Storage Gardening: Using a Virtualization Layer for Efficient Defragmentation in the WAFL File System

Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra, *NetApp*

https://www.usenix.org/conference/fast19/presentation/kesavan

This paper is included in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

Storage Gardening: Using a Virtualization Layer for Efficient Defragmentation in the WAFL File System

Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra

NetApp, Inc

As a file system ages, it can experience multiple forms of fragmentation. Fragmentation of the free space in the file system can lower write performance and subsequent read performance. Client operations as well as internal operations, such as deduplication, can fragment the layout of an individual file, which also impacts file read performance. File systems that allow sub-block granular addressing can gather intra-block fragmentation, which leads to wasted free space. This paper describes how the NetApp® WAFL® file system leverages a storage virtualization layer for defragmentation techniques that physically relocate blocks efficiently, including those in read-only snapshots. The paper analyzes the effectiveness of these techniques at reducing fragmentation and improving overall performance across various storage media.

1 Introduction

File systems typically allocate physically contiguous blocks in storage devices to write out logically sequential data and metadata. This strategy maximally uses the write bandwidth available from each storage device since more blocks can be written to it using fewer write I/Os, and it allows for optimal performance when that data or metadata is later read sequentially. Common operations such as file creations, resizes, and deletions age a file system, resulting in *free space fragmentation* and *file layout fragmentation*. Free space fragmentation results in reduced contiguous physical allocations, which in turn lowers file system write throughput [32]. Furthermore, it limits the system's ability to optimally lay out logically sequential data and metadata, thereby contributing to file layout fragmentation [33, 35]. Fragmentation impacts the I/O performance of both hard drives (HDDs) and solid state drives (SSDs), although in different ways.

File sizes rarely align with the file system block size, thus there is potential for intra-block wastage of storage space. Some file systems provide the ability to address sub-block chunks to avoid such wastage and improve storage efficiency [25, 31, 37]. However, such sub-block indexing introduces the potential for *intra-block fragmentation*, which occurs as chunks within a block are freed at different times.

A copy-on-write (COW) file system never overwrites a block containing active data or metadata in place, which makes it more susceptible to fragmentation [12]. WAFL [14] is an enterprise-grade COW file system that is subject to free space, file layout, and intra-block fragmentation. In this paper, we present techniques that efficiently address each form of fragmentation in the WAFL file system, which we refer to collectively as *storage gardening*. These techniques are novel because they leverage WAFL's implementation of virtualized file system instances (FlexVol® volumes) [9] to efficiently relocate data physically while updating a minimal amount of metadata, unlike other file systems and defragmentation tools. This virtualization layer provides two advantages: (1) The relocation of blocks needs to be recorded only in the virtual-to-physical virtualization layer rather than requiring updates to all metadata referencing the block. (2) It even allows relocation of blocks that belong to read-only snapshots of the file system, which would be ordinarily be prohibited.

Most previous studies of fragmentation predate modern storage media (i.e., SSD drives) [32, 33, 34]. Other studies were performed on commodity-grade systems (with single drives) [5, 16]; these studies draw conclusions that do not apply to enterprise-grade systems. The WAFL file system can be persisted on a variety of storage media, which makes it well-suited for this study on fragmentation. We analyze each form of fragmentation and evaluate our defragmentation techniques with various storage media permutations.

To summarize our findings, we see significant improvements in data layout metrics on HDD- and SSD-based systems using our approaches. These improvements translate into significant performance gains on HDD-based systems, which are typically I/O-bound, as well as mixed-media (HDD and SSD) systems. In contrast, the same approaches generally show negative overall performance impact on all-SSD systems, which are more sensitive to the CPU overhead incurred by defragmentation. We conclude that for SSD-based systems, it is preferable (and advantageous) to perform defragmentation only during periods of low load. Our

lessons are applicable to other file systems as well, especially ones that are COW, such as ZFS [27] and Btrfs [31].

2 An Overview of WAFL

This section presents background on WAFL—an enterprise-grade UNIX-style file system—and the trade-offs inherent in defragmentation.

2.1 File System Layout and Transaction

A Data ONTAP® storage system uses the proprietary WAFL [14] file system, which is persisted as a tree of 4KiB blocks, and all data structures of the file system, including its metadata, are stored in files. Leaf nodes (L_0s) of an inode's block tree hold the file's data. The next higher level of the tree is composed of indirect blocks (L_1s) that point with a fixed span to children L_0s; L_2s point to children L_1s, and so on. The number of levels in the block tree is determined by the size of the file. Each inode object uses a fixed number of bytes to store file attributes and the root of its block tree, unless the file size is tiny, in which case the file data is stored directly within the inode. All inodes for data and metadata are arranged in the L_0s of a special file whose block tree is rooted at the superblock. WAFL is a copy-on-write (COW) file system that never overwrites a persistent block in place. Instead, all mutations are written to free blocks and the previously allocated blocks become free.

Client operations that modify the file system make changes to in-memory data structures and are acknowledged once they have also been logged to nonvolatile memory. WAFL collects and flushes the results of thousands of operations from main memory to persistent storage as a single atomic transaction called a *consistency point* (CP) [9, 14, 19]. This delayed flushing of "dirty" blocks allows better layout decisions and amortizes the associated metadata overhead. During each CP, all updates since the previous CP are written to disk to create a self-consistent, point-in-time image of the file system. A snapshot of the file system is trivially accomplished by preserving one such image. The WAFL *write allocator* assigns available free blocks of storage to the dirty blocks during a CP. The goals of the write allocator are to maximize file system write throughput and subsequent sequential read performance.

We have previously presented the data structures and algorithms used to steer the write allocator toward the emptiest regions of storage, with built-in awareness of RAID geometry and media properties [17]. Prior work has also described how CPs manage free space in order to maximize various performance objectives [8]. In this paper, we extend these concepts further, showing how storage gardening can increase the availability of high-quality regions for writing and recreate the desired layout after file system aging has undone the initial write allocation.

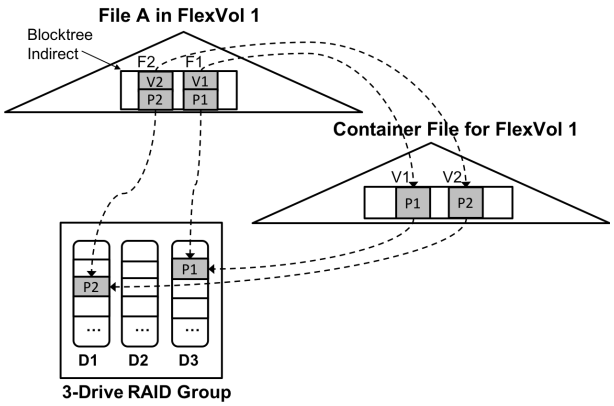


Figure 1: The relationship of a FlexVol volume with its container file and the aggregate. A real-world aggregate has many more drives.

2.2 FlexVol Volumes and Aggregates

WAFL defines collections of physical storage as *aggregates*, which are typically several dozen TiB in size. A WAFL aggregate can consist of different permutations of storage media: HDDs (hard drives) only, SSDs (solid state drives) only, HDDs and SSDs, SSDs and S3-compliant object stores, and LUNs exported from third-party storage. Storage devices with no native redundancy, such as HDDs and SSDs, are organized into RAID [6, 10, 29] groups for resiliency. Multiple aggregates are connected to each of two ONTAP® nodes that are deployed as a high-availability pair. Within each aggregate's physical storage, WAFL houses and exports hundreds of virtualized WAFL file system instances called *FlexVol volumes* [9].

Each aggregate and each FlexVol is a WAFL file system. A block in any WAFL file system instance is addressed by a *volume block number*, or *VBN*. WAFL uses a *Physical VBN* to refer to a block in the aggregate; the Physical VBN maps to a location on persistent storage. A block in a FlexVol is referenced by a *Virtual VBN*. FlexVols are stored within files that reside in the aggregate—the blocks of a FlexVol are stored as the L_0 blocks of a corresponding *container* file. The block number in the FlexVol (Virtual VBN) corresponds to the offset in the container file. Thus, the L_1 blocks of the container file are effectively an array that is indexed by the Virtual VBN to find the corresponding Physical VBN. In other words, container file L_1 blocks form a map of Physical VBNs indexed by the Virtual VBNs, which we call the *container map*. Data structures in the FlexVol store a cached copy of the Physical VBN along with the Virtual VBN pointers. In most cases, the cached Physical VBN helps avoid the extra CPU cycles and storage I/O for consulting the container map. It is possible for the cached Physical VBN in a FlexVol structure to become stale, in which case *the container map is consulted for the authoritative version*. Fig. 1 illustrates the

relationship between the Physical VBNs in an aggregate and the blocks of a File *A* in a FlexVol.

