth of free blocks to sustain a 1GiB/s client overwrite workload; this translates to finding 256k free blocks per second, because WAFL addresses its storage in 4KiB blocks.

2.1 WAFL Write Allocation

Clients connect to a storage server running ONTAP via various protocols, such as NFS, SMB, SCSI, NVMe, and send requests that perform file system operations, many of which modify the file system. WAFL collects the results of thousands of such modifying operations and efficiently flushes the changes to persistent storage. The delayed flushing of changes down to persistent storage allows better layout and amortizes the flushing overhead. This flushing of a consistent collection of changes as one single transaction is known as a *consistency point* (CP) [11, 15]. Previous work has provided details about the CP [17, 18] and its scalable write allocator [10].

ONTAP houses and exports multiple WAFL file system instances called FlexVol volumes from within a shared pool of physical storage called an aggregate, which is itself an instance of the WAFL file system [11]. An aggregate is thus a collection of physical storage devices and may comprise different permutations of storage media: HDD only, SSD only, combinations of HDDs and SSDs (called Flash Pool^{TM}), and combinations of SSDs and (third-party) object stores (called Fabric Pool^{TM}). Individual HDDs or SSDs cannot provide intrinsic redundancy, and so are configured into RAID groups to protect against device errors and failures[9, 12, 24]. A Flash Pool aggregate is composed of one or more RAID groups of SSDs together with several RAID groups of HDDs, and a Fabric Pool aggregate is composed of one or more RAID groups of SSDs and an on-premises or remote (AWS or Azure) object store. Such configurations store the "hot" (often-accessed) data and metadata in the faster media while using the slower media for the rest. ONTAP is also available in a software-only version, called ONTAP Select, that can run on prequalified hardware or as an ONTAP Cloud hyperscalar (AWS or Azure) instance.

A block in the WAFL file system is addressed by its *volume block number,* or *VBN.* WAFL uses a *physical VBN,* to refer to each block in the aggregate, which is mapped to a location on persistent media. Data contained in a FlexVol has both a physical VBN to specify the physical location of the block and a *virtual VBN* to specify the block's offset within the FlexVol. Write allocation of a block thus requires the assignment of both VBNs. FlexVol layering has been discussed in detail in earlier work [11, 18].

2.2 Importance of Free Space Contiguity

As a file system ages, its blocks are allocated and freed based on the I/O patterns of client and internal operations. As a result, the free space in the system can fragment; that is, free blocks get distributed in the block number space. For example, as data is written to an *unaged* file system, physical VBNs get consumed sequentially, but random overwrites of this data result in random frees and thus randomly located free space. The creation and deletion of files can eventually result in similar fragmentation of the free space.

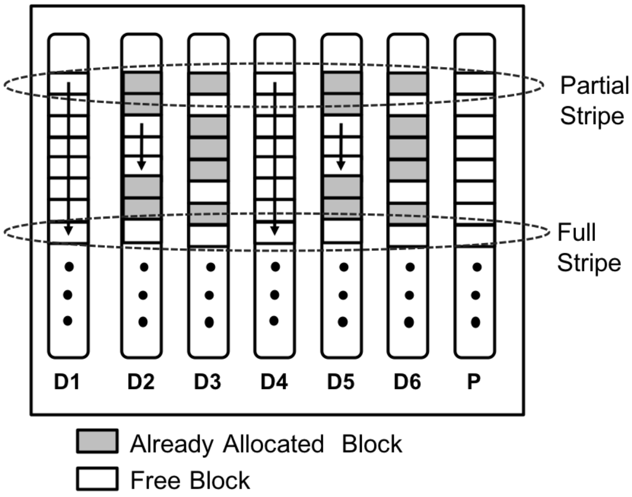


Figure 1: An example RAID 4 group with 6 data devices and 1 parity device showing fragmented free space and the potential impacts of free space fragmentation on write chain lengths and full stripe writes. Arrows indicate runs of contiguous free blocks on each device.

2.3 Full Stripe Writes

Figure 1 illustrates a RAID 4 group composed of six data devices and one parity device; typical real-world ONTAP deployments have many more data devices and multiple parity devices per RAID group [9, 12]. A *stripe* is a set of blocks, one per device, that share the same RAID parity block. A *full stripe write* is one in which all data blocks in the stripe are written out together such that RAID can compute the parity without additional reads. In contrast, a *partial stripe write* is inefficient because it requires RAID to read some of the blocks from the stripe to compute the parity [24]. Increased fragmentation of free space results in an increase in the number of partial stripe writes, as shown in Figure 1.

2.4 Long Write Chains

Efficient flushing of changes by the CP requires minimizing partial stripe writes as well as the total number of write I/Os to storage devices [14, 22]. In addition, WAFL aims to write logically sequential blocks of the file system to consecutive blocks of a storage device, which improves subsequent sequential read performance because the blocks can be read with a single I/O [2]. Contiguous free space on devices, such as on D1 and D4 in Figure 1, allows *long write chains*, which helps achieve both goals. On the contrary, writing to heavily fragmented regions of storage reduces opportunities for long write chains (D2 and D3 in Figure 1), and hurts both write and subsequent read performance.

2.5 Maximizing Colocation of Block Numbers

WAFL stores free space information in internal files called *bitmap metafiles* [17], which are flat and indexed by VBN; the i^{th} bit tracks the state of the i^{th} block of the file system. Although the assignment of virtual VBNs within a FlexVol has no direct impact on the physical layout of blocks, assigning free VBNs colocated in the number space minimizes the number of metafile blocks that need to be consulted and updated. Minimizing I/O to metafile blocks keeps the CP efficient and directly contributes to overall file system performance [18].