This virtualization of storage for blocks in a FlexVol and the corresponding indirection between Virtual VBNs and Physical VBNs through the container map provide the basis for the storage gardening techniques presented in this paper, as well as a wide range of technologies, such as FlexVol cloning, replication, thin provisioning, and more [9].

2.3 Performance and Defragmentation

Modifications to data and metadata in a COW file system, such as WAFL, fragment both the layout of files and the aggregate's free space. WAFL also supports sub-block addressing, and uses that to *compact* sub-4KiB chunks into a single block. These compacted blocks become fragmented as their constituent chunks are freed. Subsequent sections detail the impact of each form of fragmentation. Defragmentation is typically accomplished by relocating in-use blocks or chunks from badly fragmented regions of a file or file system. Relocating a block is trivial in many cases; the pointer stored in a parent indirect can be fixed up to point to a relocated child's new physical location. Although most file systems prevent relocation of blocks that belong to read-only snapshots, WAFL provides this functionality. Two requirements exist to support block relocation below the file system in the storage layer: (1) the ability to virtualize the address space, which WAFL provides in the form of FlexVol layering, and (2) the ability to detect stale pointers. Each of these abilities is detailed in Sec. 3.3. Although the CP amortizes the overhead associated with re-writing these blocks, defragmentation comes at a cost (CPU cycles and I/Os). An enterprise storage system must consider this cost in the context of the storage media type before it chooses to defragment. This paper explains the defragmentation techniques used by WAFL, and how these trade-offs play out in various Data ONTAP configurations.

3 Free Space Fragmentation

This section discusses the effect of free space fragmentation and the technique used to counter it.

3.1 Background on Space Fragmentation

WAFL groups HDDs and SSDs of an aggregate into RAID groups to protect against device errors and failures. As Fig. 2(A) shows, a *stripe* is a set of blocks, one per device, that share the same 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. Fragmentation of free space leads to *partial stripe writes*, shown in Fig. 2(B), which require RAID to read data blocks to compute parity [29]. Writing logically

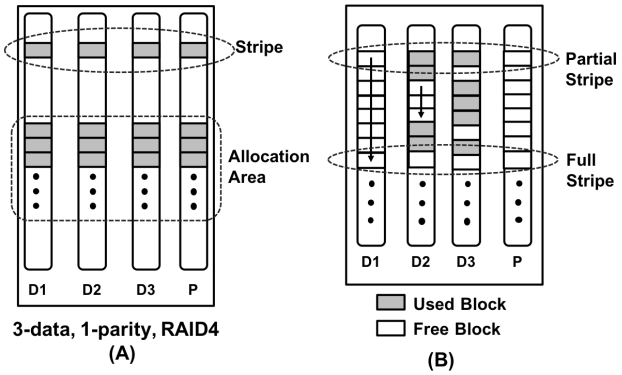


Figure 2: (A) A sample RAID-4 group with 3 data and 1 parity storage device (for simplicity). (B) A sample aged RAID group with free space fragmentation.

sequential blocks of the file system to consecutive blocks of a storage device reduces the total number of write I/Os to the device and improves sequential read performance, because the blocks can be read with a single I/O [2]. Contiguous free space on devices, such as on D1 in Fig. 2(B), is required to meet this objective, by facilitating long *write chains*. Fragmented free space decreases the availability of contiguous free blocks on each device, as shown on drives D2 and D3.

The latency of a write operation is not *directly* affected by free space fragmentation because WAFL acknowledges a write operation immediately after it is logged to nonvolatile memory. Fragmentation makes each CP more expensive, which indirectly impacts client operations. First, more CPU cycles are required to find free blocks to allocate [19] and compute RAID parity, which causes the WAFL scheduler to divert more CPU away from client operations so the CP can complete in time. Second, more I/Os of shorter write chains are required to flush out all the dirty blocks of the CP, which takes storage I/O bandwidth away from client operations.

Fragmentation can also impact performance by making free space reclamation more expensive in terms of CPU cycles and metadata updates. Over time, several improvements to free space reclamation have ensured that WAFL now performs efficiently even in the presence of fragmentation [19]. However, this concern still applies in most other file systems.

3.2 Segment Cleaning in WAFL

The goal of free space defragmentation is to make emptier regions of free space available to the write allocator. In-use blocks need to be efficiently relocated to create large areas of free space without violating invariants associated with blocks in FlexVol snapshots. Prior work [17] describes how the WAFL write allocator segments each RAID group into *allocation areas (AAs)* when choosing free Physical VBNs for the CP. As Fig. 2(A) shows, an AA is a set of consecutive RAID stripes; the AA size depends on storage media prop-

erties [17]. Defragmentation operates by *segment cleaning* at the AA granularity. The cleaning of an AA entails consulting free space metadata in WAFL [18] to pick stripes in the AA that are worth cleaning, reading all in-use blocks of such stripes into the buffer cache, and tagging them dirty. WAFL stores a *context* together with each written block [36], which identifies its file and file block offset[1]. Cleaning uses this context to determine the file and offset of the in-use block and marks the buffer dirty in the corresponding file. The subsequent CP processes these dirty buffers (together with all others) and writes them out to new Physical VBNs, thereby freeing the previously used blocks and creating an emptier AA. The parent indirect block of such a rewritten block (much like that of any dirty block) is updated by the CP to reflect its new Physical VBN.

3.3 Blocks in the FlexVol Volume

The vast majority of the blocks in an aggregate belong to its FlexVols, because they contain user data. WAFL leverages the indirection provided by the FlexVol virtualization layer to efficiently relocate FlexVol blocks. In particular, such blocks are relocated by loading and dirtying them as L_0 blocks of the corresponding container file, rather than as blocks in the block tree within the volume. Thus, a relocated block gets a new Physical VBN, but its Virtual VBN remains unchanged and no changes are made *within* the volume. Fig. 3 shows an example in which blocks are moved from within allocation areas AA_x and AA_y. The cleaner determines all in-use blocks (i.e., p_1–p_5) in these AAs and reads each of these blocks into memory, along with its associated context. The context for a block in a FlexVol refers to its container file and Virtual VBN. Thus, p_1–p_3 of File A are marked as dirty L_0s of FlexVol 1's container file, and p_4–p_5 are marked as dirty L_0s of File B (a metadata file in the aggregate). In the subsequent CP, the write-allocator rewrites these blocks together with other dirty buffers to a new AA_z, thereby emptying AA_x and AA_y. Note that File A's indirect blocks continue to point to stale Physical VBNs p_1–p_3 (as discussed in detail later in this section), whereas File B and the container file of the FlexVol are up to date.

Leveraging the virtualization layer provided by the container file yields two key benefits. First, it facilitates relocation of blocks in a snapshot because file system invariants associated with snapshots are preserved within the FlexVol layer. Blocks in a snapshot image of the FlexVol are immutable and therefore forbidden from being dirtied and processed by the CP. Such rules are typical across file system implementations. In theory, it is possible to physically relocate blocks within snapshots without virtualization, but this requires the ability to update metadata within a snapshot to

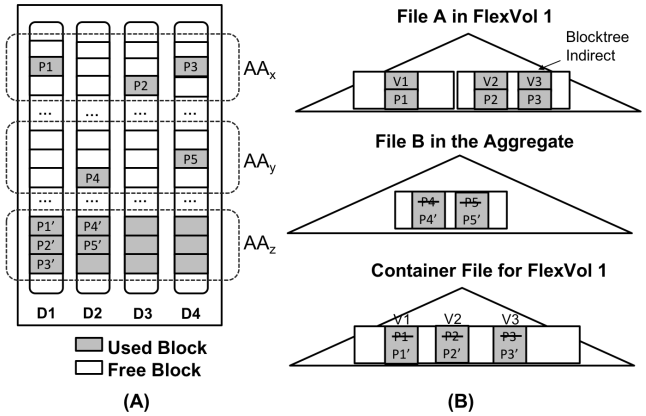


Figure 3: (A) 4 drives with 5 blocks randomly allocated in AA_x and AA_y and the same 5 blocks relocated to AA_z to create empty AAs. (B) Impact of block relocation due to segment cleaning within the L_1s of File A in FlexVol 1 and a File B in that aggregate, and the relevant changes to the container file.

reflect those relocations. In a COW file system, such updates cascade up the file system tree, resulting in further updates. Physical-only relocation via the container preserves the Virtual VBN, and that leaves the FlexVol snapshot image intact, including all file system metadata for block allocation. Given the popularity of FlexVol snapshots, such block relocation is critical to efficient defragmentation[2].

The second benefit is that the requisite metadata updates are minimized. Relocating a block is expensive in a COW file system like WAFL because every ancestor block in the file system tree needs to be rewritten to point to the new location of its child. By leveraging the container map for the FlexVol, blocks within the FlexVol are not rewritten. Further, as described in Sec. 2.1, the file system tree of blocks rooted at the superblock of a FlexVol can be quite tall, whereas the height of the container file is a function only of the size of its Virtual VBN space. Thus, the tree of blocks comprising the container file is significantly shorter, and higher-level indirects of a container file are likely to be already dirty due to the batching effect of the CP.

A file system operation that accesses a relocated block of File A uses the stale pointer in that indirect, such as p_1, to read a block from storage. If WAFL has not yet assigned the previously freed p_1 for a new write, the context check succeeds and the I/O is accomplished. Otherwise, the context check fails and the operation pays a *redirection penalty* to consult the container map, using v_1 to read p_1'. The pointer in File A's parent indirect block can optionally be fixed to p_1' either via a background scan or opportunistically after the redirected read to avoid the penalty on subsequent accesses. The use of a virtualization layer in this case provides both

[1] This context was introduced originally to protect against lost or misdirected writes [3], so that a subsequent read can detect a mismatch from the expected context.

[2] Snapshots of an aggregate are rare and short-lived, so their interaction with segment cleaning is limited, and is not discussed here.

the ability to defer fix-up work and the option to leave stale pointers in indirects in cases where the update would not be expected to improve performance. Without this virtualization, all references to the physical block would have to be corrected immediately. Although Fig. 3 depicts only L_0s of File A being relocated, any block in the block tree of any file in the FlexVol can be relocated.

3.4 Continuous Segment Cleaning

Segment cleaning was first introduced for all-HDD aggregates as a background scan that walked all AAs in each RAID group. It was expensive and had to be initiated by the administrator during periods of low load. A later release introduced *continuous segment cleaning* (CSC), which runs autonomously and is more efficient. It cleans AAs just in time as they get selected for use by the write allocator. Prior work [17] shows how the WAFL write allocator uses a max-heap to pick the emptiest AA from each RAID group. Cleaning the emptiest AAs minimizes the number of in-use blocks that are required to be relocated, which in turn minimizes the total number of I/Os and CPU cycles required for this activity. This greedy approach also minimizes the subsequent redirection penalty and fix-up work for the file system.

4 File Layout Fragmentation

This section discusses how files become fragmented in WAFL and the approach used to counteract that fragmentation.

4.1 Background on File Fragmentation

The WAFL write allocator attempts to allocate consecutive L_0s of a file sequentially on a single storage device to optimize subsequent sequential read performance. Given that WAFL is a COW file system, this layout may fragment over time. That is, even if sequential file L_0s are initially stored contiguously, continued random overwrites of the L_0s can cause them to be rewritten elsewhere. It should be noted that neighboring offsets in a file need to be overwritten several seconds apart to fragment the file because the CP collects and processes a few seconds' worth of dirty buffers. An example of suboptimal file layout is shown in Fig. 4, in which sequential L_0s of File A are scattered across the aggregate, with Physical VBNs p_1–p_5.

Sequential reads of a fragmented file require an increased number of drive I/Os [2]. Like most file systems, WAFL detects sequential patterns in the accesses to a file, and speculatively prefetches L_0s based on heuristics. (Prefetch heuristics used by WAFL are outside the scope of this paper.) Although prefetching helps sequential read performance, the associated overhead (CPU cycles and storage I/Os) increases with fragmentation of the file layout [2, 34].

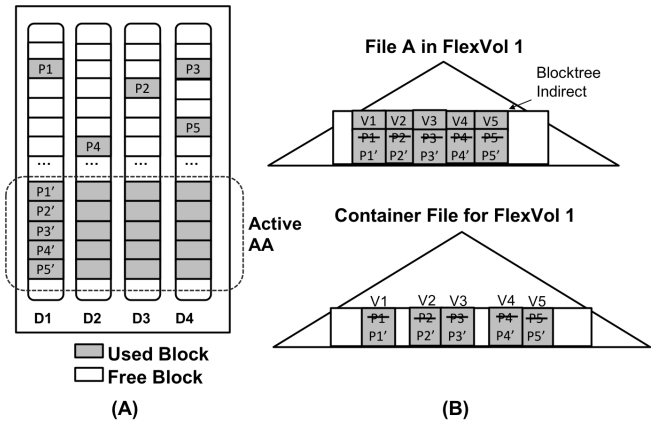


Figure 4: (A) 4 devices with 5 blocks located sequentially within a file but randomly on the storage media due to file layout fragmentation, and the same 5 blocks relocated sequentially on device D1. (B) Impact of this block movement within the L_1s of a File A in a FlexVol volume and the changes to that volume's container file.

4.2 File Defragmentation in WAFL

In theory, file layout defragmentation can be trivially accomplished in any COW file system by dirtying sequential file blocks that are not sequentially stored. These blocks will be written out sequentially during the subsequent write allocation process. File systems have two choices for how to deal with blocks shared with snapshots: (1) Update all indirect blocks pointing to the relocated block, which is not feasible because it requires modifying blocks in a snapshot. (2) Update references to the block in the "active" file system only and leave the block in place in the snapshot, which results in divergence from the snapshot and wasted storage for duplicate copies of these blocks. In WAFL, when a block in a FlexVol is dirtied, it is assigned not just a new Physical VBN but also a new Virtual VBN. The allocation of a new Virtual VBN is reflected in the FlexVol metadata, which results in divergence from the most recent snapshot of the volume. Efficient replication technologies minimize the amount of data transferred in each periodic incremental update [30], which is accomplished by *diff'ing* per-snapshot metadata to efficiently compute changes to the file system.

The need to keep FlexVol metadata intact motivates another physical-only block relocation strategy by leveraging container file indirection. In particular, WAFL tags file-defragmented blocks as *fake dirty*, which conveys that the content of the data block is unchanged and should not diverge from any snapshot to which it belongs. In the next CP, a fake dirty buffer is assigned a new Physical VBN without changing its Virtual VBN. Fig. 4 shows the result of this process on File A; the L_0s retain their Virtual VBNs while getting reallocated sequentially from p'_1 to p'_5. Thus, these relocated blocks do not create false positives during the aforementioned snapshot *diff* process, and WAFL replication tech-

nologies remain efficient. Although Physical VBNs cached in snapshot copies of indirect blocks become stale, a failed read is redirected through the container map to the new location of the block.

File layout defragmentation in WAFL is similar to free space defragmentation, in that a relocated block only acquires a new Physical VBN. However, file defragmentation is different in two ways that make fake dirties more effective for this use case. First, the file blocks being dirtied are by definition contiguous in the file block space, so the COW-related overhead for the block tree in the FlexVol is amortized across multiple fake dirty blocks. Second, file defragmentation is triggered in cases where future sequential file accesses are anticipated (as discussed in Sec. 4.3), so it is desirable to "fix up" the block tree indirects right away rather than deferring the effort.