As noted earlier, ONTAP leverages other forms of storage that provide inherent resiliency and redundancy. Although ONTAP does not employ RAID in such cases, a similar analysis applies to VBN ranges even when the objective of full stripe writes is not relevant. That is, free space fragmentation can inflate CPU and I/O costs of updating metadata, as demonstrated in section 4. In order to maximize the performance of the file system, it is critical to counteract the effects of free space fragmentation by supplying the WAFL allocator with large regions of free space that are colocated in the block number space.

2.6 Device Write Properties

Different media types vary significantly in their write performance characteristics. Therefore, device-specific properties must be considered when identifying optimal regions of storage to target writes. In particular, SMR and SSD drives underperform when writes are ignorant of shingle zone size and erase-block size, respectively. As discussed in section 3.2, the WAFL write allocator must be conscious of such constraints.

The remainder of this paper describes the algorithms and data structures used by WAFL to track the best regions of storage when writing across various media types and storage configurations. This work is complementary to existing solutions for avoiding fragmentation or defragmentation of free space, as discussed in section 5. In short, finding contiguous free space is beneficial even in the presence of these other technologies.

3 TRACKING FREE SPACE

To maximize the efficiency of the write allocator, WAFL maintains free space information and uses it to guide the write allocator to “desirable” regions of storage. In particular, WAFL defines fixed-size regions of the block number space, called *allocation areas* (AAs), and tracks the availability of free space within each region.

3.1 Allocation Areas

The specific allocation area topology—i.e., which blocks are included in an AA—is defined based on properties of the underlying media and the RAID configuration, if any. On RAID group creation and growth, WAFL maintains the mapping of physical VBN ranges to storage devices based on their RAID topology. That is, WAFL is aware of the physical locations of all blocks and uses this information to define allocation areas that result in efficient free space tracking.

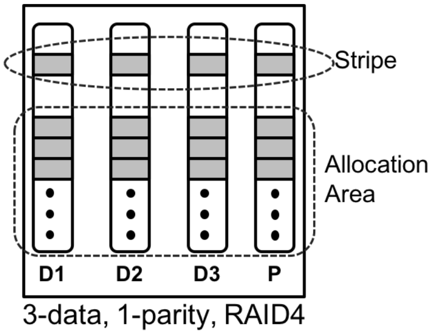


Figure 2: RAID group with 3 data devices and 1 parity device; real-world deployments have more data devices and multiple parity devices per RAID group.

For storage media arranged into a RAID group, an AA is a set of consecutive stripes, as shown in Figure 2. WAFL always targets writes to the emptiest AA in order to increase the probability of full stripe writes and long write chains to consecutive blocks on a device. In theory, the emptiest AA may or may not maximize this probability, because this metric does not account for the layout of available free space *within* the AA. However, we have found it to be empirically effective in practice. Furthermore, as discussed in section 3.3.1, the emptiness of AAs serves as a convenient metric for minimizing work done by defragmentation technologies such as segment cleaning.

On the other hand, for underlying storage with native resiliency and redundancy such as object stores, an AA is a set of consecutive blocks in the block number space. This simpler definition is sufficient because, in the absence of a RAID topology, write allocation does not need to pursue full stripe writes and must only attempt to write to consecutive blocks on such storage. This definition also applies to AAs within a FlexVol (for virtual VBNs); as explained earlier, FlexVols lack a RAID topology, and the primary goal is to minimize I/O to metafile blocks. In all cases, the write allocator picks an AA and then assigns all free VBNs from the AA in sequential order. Details of the actual assignment of VBNs to blocks in a CP, and how it is parallelized are provided in previous work [10].

3.2 Allocation Area Sizing

Several factors, including the media type of the device, influence the choice of an effective size for the AA. In general, smaller AAs provide a finer granularity of differentiation between regions of storage based on free space. On the other hand, larger AAs result in fewer overall AAs for a given amount of storage, which in turn reduces the overall memory and processing overhead for tracking all AAs.

3.2.1 Default AA Sizing. Historically, experiments showed that an AA size of 4k stripes (shown in Figure 3) works well for HDDs arranged in a RAID group; ONTAP originally used traditional SAS and SATA hard drives. An AA size of 32k consecutive VBNs works in the absence of a RAID geometry because it matches the alignment of bitmap metafiles [18] and maximizes updates within each block of these files. A 4KiB bitmap metafile block contains 32k bits—one

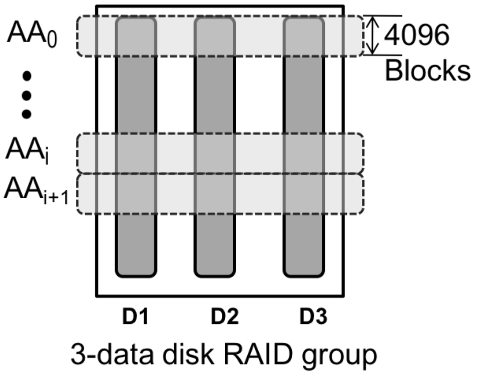


Figure 3: Allocation Area (AA) numbering within a RAID group with 3 data devices; the parity device is not shown here.

per VBN—and this alignment maximizes the number of blocks that can be allocated within an AA even while updating a single metafile block, thereby achieving the goal discussed in section 2.5.

3.2.2 SSD AA Sizing. Certain types of drives have a translation layer in their firmware that dynamically maps the exported block number space to the actual blocks of the physical media. SSDs have a flash translation layer (FTL) that generates empty erase blocks for new writes and evenly wears out the erase blocks by moving data around within the SSD [23]. Thus, if a host writes N random blocks to an SSD, the FTL of the SSD typically writes more than N blocks. The ratio of blocks actually written by the SSD to blocks written by a host is called *write amplification*; in theory, a value of 1 is the best achievable write amplification. SSDs come with a program/erase-cycles rating that indicates their endurance, so minimizing write amplification is critical to maximizing device lifetime. Figure 4 (A) illustrates that the use of smaller AAs (such as the default HDD AA size) results in the writing of a partial erase block. In such cases, the FTL must first relocate all active data in the erase block elsewhere on the drive and then erase the entire block before writing new data there [4]. We therefore choose an AA size for SSD RAID groups that is several erase blocks; this choice works across the various SSDs supplied by our vendors. Figure 4 (B) illustrates this with an example AA that is larger than 2 erase blocks. Because the WAFL write allocator picks the emptiest AA and then writes to all free physical VBNs in it, this AA sizing reduces the number of blocks relocated by the FTL in each erase block when processing these writes and therefore reduces write amplification. Perfect alignment of the AA to the erase block boundary is ideal but not necessary. The FTL may need to relocate a few in-use blocks at each edge of an unaligned AA.

SSD vendors hide some fraction of the drive capacity for smooth operation of the FTL; this is called *overprovisioning (OP)* [5]. Enterprise-quality workloads are characterized by high IO operations per second, and the FTL in SSDs productized for such workloads can hide up to 30% of the drive capacity. SSD AA sizing has been crucial to reducing write amplification, and has enabled NetApp to ship SSDs in ONTAP with significantly lower OP, thereby increasing available capacity for the file system.

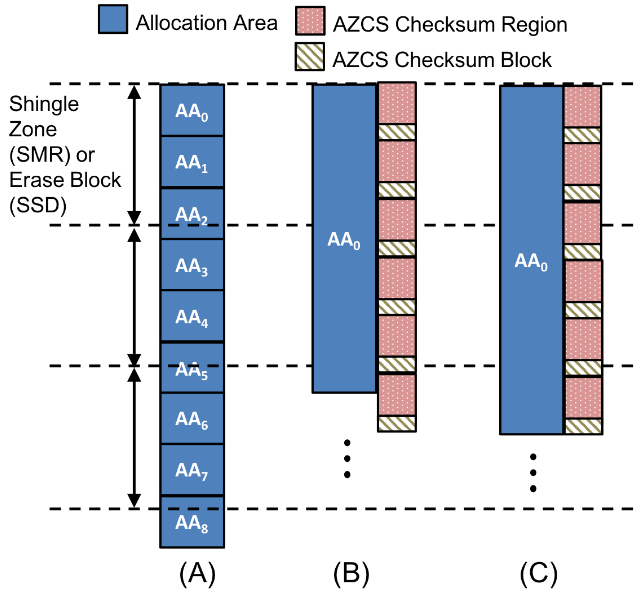


Figure 4: Example alignments of allocation areas to underlying shingle zones on SMR drives or erase blocks on SSD drives. (A) Historical AA sizing applied to SMR or SSD. (B) Custom AA size for SMR or SSD that is larger than SMR shingle zone or SSD erase block. (C) Custom AA size for AZCS that is aligned to checksum region. The size of a shingle zone is unrelated to and different from the size of an erase block.

3.2.3 SMR AA Sizing. Shingled magnetic recording (SMR) drives overlap neighboring tracks to increase areal data density. Therefore, writing to a track overwrites subsequent tracks in that shingled zone [1, 14], which can corrupt the data in those subsequent tracks. Drive-managed SMR drives avoid this problem by either: (1) Reading subsequent tracks and writing out the new and old data together (in-place update); or (2) Writing the data to a new location in an empty track and freeing the old physical block (out-of-place update), which carries overhead of maintaining the new logical-to-physical mapping as well as garbage collection [14]. If the AA size is smaller than the SMR zone, as shown in Figure 4 (A), then writes to the last track of each AA (in the middle of an SMR zone) will require drive intervention to avoid corrupting data at the beginning of the next AA. As shown in Figure 4 (B), WAFL selects an AA size for SMR RAID groups that is much larger than the shingle zone size, to reduce the frequency of drive intervention.

3.2.4 Checksums and AA Sizing. Another factor affecting AA size for SMR drives is ONTAP's use of advanced zone checksums (AZCS). WAFL persists with each block a 64-byte identifier to protect against media errors and lost and misdirected writes [6]; among other things, the identifier includes a checksum. If the storage device supports 520-byte sectors, the 64 bytes of space left over after storing a 4096-byte block to 8 sectors stores the checksum for the block. However, when a device's sector size aligns exactly to 4KiB, checksums must be stored in a separate block. With AZCS encoding, 63 consecutive blocks use the 64th as a checksum block, which stores each of their 64-byte identifiers, as illustrated in Figure 4; a pink region is an AZCS region of 63 data blocks. If ACZS regions cross AA boundaries, as shown in Figure 4 (B), then writes near the end of an AA require a nonsequential write to update the corresponding checksum block. This can be very harmful to performance on SMR drives. Therefore, if the media in an SMR-based RAID group uses AZCS, WAFL can choose to align AA size to a multiple of the AZCS region size, as shown in Figure 4 (C). This alignment ensures that all blocks sharing a checksum block are written sequentially along with their shared checksum block, because AAs are written fully from beginning to end.

3.3 Allocation Area Caches

In order to efficiently provide high-quality AAs to the write allocator, WAFL utilizes an allocation area cache (*AA cache*). This data structure leverages natural variability in AA scores to provide access to the emptiest AAs, even while the file system ages. As explained earlier, this process differs based on whether the storage media provides native redundancy or ONTAP arranges them into RAID groups. Therefore, we separately optimize this data structure for the cases of *RAID-aware* layout and *RAID-agnostic* layout.

One AA cache is built for each physical VBN range that maps to a RAID group, one for each physical VBN range that maps to storage that does not require RAID, and one for the virtual VBN range of each FlexVol volume. The free space of an AA is quantified by its *AA score*: it is the number of free blocks in the AA, computed by consulting bitmap metafiles. The AA score decreases when the write allocator allocates VBNs from that AA, and it increases when VBNs from that AA are freed. AA score updates resulting from frees (increments) and allocations (decrements) are delayed and performed efficiently in batched fashion at the CP boundary. Data structures used to optimize the batching of random VBN frees are detailed in [18].