Relocating L_0 blocks that are shared with other files as a result of deduplication or file cloning does not create inconsistencies. For example, if some File B shares the first L_0 of File A, the parent L_1 in File B is not changed by the defragmentation of File A and therefore points to v_1 and p_1 even after the L_0 of File A is relocated to p'_1. As described in Sec. 3.3, once the now-free p_1 is reused, any subsequent read via that stale pointer in File B fails the context match, and is redirected to p'_1 via the container map. It should be noted that relocating blocks p_1–p_5 in File A could potentially fragment File B if it shares some of these blocks but at different offsets. However, fragmentation resulting from deduplication is unlikely in real-world datasets and as far as we know has not been encountered in our customer deployments. We find that multiple L_0s are shared in the same order between files, so defragmentation of one helps all such files[3].

4.3 Write After Read

File defragmentation was originally introduced as an administrator-initiated command to kick off defragmentation of a specific file or all user files in a FlexVol. When invoked, the Physical VBN pointers in the L_1s of the file are inspected to determine whether defragmentation could result in improved read performance. If so, the L_0 blocks are read into the buffer cache and tagged fake dirty. Autonomous file defragmentation—*write after read* (WAR)—was introduced in a later release. When enabled, it uses heuristics to defragment files that get accessed sequentially by client operations, but only when the system has sufficient availability of CPU cycles and I/O bandwidth. These techniques can also be applied to metadata such as directories[4].

[3]WAFL deduplication code paths track how many consecutive file blocks are detected as duplicates and replaced. Statistics from customer deployments show this number to be mostly in the 4 to 8 range.

[4]WAFL uses several techniques to optimize metadata access, some of which are described in prior work [19]. Such optimizations have made it unnecessary to employ WAR on metadata.

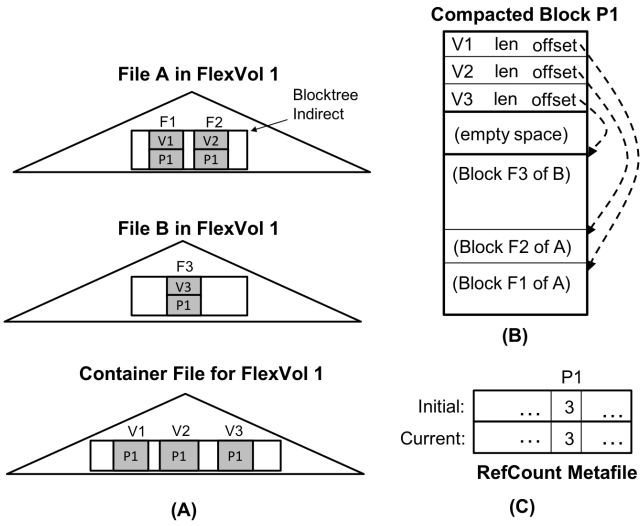


Figure 5: (A) Files A and B with sub-block chunks that share a physical block and the corresponding container file. (B) Format of the physical block containing the three chunks. (C) The refcount file with an "initial" and "current" value of 3 for physical block p_1.

5 Intra-Block Fragmentation

This section describes intra-block fragmentation in WAFL and its mitigation by using the virtualization layer. There are two sources of intra-block wastage in WAFL. First, when the size of a file is not an exact multiple of 4KiB (the block size used by WAFL), the unused portion of the last L_0 of that file is wasted. This may result in significant wastage of storage space, but only if the dataset contains a very large number of small files. Second, data compression in WAFL can result in intra-block wastage even for large files. WAFL uses various algorithms to compress the data stored in two or more consecutive L_0s of a file, and writes the compressed result to the file. Thus, a user file L_1 points to fewer L_0s and some "holes" to indicate blocks saved by that compression. Each set of such L_0s is called a *compression group*. The compressed data rarely aligns to the 4 KiB block boundary; therefore space is almost always wasted in the tail L_0 of every compression group. For example, 8KiB of data may compress down to 5KiB. This would consume two 4KiB blocks of storage where 3KiB of the second block is wasted. Compression has become ubiquitous in Data ONTAP deployments, with significant savings reported for large-file datasets. *Sub-block compaction* enables WAFL to pack multiple sub-block chunks from tails of one or more compression groups and/or files into a single physical block.

5.1 Sub-Block Compaction

Sub-block chunk addressability leverages FlexVol virtualization to remain transparent to the FlexVol layer. Blocks within

a FlexVol retain their Virtual VBNs, but now multiple Virtual VBNs can share one Physical VBN. The corresponding physical block contains the sub-block chunk associated with each Virtual VBN. Similarly, a container file can contain multiple references to the Physical VBN of a single compacted block. Fig. 5(A) shows how a compacted block p_1 is pointed to by three Virtual VBNs from a container file. These chunks are from two different files, A and B, and can be either tails of compression groups or uncompressed but partially filled blocks. As Fig. 5(B) shows, compacted block p_1 starts with a vector of tuples, which allows for chunks of different lengths to be compacted together. When a block is read, the tuples are parsed to locate the desired data. Each WAFL instance has a *refcount metadata file* that tracks references to a block [19]. The refcount file of an aggregate tracks in-use chunks in a compacted Physical VBN[5].

This design offers several clear benefits. First, compaction through the container file keeps it independent of the FlexVol. Thus, although blocks in a snapshot are immutable, they can be compacted or recompacted via the container file. Second, without compaction, compression is beneficial only if the compression group yields at least one block in savings, whereas compaction can exploit savings of less than 4KiB. Third, there is no fixed chunk size, which means that sub-blocks can be compacted together based on workload-aware criteria rather than on their sizes. For example, client-access heuristics can be used to compact together "hot" blocks that might get overwritten soon versus "cold" blocks.

5.2 Recompaction

Over time, one or more chunks within a compacted physical block may get freed due to overwrites or file truncations, which results in intra-block fragmentation within previously compacted blocks, providing an opportunity to reclaim wasted space via *recompaction*.

Recompaction in WAFL is performed by a *recompaction scanner* that walks the container map in Virtual VBN order and chooses blocks to defragment. The per-Physical VBN entry in the refcount file contains two sub-counts: r_i, the initial number of chunks in the Physical VBN when it was first written out, and r_c, the current number of chunks referenced by the container map. As shown in Fig. 5(C), both sub-counts are initialized to the number of chunks when a compacted block is written out. As chunks are freed, r_c is decremented. The recompaction scanner uses the two sub-counts to predict whether a block is worth recompacting. If $\frac{r_c}{r_i}$ is below some threshold (specified by the administrator based on desired aggressiveness), the block is read in from storage and its contents are examined to determine the actual free space in the block. If the block is truly worth recompacting, each chunk is marked dirty as a standalone container L_0 block. The subsequent CP applies compaction to all such blocks and writes them out in newly compacted blocks. In this way, Physical VBNs are changed transparently under the FlexVol virtualization layer, leaving stale Physical VBNs cached in FlexVol data structures.

A fourth type of fragmentation occurs in Data ONTAP that is similar to intra-block fragmentation. Recently introduced all-SSD FabricPool aggregates collect and tier cold blocks as 4MiB objects to an object store, for example to a remote hyperscaler such as AWS. As with compacted blocks, objects may become fragmented due to block frees, and objects can be freed only once all used blocks are freed. Object defragmentation consists of marking all blocks within sparsely populated objects dirty and rewriting them into new objects. It leverages the container file indirection to avoid changes within the FlexVol when the block is moved. Defragmentation is triggered by comparing the monetary cost of wasted storage versus the cost of GETs and PUTs to rewrite the defragmented data. Thresholds of allowed free blocks in an object are defined per-hyperscaler such that the cost of GETs and PUTs to defragment objects breaks even within a month when compared to savings from the reduced storage.

6 Interactions between Techniques

CSC, WAR, and recompaction can run concurrently and with very little adverse interaction. Segment cleaning generates empty AAs for use by the write allocator, which naturally facilitates efficient file block reallocation. No additional requirement is placed on CSC because of WAR. CSC and recompaction both operate by generating dirty L_0s of container files. A block being relocated by CSC may be a compacted block, in which case it may be unnecessarily recompacted in order to reclaim the old Physical VBN.

In theory, defragmentation techniques could invalidate a large number of cached Physical VBNs, which may affect performance because of more fix-up work and/or read redirection penalty. As described in this paper, defragmentation is used with care and only when the associated overhead is justified; as far as we know, no customer systems have been impacted by any pathological scenarios of defragmentation[6].