3.3.1 RAID-Aware Allocation Area Cache. This is an in-memory max-heap of all AAs in a RAID group sorted by score. The max-heap is rebalanced at the end of each CP after updating the scores of AAs in which VBNs were allocated or freed. Section 4.1 shows that optimizing the selection of the AA in the physical RAID group storage has a large impact on write performance, which justifies the memory used for storing *all* AAs in this data structure rather than limiting it to tracking only some subset of AAs. The memory requirement for a RAID-aware AA cache increases linearly with the storage capacity of each device, but remains unaffected by the number of devices in the RAID group. As an example, the AA cache for a RAID group composed of 16TiB devices requires around 1MiB of memory to track 1 million default-sized AAs; $\frac{16TiB}{4KiB} = 1G$ physical VBNs and $\frac{1G}{4k} = 1M$ AAs of 4k VBNs each.

WAFL attempts to write to all RAID groups available in an aggregate in order to maximize the total write throughput, so it selects the best AA from each RAID group by consulting its max-heap; the assignment of free blocks from those selected AAs is detailed in [10]. Performance can sometimes be improved by not writing to a RAID group with excessive free space fragmentation, because writing partial stripes to such a group is more expensive than writing (fewer) fuller stripes to busier RAID groups. The write allocator can use the score of the best AA of the RAID group as an indicator of its fragmentation and so judge when to stop and when to resume writing to that RAID group. For example, if the best AA score in a RAID group is below some threshold, then the allocator knows that the overhead of writing to these drives will outweigh any benefit otherwise provided by leveraging the additional I/O bandwidth of these drives. Ideally, the AA score could incorporate additional fragmentation information, such as the desirability of the layout of free space within the AA, but using scores based only on free space suffices in practice.

WAFL improves AA scores through a process similar to segment cleaning [26], in which the content of all in-use blocks in an entire allocation area is relocated elsewhere on storage in order to generate completely empty AAs. Each AA near the top of the max-heap goes through this cleaning process once, thereby ensuring a small pool of cleaned AAs. Cleaning AAs with the best scores implies the relocation of the fewest in-use blocks [22, 26], so just-in-time cleaning of AAs provided by the AA cache yields the best return on investment. The techniques used for free space defragmentation in WAFL is a larger topic, and will be detailed in a future publication.

3.3.2 RAID-Agnostic Allocation Area Cache. In the absence of RAID layout considerations, we have found that the selection of the single best AA is not worth the memory overhead associated with the aforementioned max-heap approach to AA selection. This calls for an efficient data structure with a small memory footprint that can provide access to near-optimal AAs. The ONTAP *thin provisioning* feature [3] enables a single aggregate to house a collection of FlexVol volumes whose total sizes exceed the physical storage. Such a system can have hundreds of FlexVols, each several hundred terabytes in size. A 128 TiB FlexVol volume has a million AAs; as described earlier, each AA has 32k virtual VBNs. The tracking of all AAs of the FlexVol in its AA cache requires too much memory, which is not justified because block allocation within a volume is disconnected from physical layout, and is only aimed at efficiency in updating bitmap metafiles, as discussed in section 2.5. This is also true for writing to an object store that provides native redundancy. That is, we needed a data structure that efficiently provided AAs with close-to-best scores, but used a finite amount of memory even when tracking millions of AAs.

We developed a novel *histogram-based partial sort (HBPS)* data structure to track the AA scores in a FlexVol volume or underlying non-RAID storage. The HBPS data structure uses at least two 4KiB pages. As shown in Figure 5, the first page is a histogram that counts the number of AAs in different bins of score ranges. A list of *all* the AAs from the best bins is stored in the additional pages, although AAs that fall within a bin remain unsorted.

For RAID-agnostic AAs, a best score is 32K—the number of free blocks in an empty AA—and the worst score is 0, which indicates a full AA. The AA score space is divided into bins covering score ranges of 1K. The histogram page tracks the number of AAs within each bin as well as a pointer to the first element in the list component of the HBPS that falls within the corresponding score range. The first bin tracks the number of AAs with scores that fall in

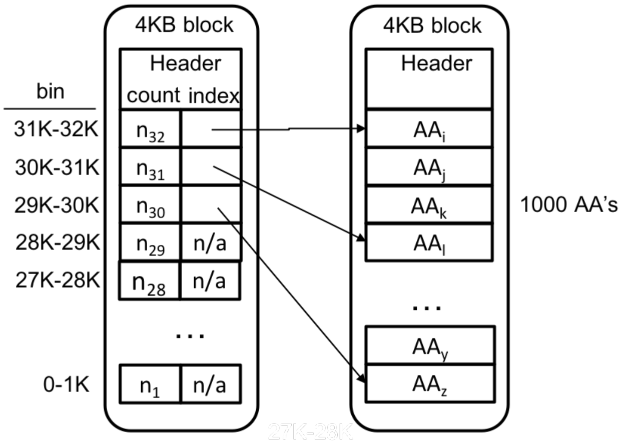


Figure 5: The histogram-based partial sort data structure used for RAID-agnostic AA caches.

the range 31K-32K, the second bin tracks the number of AAs with scores that fall in the range 30K-31K, and so on. In Figure 5, the histogram entry for the 31K-32K bin (n_{32}) is 3 and its index field points to the first AA from that bin in the list (AA_i), followed by AA_j and AA_k. The next bin shows n_{31} AAs in the 30K-31K range and points to AA_l to AA_y, and so on.

The number of additional pages beyond the histogram page is based on the number of entries needed for that HBPS use case. For use as a RAID-agnostic AA cache, one page of entries is found to be sufficient, so the total memory overhead is limited to two pages (one for the histogram and one for the list), regardless of the size of the FlexVol volume or storage target. This second page stores 1,000 AAs that fall into the top score ranges. Thus, AAs from only the best score ranges are present in the list and have valid indices stored in the histogram. The remaining histogram entries still track accurate counts, even if their AAs do not qualify for the list.

The benefit provided by sorting AAs within a range was found to be negligible, hence the "partial sort" in the name of the data structure. The write allocator always picks the first AA in the second page because that will have a score from the highest possible range. The AA cache is always guaranteed to provide an AA with the highest score within a 3.125% error margin (as determined by the ratio of the size of each range to the maximum score; i.e., $\frac{1k}{32k}$), by picking an AA from the highest populated range in the cache.

If an AA's score moves into a different bin due to its VBNs getting allocated or freed, it is a constant time operation to decrement the previous bin's total and increment the next bin's total. If an AA that was previously absent from the list moves into one of the top intervals due to VBNs in it getting freed, then it is inserted into the list and the index in the histogram is changed appropriately. Because the AAs in the list within a bin are unsorted, only one AA needs to be moved down from each bin present in the list to the next lower bin in that list. As mentioned earlier, updates to the HPBS get efficiently batched at the CP boundary. In the rare case that the write allocator consumes more AAs than are being inserted due to freeing of blocks, a background scan replenishes the list by walking bitmap metafiles. Thus, this AA cache uses exactly two pages of memory to efficiently provide close-to-optimal AAs with very little overhead.

The HBPS data structure has other uses in WAFL when millions of items need to be sorted in close-to-optimal order and with minimal memory usage. For example, it is used to track *delayed-free scores* [18].

3.4 The TopAA Metafile

ONTAP operates in an active-active high-availability (HA) pair configuration. When either node fails, its partner node mounts the failed node's FlexVols and aggregates, replays all client operations logged since the last completed CP to guarantee no data loss [15], and then starts servicing operations from clients. Thus, client access is delayed after a crash until the mounting is completed. When an aggregate or FlexVol volume is mounted, write allocation cannot begin until an AA is selected, which in turn requires that AA caches be operational.

Rebuilding AA caches requires a linear walk of the bitmap metafiles in order to compute the scores of each AA and to insert them into the respective AA caches; this may take multiple seconds. Instead, each WAFL file system instance stores the AA cache structure in a *TopAA* metafile. The RAID-aware TopAA metafile uses one 4KiB block for each RAID-aware AA cache that it fills with the 512 best AAs and their scores. Although this represents only a small subset of all AAs, it is enough to seed the max-heap with high-quality AAs until background work can rebuild the entire cache. Client operations and CPs can be sustained for dozens of seconds using the seeded AAs while the max-heap is fully populated in the background. Each RAID-agnostic AAcache is persisted as a two-block TopAA metafile. In fact, the HBPS structure is directly embedded into the pages corresponding to the two blocks of the metafile that are kept pinned in the WAFL buffer cache. Therefore, very little I/O and CPU is necessary to get the AA cache structure ready after an aggregate or FlexVol volume is mounted.

Prior work describes the techniques used in WAFL to prevent metadata such as TopAA metafiles from being corrupted by memory scribbles or by logic bugs in the file system software [19]. In rare cases, if the metafiles blocks are damaged in the physical media and RAID is unable to reconstruct them (say multiple devices in the RAID group have failed), the online WAFL repair tool—WAFL Iron—is used to recompute and recover them; WAFL Iron is described in a previous publication [16].

4 PERFORMANCE ANALYSIS

In this section, we evaluate the solutions presented earlier. We study the performance advantage of using both forms of AA caches with real-world workloads on aged data sets, the change in performance when the the AA size is tuned for different media types, as well the advantages of maintaining the TopAA metafiles.

4.1 AA Cache Performance

First, we evaluate the effects of using AA caches to provide the write allocator with high-quality AAs on a midrange all-SSD NetApp server with 20 Intel Ivy Bridge cores and 128GB of DRAM. A set of LUNs was configured on the NetApp server, and a number of

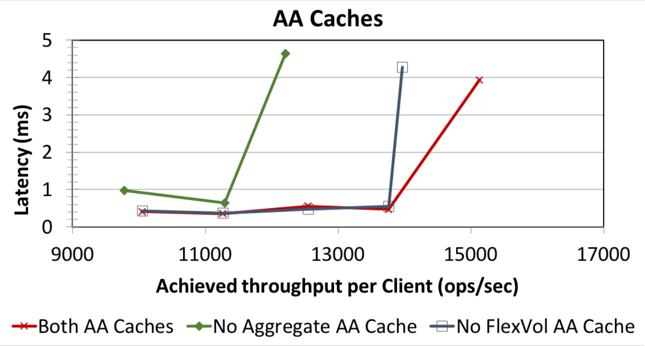


Figure 6: Latency versus achieved throughput (per client) using AA caches for both the FlexVol and aggregate VBNs (both AA caches), for the FlexVol only (FlexVol AA cache), and for the aggregate only (Aggregate AA cache). In the latter two cases, the other AA cache is disabled. Lower and to the right is better.

clients were set up to send 8KiB random overwrites to these LUNs over a Fibre Channel network. Random overwrites create worst-case fragmentation in a COW file system, because each overwrite frees the previously used block. The resulting freeing of random blocks result in significant fragmentation of the available free space. Before measurement, the aggregate was filled up to 55% and was thoroughly fragmented by applying heavy random write traffic for a long period of time. We disabled WAFL free space defragmentation in order to isolate the benefits of the AA cache; this is the standard configuration for many ONTAP platforms.