7 Evaluation

It is not practical to formulate an apples-to-apples comparison of the defragmentation techniques in WAFL with that in other file systems, due to the configurations, sizes, and its large feature set. Instead, this section provides some historical context and explores the trade-offs inherent to each of

[5] The refcount file supports deduplication; blocks from different files and FlexVols may refer to the same Physical VBN. Prior work [19] studies the performance implications of maintaining a refcount file.

[6] Specific features unrelated to defragmentation, such as Volume Move [1], may create scenarios leading to severe redirections, but purpose-built scanners have been designed to handle them.

our techniques across some key configurations.

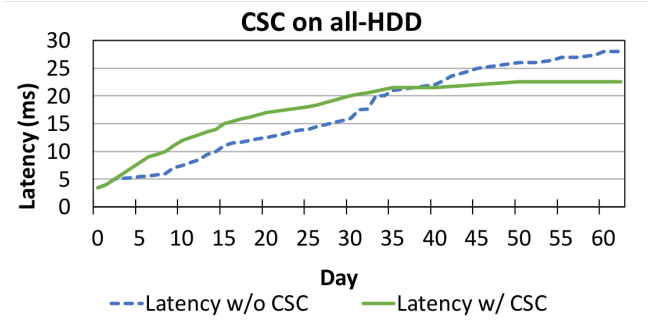
Data ONTAP is deployed by enterprises in different business segments for a wide variety of use cases. A typical NetApp storage controller might be hosting datasets for multiple instances of different applications that are actively accessed at the same time. In such multitenant environments, no individual customer workload represents the range of possible outcomes. Instead, we use a set of microbenchmarks and an in-house benchmark that represent specific average and worst-case scenarios, but the conclusions are applicable across the majority of benchmarks that we track. The IOPS mix—random reads, random and semisequential overwrites—of the in-house benchmark is designed to be identical to that of the industry-standard SPC-1 benchmark [7], and models the query and update operations of an OLTP/DB application. We generate load by using NFS or SCSI clients based on convenience, but the choice of protocol does not make any material difference to the results presented. Unless otherwise specified, all experiments use our midrange system, which has 20 Intel Xeon E5v2 2.8GHz 25MB-cache cores (10-cores x 2 sockets) with 128GiB of DRAM and 16GiB of NVRAM.

7.1 Free Space Defragmentation

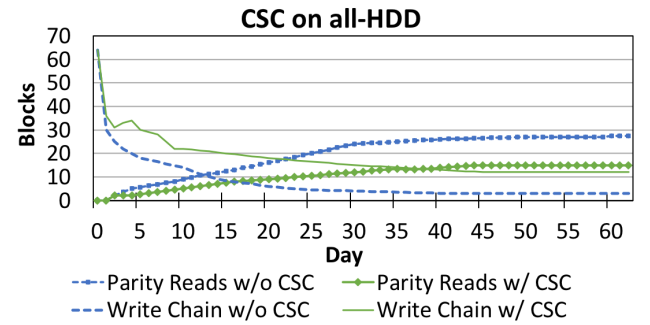
The original ONTAP deployments (20+ years ago) were HDD-only, and WAFL had no defragmentation capability at the time. Because those systems were typically bottlenecked by hard-drive bandwidth, higher performance required attaching more HDDs. Segment cleaning was designed into WAFL soon after the FlexVol layering was introduced. However, it could be initiated only by the administrator, based on observed system performance. In most cases, background defragmentation scans were configured to run during weekends or known times of low load. Just-in-time segment cleaning (CSC) was introduced later, as discussed in Sec. 3.2. Here, we evaluate the performance benefits of CSC across different storage media configurations.

To evaluate the benefits of CSC on all-HDD aggregates, we directed a load of 3K ops/sec of the in-house OLTP/DB benchmark using clients connected over Fibre Channel to a low-end 8-core system with an aggregate composed of 22 10K RPM HDDs of 136GiB each. Fig. 6 presents the results of our test over a 60+ day window[7]. Without CSC, client observed latency continues to increase over time due to increasing fragmentation in the file system. As shown in Fig. 6(A), CSC carries some initial overhead that results in higher client latency for the first 35 days, but it eventually delivers layout benefits that yield a stable and lower client latency over time. Both write chain length and parity reads are greatly improved by using CSC (Fig. 6(B)), as write chain length converges toward a worst-case value of 1 without CSC.

[7]It takes a long while to fragment a real-world-sized all-HDD aggregate, given its low IOPS capability; all-SSD aggregates can be fragmented faster.



(A)



(B)

Figure 6: (A) Client observed operation latency and (B) parity reads/sec and write chain lengths with and without CSC on an all-HDD aggregate over 62 days running an OLTP/DB benchmark.

Introduced in 2012, NetApp Flash Pool® aggregates mix RAID groups of SSDs together with RAID groups of HDDs. At that time, enterprise-quality SSDs were 100 to 200GiB in size and relatively expensive. Therefore, based on cost-benefit analysis for ONTAP systems, SSDs could make up at most 10% of an aggregate's total capacity. WAFL used heuristics to determine where a particular block was to be stored. For example, "hot" (based on access patterns) data and metadata blocks were stored or even cached in the SSDs, and "cold" blocks were stored in or tiered down to HDDs.

An interesting effect of biasing hot blocks to SSDs was that fragmentation was mostly isolated to the SSD tier, and infrequent deletion of cold blocks in the HDD tier was insufficient to fragment the HDD tier. This was verified by repeating the OLTP/DB experiment previously described, but at a higher load on a Flash Pool aggregate composed of 12 SSDs and several HDDs. Hot spots of the working set stayed in the SSD tier, and write chains to HDDs declined very slowly, leveled out at around 48 blocks after 22 days, and remained stable for the remainder of the measurement interval (60+ days), without any need for CSC; the graph for this experiment is not shown.

On the other hand, the SSD tier of a Flash Pool aggregate fragments very quickly. We studied this by running the OLTP/DB benchmark benchmark on the midrange system with 12 200GiB SSDs and a large number of HDDs.

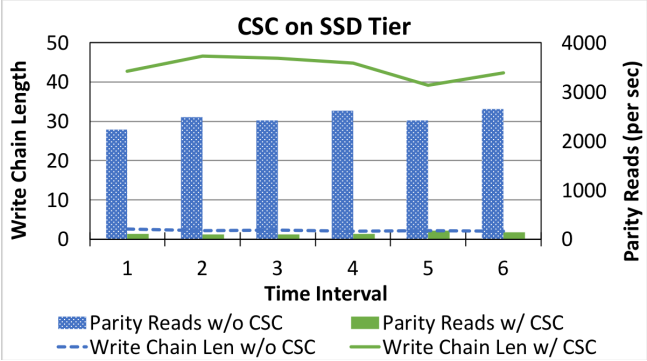


Figure 7: Parity reads per second and write chain lengths for the SSDs in a mixed aggregate during the in-house OLTP/DB benchmark, with and without CSC on the SSD tier.

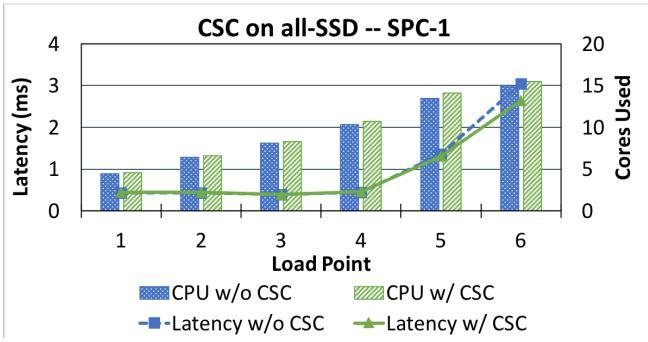


Figure 8: Latency and CPU utilization with the in-house OLTP/DB workload on an all-SSD aggregate, with and without CSC.

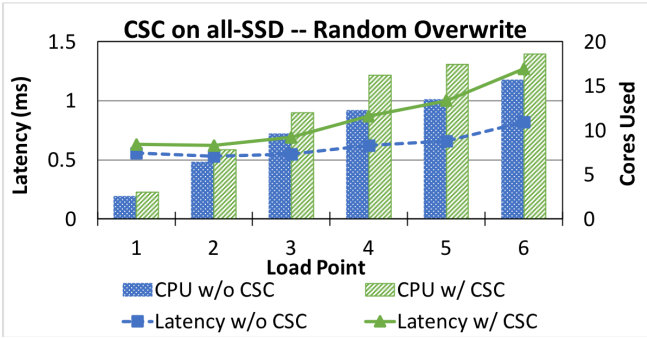


Figure 9: Latency and CPU utilization with a pure random overwrite workload on an all-SSD aggregate, with and without CSC.

The SSD tier is fragmented by running a heavy load (100K op/sec) for 2 hours, followed by another 2 hours of a more moderate and recommended load (50K op/sec) for that configuration. Fig. 7 shows write chains and parity reads per second within the SSD tier during the latter period. Both metrics show a marked improvement with CSC, with write chain lengths of about 40 instead of 2. Although CSC results in a small increase in client latency—from 0.79ms to 0.82ms (not shown here)—it was still beneficial for this earlier generation of SSDs, which were more prone to wear out. SSDs have a flash translation layer (FTL) that generates empty *erase blocks* for new writes and evenly wears out the SSD by moving data around within the SSD [28]. It is well known that shorter and more random write chains lead to higher *write amplification* on SSDs, which impacts SSD lifetime. Prior work [17] explains how the choice of the AA size in WAFL minimizes negative log-on-log behaviour [41] in devices using translation layers such as SSDs or SMR drives.

Following architectural improvements to the WAFL I/O path, ONTAP systems with all-SSD aggregates were introduced in 2015. As the size of the enterprise-quality SSD has increased from 100GiB to 16+ TiB in less than 5 years (remarkably), and the promise of new interconnect technologies such as NVMe [39, 40] has become a reality, the performance bottleneck in these storage systems has shifted from the media to the available CPU cycles to maximally use storage I/O bandwidth. In addition, emphasis has shifted away from SSD lifetimes and avoiding burnout due to write amplification in favor of total cost of ownership benefits as vendors manufacture SSDs with larger *drive writes per day* [4]. RAID-style fault tolerance also provides protection from burnout. Thus, write amplification due to free space fragmentation on enterprise-class SSDs is a performance problem that manifests mostly when a storage server has an excess of free CPU cycles but insufficient SSD I/O bandwidth. This is unlikely on systems with RAID-based redundancy, which require a minimum number of SSDs to amortize the space needed for storing RAID parity; ONTAP aggregates contain anywhere from 12 to 20+ SSDs. Much higher performance with consistently low operational latency is required of all-SSD systems, and therefore these systems are sensitive to changes in available CPU cycles.

We first evaluate CSC on all-SSD systems by running the OLTP/DB benchmark on the midrange system with an aggregate comprising 21 SSDs of 1TiB each. The aggregate was first filled to 85% of its capacity and subjected to severe load for approximately 1 day, until fragmentation metrics plateaued. Fig. 8 presents the achieved latency and CPU utilization at discrete increasing levels of load on this preaged dataset. The use of CSC dramatically improves write chain length, from 10.9 to 60.6 blocks, and nearly doubles the read chain length, from 2.2 to 3.7 blocks, by providing emptier AAs for write allocation; this is not shown in the figure. The CPU overhead of CSC is limited to a fraction of a core, because writes represent only a portion of the load to the system and therefore demand for clean AAs is limited. Despite the layout improvements with CSC, performance is unaffected until the maximum load is requested, where CSC reduces latency from 3.0ms to 2.6ms.

Given the variety of workloads and the prevalence of multitenancy on deployed systems, performance cannot be fairly evaluated by any one benchmark. Thus, we next target a

pure random overwrite to the same pre-aged setup as just described to increase demand for clean AAs and determine *worst-case* CSC overhead, as shown in Fig. 9. These results do not represent the expected behavior in practice, but they can inform the decision of whether to enable the feature by default. Without CSC, write chain lengths quickly degrade to 3, but with CSC they never fall below 12. Parity reads/sec without CSC are around 10 times those with CSC. However, CSC consumes significant CPU cycles in this workload—almost 3 out of the 20 cores—which results in higher latencies, especially at higher load.

Given the ubiquity of SSDs these days, all-HDD aggregates are now mostly used for backup and archival purposes. Such systems do not experience sufficient free space fragmentation to benefit from CSC, so it is disabled by default. CSC can be (and is) enabled on customer deployments that target more traditional I/O loads to achieve the benefits we describe. In hybrid SSD and HDD aggregates, CSC is enabled only on the SSD tier to provide reduced write amplification and extend device lifetimes. Finally, our results show that free space fragmentation plays a smaller role in all-SSD configurations (with sufficient SSDs) than do CPU bottlenecks. Given the expectation of consistent low latency at higher IOPS from all-SSD systems, and the higher endurance of modern enterprise-quality SSDs, CSC is disabled by default on such systems. Free space defragmentation can still be enabled on a case-by-case basis or performed by a background scan as needed at known periods of low load; Sec. 7.4 discusses this further.

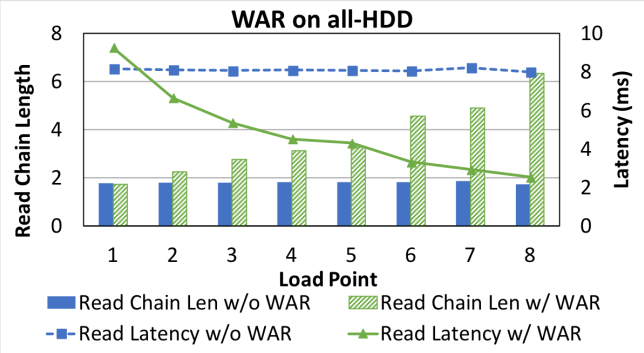


Figure 10: Read chain length and read operation latency on an all-HDD aggregate, with and without WAR. Increasing loads of sequential read are run for fixed intervals to fragmented files.

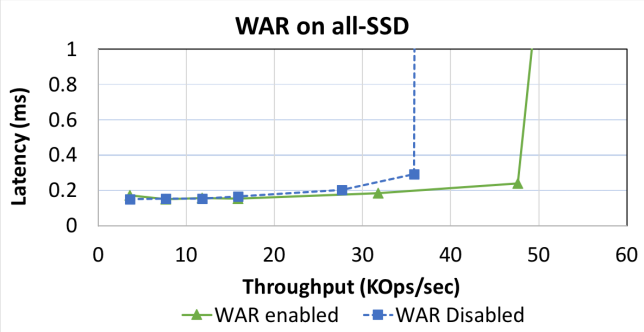


Figure 11: Latency versus achieved throughput of sequential reads to a pre-aged dataset. With WAR enabled, measurements were taken after WAR had completed defragmenting file layout.

7.2 File Layout Defragmentation

As discussed in Sec. 4.3, WAFL uses WAR to defragment file layouts, and that reduces the number of drive I/Os required to satisfy a client request. We use the *read chain length*—the number of consecutive blocks read by a single request to a drive—as a primary metric for measuring file fragmentation. In these experiments, we used an internal tool to generate pre-fragmented datasets on the 20-core midrange system. The tool randomizes the Physical VBNs assigned to the L_0s of a set of files, which efficiently mimics file fragmentation.

First, we analyze all-HDD aggregates. Sequential reads of 64KiB each from several clients were aimed at the system with an all-HDD aggregate. Fig. 10 presents both the average latency and read chain lengths at increasing levels of that load. Without WAR, both metrics remain stable at around 8ms and 1.8 blocks, respectively. With WAR enabled, the incoming read operations trigger WAR to improve file layout, which translates into longer read chains and reduced latency in successive load points. Overall client read throughput (not shown in the graph) also improves when using WAR, from 1GiB/sec without WAR to 2.6GiB/sec with WAR at the higher load points. At the drive level, the number of read I/Os decreases from 639 to 345 per second, in spite of the significantly higher read throughput. This reduced load translates to a reduction in I/O latency, measured at the drive, from 3.9ms to 1.2ms. As expected, WAR writes to the storage to relocate fake dirty blocks, but the reduction in the number of drive-reads outweighs these writes.