4.1.1 RAID-Aware AA Cache. Figure 6 presents experimental data comparing latency versus achieved throughput at increasing levels of load. We begin by comparing performance with and without a RAID-aware AA cache for the physical VBNs in the aggregate. Although not shown in the figure, we also traced the scores of the AAs picked by the write allocator. With the AA cache enabled, the selected AAs average 61% free space (despite the aggregate having only 45% free space), whereas with the AA cache disabled, randomly selected AAs average only 46% free space. This demonstrates that the resulting free space fragmentation is not uniform and that the AA cache leverages this nonuniformity to locate more desirable regions of storage. Although not enabled in this experiment, the freeing of blocks due to other internal activity, such as snapshot deletion, further adds to this nonuniformity. Critically, the use of emptier AAs translates to 24% better throughput with an 18% lower latency under peak load to the server. This further confirms our assertion that writing to emptier regions does, in fact, improve performance. If we focus on performance at a lower load to the server, we see that the latency achieved at a per-client load of 12k ops/sec is dramatically lower with the AA cache enabled (0.56ms versus 4.6ms) because the storage server is able to absorb heavier load before latencies increase. Beyond these performance concerns, we measured that directing writes to emptier AAs on SSDs translated to reduced block remapping by their FTLs, which reduced write amplification from 1.77 to 1.46. This reduction in the number of writes is known to extend the life of SSDs.

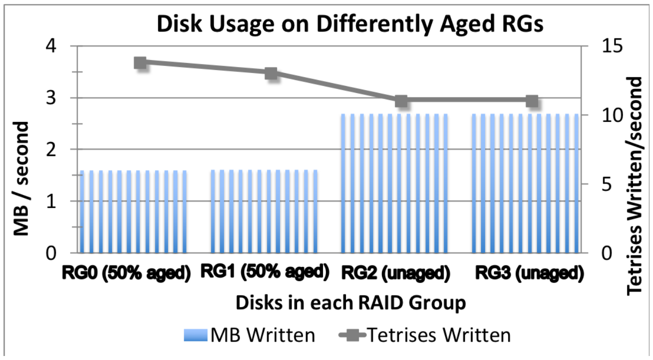


Figure 7: Disk usage across differently aged disks and HDD RAID groups for an OLTP benchmark test.

4.1.2 RAID-Agnostic AA Cache. We next compared performance with and without a RAID-agnostic AA cache for managing virtual VBNs within FlexVols. Similar to the RAID-aware AA cache evaluation, our traces showed significantly better scores for the selected AAs. In particular, the average free space available in the chosen AAs is 78% compared to 61% for randomly selected AAs. As discussed in section 2.5, picking AAs with more free blocks reduces the number of metafile blocks that must be updated. This amortization manifests as a 5.7% reduction in the average computational overhead per operation (309usec/op versus 293usec/op); this overhead is defined as the total CPU cycles used by the WAFL file system code path per client operation. Thus, the FlexVol AA cache improves throughput by 8.0% and reduces latency by 8.6% at peak load. Despite the independence of virtual VBN allocation from physical layout, it is critical to focus writes to comparatively emptier regions of the virtual block number space.

Code-path profiles show that under heavy I/O load, only about 0.002% of the total CPU cycles was spent maintaining each of the RAID-aware and RAID-agnostic AA caches. The negligible CPU overhead comes with a large benefit to performance. This result emphasizes the efficiency of the histogram-based partial sort (HBPS) data structure.

4.2 Evaluation of Imbalanced Aging

Customers increase the storage capacity of an aggregate over time by adding discrete RAID groups. This typically results in different fragmentation across RAID groups within an aggregate; older RAID groups tend to be more fragmented. However, the WAFL write allocator attempts to maximize the write throughput by writing to all the data disks that comprise an aggregate. In the presence of fragmentation imbalance, it is preferable to direct a larger percentage of writes to the less aged media, which can operate with higher efficiency and throughput. To evaluate this scenario, we ran an internal OLTP benchmark on the aforementioned midrange system but with all-HDD storage and differently aged RAID groups. This workload is characterized by predominantly random read and write I/O operations (that model query and update operations typical to a database).

Figure 7 plots the number of blocks and tetrises written per second to each disk and RAID group (RG), respectively, at a cumulative

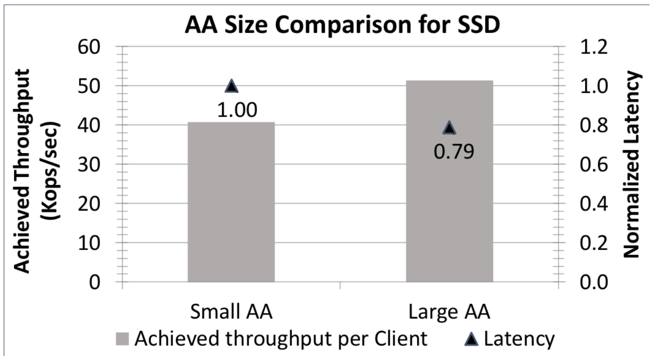


Figure 8: Latency (normalized to baseline) versus achieved throughput (per client) with a small AA cache size optimized for HDD (4K stripes) and a larger AA size designed for SSD erase block size.