We now evaluate WAR for all-SSD aggregates by replacing the HDDs in the previous experiment with 21 1TiB SSDs. We first isolate the benefits of file layout improvements on the performance of sequential reads by measuring throughput and latency *after* WAR has completed defragmenting the pre-aged dataset, as shown in Fig. 11. While the file fragmented test sees average read chain lengths of 1.7 blocks, post-WAR the system sees read chains of 32 blocks. Thus, file defragmentation significantly reduces SSD read I/Os per second and lowers CPU cycles needed by the storage driver code in ONTAP. As a result, with WAR, the system is capable of much higher throughput before the system saturates and latency climbs to unacceptable levels. This experiment demonstrates that file layout can still have a major impact on read performance on SSDs, even though random read performance is less of a factor than on HDDs.

As mentioned earlier, all-SSD systems are CPU-bound,

and operational latencies are more sensitive to CPU consumed by other activity. To evaluate the *worst-case* performance impact of WAR overhead and inform the enablement of this feature by default, we issued a mixed workload of sequential read and sequential write on the same fully pre-aged dataset (graph not shown). Writes are more CPU intensive than reads and so are a better indicator of CPU interference. In this test, WAR overhead pushes operation latency up from 1.7ms to 2.5ms and throughput is lowered from 2GiB/sec to 1.7GiB/sec. The WAR interference particularly comes from the increase in drive writes and 1.6 extra cores used in an already CPU-saturated system.

All-HDD backup and archival systems typically experience sequential writes (backup transfer streams) and sequential reads (restore transfer streams), and they get fragmented by the deletion of older snapshots and archives. Thus, WAR is beneficial to such deployments. WAR is disabled on all-SSD platforms due to its performance overhead, but can be enabled as needed during periods of low activity to achieve the demonstrated file layout benefits.

7.3 Compaction and Recompaction

Sec. 5 presented the compaction technology in WAFL to pack multiple compressed blocks within a single block on persistent storage. In our evaluation, we first created 2.2TiB of data across several files in the midrange system with 21 1TiB SSDs. The data written to these files was designed to be highly compressible. Once created, this data set consumed only 511GiB of physical storage, representing a 1.7TiB (77%) savings due to the cumulative effect of compression and compaction. In particular, compaction was able to store an average of 4.8 chunks per block. These were large files, so the benefit of compaction was primarily due to the tail-end of compression groups being compacted together.

The compacted dataset was then fragmented by using random overwrites until the storage savings were reduced to 27%, indicating sparsely compacted blocks and significant intra-block fragmentation. Overwrites of compressed user L_0s resulted in the freeing of chunks within compacted blocks. Then we initiated a moderate random read load (80MB/sec) with and without a recompaction scanner to measure the rate of intra-block defragmentation and the associated interference to the client workload. We observed storage space being reclaimed at a rate of 3.78GiB/min, with somewhat significant impact on client latency. In particular, we saw the average latency of the client read operations rise from 0.63ms to 2.07ms, which comes from an additional 1.25 cores worth of CPU cycles, as well as the additional blocks written to the SSD drives. The primary reason for the latency increase is a background scan[8] that runs after recompaction, but at a coarser parallelism that precludes client operations. Until that limitation is fixed and recompaction is made sufficiently lightweight, it should be initiated by the administrator only at known time periods of low load. Once recompaction is made lightweight, it can run autonomously.

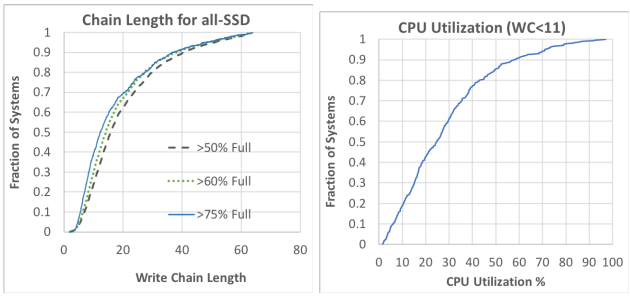
[8]This scan is mentioned in Sec. 3.3, and is used to fix up stale Physical VBNs in indirect blocks of the FlexVol.

7.4 Customer Data and Summary

We mined data from customer deployments running a recent Data ONTAP release on all-SSD configurations, with different space utilization levels (aggregate fullness); Fig. 12(A) presents the distribution of write chain length observed. It shows that higher space utilization adds to the fragmentation effect caused by file system aging—higher utilization shows smaller write chain lengths. About 40% of systems that were at least 75% full have write chain lengths below 11; such systems stand to benefit from CSC. Logs collected over a recent 3-month period also showed that 17% of all-SSD systems experienced the less-efficient administrator-invoked segment cleaning (versus 7% for all-HDD systems), justifying the need for CSC. A similar analysis across all-HDD systems (not shown here) reveals that write chain length distribution skews to higher values. There are two reasons for this. One, it takes much longer to fragment an all-HDD aggregate given its lower IOPS capability, while our data was from a recent release. Two, a sizeable fraction of these systems are archival, hosting "secondary" FlexVols that are replicas of FlexVols accessed by customer applications. Incremental updates to such FlexVols [30] are slow to fragment free space because they overwrite (and free) large ranges of blocks.

To determine available CPU headroom, we next mined CPU utilization during a particularly busy hour of a weekday, but specifically for all-SSD systems with write chains less than 11. Fig. 12(B) shows that about 85% of systems had CPU utilization of less than 50%; a similar trend was seen across all-HDD systems as well. This indicates that enabling CSC would not really impact client operations. Further data mining also showed that CPU utilization of systems varies during a day, depending on the workload and the time zone, as well as the customer workflow. Several features in Data ONTAP autonomically detect periods of low user activity to selectively enable themselves; our data indicates that CSC, WAR, and recompaction can behave similarly.

To summarize this section, CSC and WAR are consistently able to deliver improved layout, and these improvements successfully translate to significant performance gains for HDD-based systems that are typically storage bound, and where layout plays a more critical role. The CPU and I/O overhead of data relocation on all-SSD systems occasionally outweighs improvements in layout, motivating autonomic defragmentation during periods of low load. Other media such as QLC flash and SMR are being analyzed. There is insufficient customer data to analyze recompaction at this time.



(A) (B)

Figure 12: (A) Distribution of write chain lengths on customer deployments at different levels of fullness, and (B) Distribution of CPU utilization for systems with write chain length less than 11.

8 Related Work

The original LFS work [32] introduced log structuring and evaluated several policies for performing segment cleaning to constrain the associated overhead. Seltzer, et al. [34] analyze the performance impacts of free space fragmentation in FFS and the overheads associated with cleaning in LFS. It was shown that a policy based on grouping similarly aged blocks into new segments is efficient. Our technique targets areas that have the least cost, and because the emptiest segments (AAs) generally have the oldest blocks, cold blocks become colocated when rewritten out together. The SMaRT file system employs free space defragmentation on SMR drives [13], using either background or on-demand garbage collection based on a set of SMR-specific heuristics.

F2FS is a log-structured file system that is optimized for SSD [21]. Similar to our work, F2FS has the ability to perform both foreground and background segment cleaning and seeks to minimize the impact of cleaning on system performance. Converting random overwrites into sequential write requests in the block device driver can provide the benefits of log-structured writes without paying the cost of segment cleaning [20, 22, 43]. Geriatrix is an aging tool to fragment both files and free space [16]; it reports that free space fragmentation significantly affects file system performance on SSDs. These findings were not on enterprise-grade systems with large storage arrays and do not conflict with our results.

Sequential file read performance degrades as a file's data becomes fragmented [2, 12, 33]. Betrfs is a file system whose format inherently reduces fragmentation; the authors found that performance was sustained better over time than other file systems [5]. Aging can also be partially avoided through preallocating space for a file, multiblock allocations, and delayed allocation [24]. Unfortunately, some degree of aging is inevitable in a log-structured, copy-on-write file system [12]. The DFS file system relocates data in order to reduce fragmentation [2] and improve the subsequent read performance. Recent work has found similar negative effects of file fragmentation on mobile storage and tuned defragmentation for such platforms [11, 15]. A study of deduplication using Windows desktops showed that file fragmentation does not impact performance because a large fraction of their files are not overwritten after creation, and the background defragmenter patches up the fragmented files [26].

The problem of intra-block fragmentation is most commonly solved by using *tail packing*, in which the non-block-size aligned ends of files are persisted to a shared block [5, 31, 37]. The most popular form of this consists of defining some *fragment* size less than the block size, which becomes the minimum unit of allocation [25, 42]. ReconFS [23] dynamically compacts sub-block sized updates to metadata in order to reduce the number of drive writes on flash. Our approach is more general, in that there is no minimum or fixed chunk size. Further, we first compress data so that blocks within large files can also benefit from this technique. Our process of recompaction is similar in concept to garbage collection in the NOVA file system [38], in which log entries in nonvolatile memory are written compactly to a new log when less than 50% of log entries are active.

9 Conclusion

We investigated various forms of fragmentation in the WAFL file system, and showed that it can have significant implications on both performance (as in the cases of free space and file block fragmentation) and storage efficiency (as in the case of intra-block fragmentation). We then presented *storage gardening* techniques that leverage the FlexVol virtualization to counteract each type of fragmentation. Although the techniques dramatically improved data layout across a variety of workloads, performance gains did not universally follow. I/O-bound HDD systems showed significant benefits. However, operational latency on all-SSD storage systems is very sensitive to the availability of CPU cycles, and therefore CPU and I/O overhead of defragmentation may outweigh its benefits. Intra-block defragmentation provided significant storage savings, but with a performance penalty.

10 Acknowledgements

We thank the many WAFL engineers who contributed to these designs over the years; they are too many to list. We thank Pawan Rai for helping gather field data. We thank Keith A. Smith for his input into this work. We also thank our reviewers and shepherd for their invaluable feedback.

References

[1] NetApp cDOT - Volume Move. `https://www.storagefreak.net/2017/07/netapp-cdot-volume-move`.

[2] AHN, W. H., KIM, K., CHOI, Y., AND PARK, D. DFS: A defragmented file system. In *Proceedings. 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. (MASCOTS)* (2002), pp. 71–80.

[3] BARTLETT, W., AND SPAINHOWER, L. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on dependable and secure computing 1*, 1 (2004).

[4] BJØRLING, M., GONZÁLEZ, J., AND BONNET, P. LightNVM: The linux open-channel SSD subsystem. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2017).

[5] CONWAY, A., BAKSHI, A., JIAO, Y., JANNEN, W., ZHAN, Y., YUAN, J., BENDER, M. A., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., ET AL. File systems fated for senescence? nonsense, says science! In *Proceedings of Conference on File and Storage Technologies (FAST)* (2017).

[6] CORBETT, P., ENGLISH, B., GOEL, A., KLEIMAN, T. G. S., LEONG, J., AND SANKAR, S. Row-diagonal parity for double disk failure correction. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2004).

[7] COUNCIL, S. P. Storage performance council-1 benchmark. `www.storageperformance.org`.

[8] CURTIS-MAURY, M., KESAVAN, R., AND BHATTACHARJEE, M. Scalable write allocation in the WAFL file system. In *Proceedings of the Internal Conference on Parallel Processing (ICPP)* (2017).

[9] EDWARDS, J. K., ELLARD, D., EVERHART, C., FAIR, R., HAMILTON, E., KAHN, A., KANEVSKY, A., LENTINI, J., PRAKASH, A., SMITH, K. A., ET AL. FlexVol: flexible, efficient file volume virtualization in WAFL. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2008).

[10] GOEL, A., AND CORBETT, P. RAID triple parity. In *ACM SIGOPS Operating Systems Review* (2012), vol. 46, pp. 41–49.

[11] HAHN, S. S., LEE, S., JI, C., CHANG, L.-P., YEE, I., SHI, L., XUE, C. J., AND KIM, J. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2017).

[12] HE, J., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The unwritten contract of solid state drives. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2017).

[13] HE, W., AND DU, D. H. SMaRT: An approach to shingled magnetic recording translation. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2017).

[14] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *Proceedings of USENIX Winter Technical Conference* (1994).

[15] JI, C., CHANG, L.-P., SHI, L., WU, C., LI, Q., AND XUE, C. J. An empirical study of file-system fragmentation in mobile storage systems. In *Procedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2016).

[16] KADEKODI, S., NAGARAJAN, V., GANGER, G. R., AND GIBSON, G. A. Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2018).

[17] KESAVAN, R., CURTIS-MAURY, M., AND BHATTACHARJEE, M. Efficient search for free blocks in the WAFL file system. In *Proceedings of the Internal Conference on Parallel Processing (ICPP)* (2018).

[18] KESAVAN, R., SINGH, R., GRUSECKI, T., AND PATEL, Y. Algorithms and data structures for efficient free space reclamation in WAFL. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2017).

[19] KESAVAN, R., SINGH, R., GRUSECKI, T., AND PATEL, Y. Efficient free space reclamation in WAFL. *ACM Transactions on Storage (ToS) 13* (October 2017).

[20] KIM, H., SHIN, D., JEONG, Y., AND KIM, K. H. SHRD: Improving spatial locality in flash storage accesses by sequentializing in host and randomizing in device. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2017).

[21] LEE, C., SIM, D., HWANG, J. Y., AND CHO, S. F2FS: A new file system for flash storage. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2015).

[22] LEE, Y., KIM, J.-S., AND MAENG, S. ReSSD: a software layer for resuscitating SSDs from poor small random write performance. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (2010).

[23] LU, Y., SHU, J., WANG, W., ET AL. ReconFS: A reconstructable file system on flash storage. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2014).

[24] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium* (2007).

[25] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for unix. *Transactions on Computer Systems 2*, 3 (1984), 181–197.

[26] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *Proceedings of the 9th USENIX conference on File and storage* (2011).

[27] MICROSYSTEMS, S. ZFS at OpenSolaris Community. `http://opensolaris.org/os/community/zfs/`.

[28] MITTAL, S., AND VETTER, J. S. A survey of software techniques for using non-volatile memories for storage and main memory systems. In *IEEE Transactions on Parallel and Distributed Systems* (Jan 2015).

[29] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the International Conference on Management of Data (SIGMOD)* (1988).

[30] PATTERSON, H., MANLEY, S., FEDERWISCH, M., HITZ, D., KLEIMAN, S., AND OWARA, S. SnapMirror: File-system-based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), USENIX Association.

[31] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (2013), 9.

[32] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10* (1992), 1–15.

[33] SATO, T. ext4 online defragmentation. In *Proceedings of the Linux Symposium* (2007), vol. 2, pp. 179–86.

[34] SELTZER, M., SMITH, K. A., CHANG, H. B. J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (1995).

[35] SMITH, K. A., AND SELTZER, M. I. File system aging—increasing the relevance of file system benchmarks. In *ACM SIGMETRICS Performance Evaluation Review* (1997), vol. 25, ACM, pp. 203–213.

[36] SUNDARAM, R. The Private Lives of Disk Drives. https://atg.netapp.com/?p=13640, 2006.

[37] WIKIPEDIA. Reiserfs. Wikipedia, the free encyclopedia, 2017. (Online; accessed 18-April-2018).

[38] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2016).

[39] XU, Q., SIYAMWALA, H., GHOSH, M., AWASTHI, M., SURI, T., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance characterization of hyperscale applicationson on nvme ssds. In *ACM SIGMETRICS Performance Evaluation Review* (2015), vol. 43, ACM.

[40] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR)* (2015), ACM.

[41] YANG, J., PLASSON, N., GILLIS, G., TALAGALA, N., AND SUNDARARAMAN, S. Don't stack your log on my log. In *INFLOW* (2014).

[42] ZHANG, Z., AND GHOSE, K. yFS: A journaling file system design for handling large data sets with reduced seeking. In *Proceedings of Conference on File and Storage Technologies (FAST)* (2003).

[43] ZUCK, A., KISHON, O., AND TOLEDO, S. LSDM: Improving the performance of mobile storage with a log-structured address remapping device driver. In *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on* (2014).