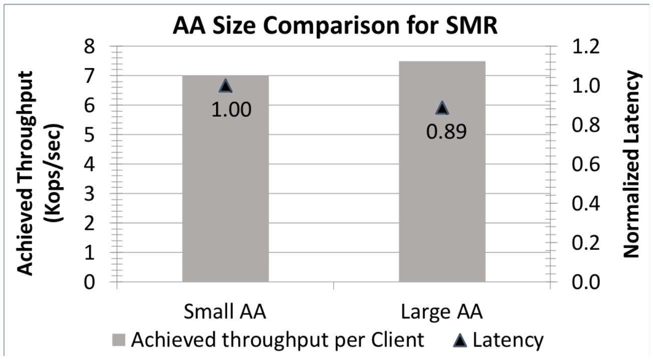


Figure 9: Latency (normalized to baseline) versus achieved throughput (per client) with a small AA cache size optimized for HDD (4K stripes) and a larger AA size designed for SMR zone size and AZCS region size.

client load of 68K ops/sec. A tetris is the unit of write I/O sent from WAFL to a RAID group, and is composed of 64 consecutive stripes. In general, tetrises written to fragmented regions of physical storage are inefficient because they contain partial stripes. Before the experiment, disks in RG0 and RG1 were aged by overwriting and freeing its blocks several times until a random 50% of its blocks were used. RG2 and RG3 have not been fully written to, and therefore contain many empty AAs. The figure shows two key results. First, that blocks are evenly distributed across all disks with the same fragmentation level. Second, that more blocks are written to the newer and emptier RAID groups, which is desirable. It is well known that the write throughput achievable to a disk with fragmented free space is less than that to a disk with contiguous free space. The write allocator is biased to assign fewer free physical VBNs from fragmented RAID groups (RG0 and RG1) in each CP, which results in a marginally higher rate of tetrises written to those RAID groups. Therefore, not only does the use of an AA cache help identify higher-quality disk regions for writing, it also helps balance out writes proportionately based on RAID group aging.

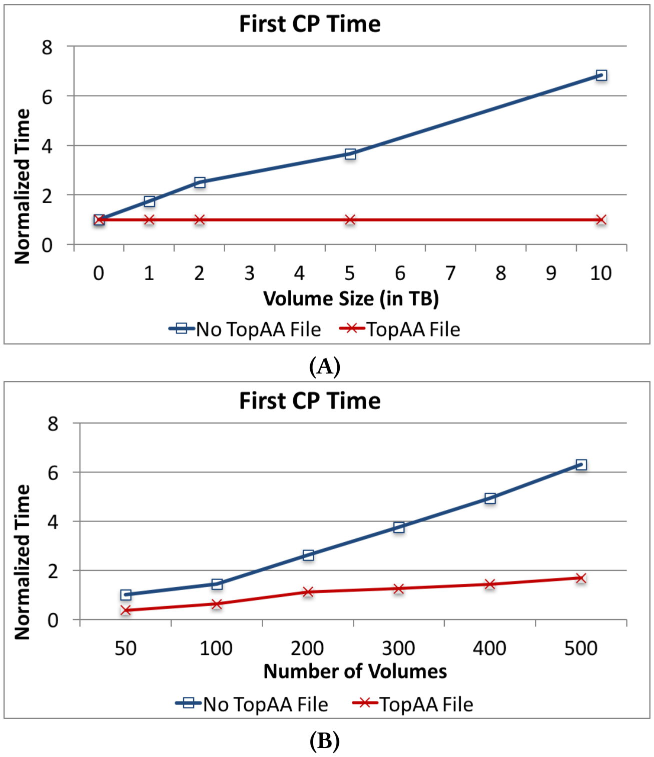
4.3 AA Sizing on SSD and SMR Drives

Figure 8 shows the latency versus achieved throughput on a 16-core NetApp server with all-SSD storage, aged to 85% fullness using 4KiB random reads and writes to several configured LUNs from multiple clients connected via a Fibre Channel network. Setting AA size to a multiple of SSD erase block size ("Large AA"), combined with leveraging the AA cache to find the emptiest AAs, allows WAFL to direct writes to the emptiest erase blocks on the drive in order to minimize the block remapping overhead in the FTL, as discussed in section 3.2. This approach allows the system to deliver 26% higher throughput with 21% lower latency under peak load compared to an AA size optimized for HDD. In this experiment, the use of large AAs halved the write amplification when compared with HDD-optimized AAs, thereby demonstrating the importance of optimizing AA size for the underlying storage media for both performance and device lifetime.

SMR drives are not officially supported on ONTAP yet, and WAFL has not been fully tuned for these drives, so we cannot publish a full evaluation at this time. As a single data point, we issued sequential writes to an *unaged* file system on a 12-core NetApp server with sample drive-managed Seagate SMR drives and an AA size larger than SMR zones and aligned to AZCS regions. Consideration of shingle zone size reduces costly writes to the middle of a new shingle zone when switching AAs, but this is less important for unaged systems because WAFL already uses AAs sequentially in such cases. AZCS zone alignment remains critical to avoiding random disk writes to update checksum blocks when switching AAs and thereby increasing the sequentiality of disk accesses, as discussed in section 3.2; this functionality has been implemented only for internal builds specifically for SMR drives. This experiment showed a 7% increase in drive throughput and an 11% reduction in latency due to avoiding random checksum block writes, as shown in Figure 9. Based on the improvements seen from AA selection in the SSD data just reported, we can reasonably expect *aged* SMR drives to see performance benefits, given the similarities between SMR and SSD in terms of the block remapping overhead with fragmented free space.

4.4 Persistent TopAA Improvements

As discussed in section 3.4, the best AAs in each aggregate and FlexVol volume are persistently saved as TopAA metafiles; Therefore, very little work is required to build the AA caches when file systems are mounted, which is very important because it impacts the time taken to restore access to clients after an outage event like a failover or a reboot by gating the completion of the first CP. Figure 10 (A) shows the time required to complete the first CP when mounting a 10TB aggregate with 50 FlexVol volumes of varying sizes on a midrange system, with and without TopAA metafiles. We can see that the time needed to read 51 fixed-size TopAA metafiles (1 for the aggregate) and to build their corresponding AA caches is independent of the file-system size, whereas that time increases linearly with file-system size in the absence of TopAA metafiles. Figure 10 (B) shows the much improved performance using TopAA metafiles with an increasing number of 100GB FlexVol volumes. These results demonstrate the scalability—in terms of both FlexVol volume count and size—of using the TopAA metafile.



(A)

(B)

Figure 10: Normalized time taken for the first CP after boot with and without the TopAA metafiles measured with an increase (A) in the FlexVol volume size, and (B) in the number of FlexVol volumes.

5 RELATED WORK

Much work has been done to evaluate the effect of file system aging on system performance and to demonstrate the importance of testing on aged file systems [8, 13, 28]. Some previous work has sought to avoid fragmentation [8, 29]; however, in a COW file system some degree of fragmentation is inevitable [13]. Other work has considered free space defragmentation [14, 20, 26, 27]. Our solutions for finding desirable regions even in the presence of fragmentation are orthogonal and complementary; they also bring an awareness of underlying device write performance characteristics. In particular, our approach can be used to direct defragmentation to clean the emptiest regions, thereby reducing the associated overhead of defragmentation [22, 26]. The techniques used for free space defragmentation in WAFL will be detailed in a future publication.

Most related to our own work are efforts to intelligently select regions of storage for writing. For example, FFS [21] defines disk regions called “cylinder groups” and targets writes to areas with the most free inodes. XFS [29] rotates file assignment across “allocation groups,” not based on emptiness, but to increase parallelism of free space management. ZFS [7] manages “metaslabs” similarly and targets writes based on a weighted score that includes expected seek distances. We expect that systems like these would benefit from ranking and sizing their respective allocation groups and cylinder groups to make better and faster block allocation decisions. He and Du developed a system for SMR drives called SMaRT [14] that directs writes to the emptiest regions of the disk to minimize the overhead of track management.

6 CONCLUSION

In this paper, we have presented data structures and algorithms that allow WAFL to quickly and efficiently target writes to the emptiest regions of storage to counteract the effects of aging. Further, our work influences free space selection based on knowledge of the underlying RAID geometry, if any, and various media-specific attributes to optimize performance and even improve SSD device lifetime. Finally, we showed that by persisting our data structures on storage media in an efficient way, we are able to supply the write allocator with free space more quickly after a reboot or failure. Although the work is presented in the context of the WAFL file system, the lessons learned apply generally to the file system community, especially in the context of COW file systems.

REFERENCES

[1] Abutalib Aghayev, Y. Theodore, Garth Gibson, and Peter Desnoyers. 2017. Evolving Ext4 for Shingled Disks. In *Proceedings of Conference on File and Storage Technologies.*
[2] Woo Hyun Ahn, Kyungbaek Kim, Yongjin Choi, and Daeyeon Park. 2002. DFS: A de-fragmented file system. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS).*
[3] Carlos Alvarez. 2011. NetApp Thin Provisioning Deployment and Implementation Guide. www.netapp.com/us/media/tr-3965.pdf. (2011).
[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2015. *Operating Systems: Three Easy Pieces* (0.91 ed.). Arpaci-Dusseau Books.
[5] Seagate Tech Articles. 2018. SSD Over-Provisioning And Its Benefits. https://www.seagate.com/tech-insights/ssd-over-provisioning-benefits-master-ti/. (2018).
[6] Wendy Bartlett and Lisa Spainhower. 2004. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 87–96.
[7] Jeff Bonwick and Bill Moore. 2007. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf. (2007).
[8] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, et al. 2017. File Systems Fated for Senescence? Nonsense, Says Science!. In *Proceedings of Conference on File and Storage Technologies.*
[9] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac Steven Kleiman, James Leong, and Sunitha Sankar. 2004. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of Conference on File and Storage Technologies.*
[10] Matthew Curtis-Maury, Ram Kesavan, and Mrinal K. Bhattacharjee. 2017. Scalable Write Allocation in the WAFL File System. In *Proceedings of the Internal Conference on Parallel Processing (ICPP).*
[11] 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. 2008. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the USENIX Annual Technical Conference.*
[12] Atul Goel and Peter Corbett. 2012. RAID Triple Parity. In *ACM SIGOPS Operating Systems Review*, Vol. 46. 41–49.
[13] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *Proceedings of the European Conference on Computer Systems (EuroSys).*
[14] Weiping He and David H.C. Du. 2017. SMaRT: An Approach to Shingled Magnetic Recording Translation. In *Proceedings of Conference on File and Storage Technologies (FAST).*
[15] Dave Hitz, James Lau, and Michael Malcolm. 1994. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter Technical Conference.*
[16] Ram Kesavan, Harendra Kumar, and Sushrut Bhowmik. 2018. WAFL Iron: Repairing Live Enterprise File Systems. In *16th Usenix Conference on File and Storage Technologies (FAST).*
[17] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. 2017. Algorithms and Data Structures for Efficient Free Space Reclamation in WAFL. In *Proceedings of Conference on File and Storage Technologies (FAST).*
[18] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. 2017. Efficient Free Space Reclamation in WAFL. *ACM Transactions of Storage* 13 (October 2017).

[19] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. 2017. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *15th Usenix Conference on File and Storage Technologies (FAST)*.

[20] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage.. In *Proceedings of Conference on File and Storage Technologies (FAST)*.

[21] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. 1984. A fast file system for UNIX. *Transactions on Computer Systems* 2, 3 (1984), 181–197. https://doi.org/10.1145/989.990

[22] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of Conference on File and Storage Technologies (FAST)*.

[23] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. In *IEEE Transactions on Parallel and Distributed Systems*.

[24] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[25] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage* 9, 3, Article 9 (Aug. 2013), 32 pages. https://doi.org/10.1145/2501620.2501623

[26] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems* 10 (1992), 1–15.

[27] Takashi Sato. 2007. ext4 online defragmentation. In *Proceedings of the Linux Symposium*, Vol. 2. 179–86.

[28] Keith A. Smith and Margo I. Seltzer. 1997. File system aging–increasing the relevance of file system benchmarks. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 25. ACM.

[29] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.