Anti-Aging LFS: Self-Defragmentation With Fragmentation-Aware Cleaning

JONGGYU PARK AND YOUNG IK EOM

1Department of Computer Science and Engineering, Sungkyunkwan University, Suwon 16419, South Korea
2Department of Electrical and Computer Engineering, College of Computing, Sungkyunkwan University, Suwon 16419, South Korea

Corresponding author: Young Ik Eom (yieom@skku.edu)

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (IITP-2015-0-00284, (SW Starlab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2017R1A2B3004660).

ABSTRACT For several decades, filesystem aging has been widely studied, but nonetheless, it still remains an unsolved problem. Among various filesystems, log-structured filesystems have been reported to be vulnerable to fragmentation due to their append-only write policy. Fragmentation hinders various I/O activities such as sequential read and trim operations, regardless of the underlying storage types. This paper extensively analyzes the rationale behind performance degradation incurred by fragmentation on various types of storage devices. To eliminate fragmentation without additional I/O overhead, we propose an anti-aging log-structured filesystem, called AALFS. During segment cleaning, AALFS re-arranges the order of valid blocks based on inode number and file offset to eliminate existing fragmentation. To enhance the efficacy of the re-ordering process, the new victim selection policy of AALFS reflects the fragmentation degree of each segment in the selection of victim segments. Our experimental results show that AALFS effectively eliminates fragmentation by up to 99.8% and significantly improves sequential read performance on various types of storage devices. Particularly, AALFS improves the sequential read throughput of IOzone on hard disk drives by up to 22.8 times.

INDEX TERMS Fragmentation, log-structured filesystems, operating systems, storage.

I. INTRODUCTION Filesystem aging, which is also called fragmentation, is a phenomenon wherein contents of files are scattered throughout a filesystem instead of being located in one contiguous location. Unfortunately, filesystem aging still remains an unsolved problem despite the fact that it has been studied over the past several decades [1]–[9]. Files become fragmented while performing repetitive modification operations, such as write, delete, and truncate. Although its root causes differ depending on the type of filesystem, previous studies have asserted that most filesystems eventually fragment over time [6], [7].

Among various filesystems, log-structured filesystems (LFSs) [10]–[12] have been reported to be extremely vulnerable to fragmentation on account of their append-only write policy [6], [7]. LFS regards the storage space as a sequential log and allocates new blocks according to the order of writes.

Since LFS does not allow in-place updates, it handles update operations by invalidating the original data and writing the updated data in a new area. Accordingly, the newly updated data are located apart from the other data that are not updated in the file, resulting in fragmentation. Additionally, simultaneous write operations by multiple processes can generate fragmentation, because LFS alternately allocates blocks to the processes, and thus, each process acquires striped blocks.

Log-structured filesystems inherently perform a segment cleaning process, which consolidates valid blocks in some segments to secure free space [11]–[14]. The cleaning process consists of two phases. First, LFS chooses a victim segment, which will be cleaned, according to its victim selection policy. Second, it checks the validity of each block in the victim segment and migrates valid blocks into a new segment. A cleaning process is an opportune chance to relocate blocks in an appropriate way, which can possibly eliminate fragmentation. Nonetheless, the conventional cleaning process migrates valid blocks in the original block order, without trying to eliminate fragmentation.
Filesystem aging degrades I/O performance regardless of the underlying storage type. Previous studies [4], [5], [7] demonstrated that hard disk drives (HDD) suffer from fragmentation due to their seek time. However, although modern storage devices such as solid-state drives (SSD) do not involve seek time, fragmentation can also degrade their performance due to the fact that it increases the number of I/O operations and their randomness. In the current Linux kernel, a single I/O request can represent only a sequence of contiguous filesystem blocks, and thus fragmentation splits a single large sequential I/O into multiple small random ones.

Research on filesystem aging [15]–[21] mainly consists of two directions; prevention and elimination. Prevention of aging [15], [16], [21] allocates contiguous space to files as much as possible on the time of creation using pre-allocation, delayed allocation, etc. However, previous studies [6], [7] insist that the prevention of filesystem aging eventually fails over time. The other side of the research [17]–[20] focuses on eliminating existing fragmentation. Most solutions [17]–[19] rely on migrating fragmented files into a new contiguous area. However, such migration-based methods induce additional write I/O operations into the storage device. Since modern storage technologies such as flash memory have a limited number of program/erase cycles, the additional write operations significantly curtail their lifetime.

In this paper, we show how vulnerable LFS is to fragmentation, compared with the in-place update filesystem, and also measure the performance degradation incurred by filesystem aging with various types of storage devices. In addition, since performance degradation exhibits a different tendency depending on the underlying storage type, we investigate in detail the extent in which each storage device suffers from fragmentation and the rationale behind this fragmentation.

To minimize filesystem aging without incurring additional write operations, we propose an anti-aging log-structured filesystem, called AALFS. AALFS endows the cleaning process with fragmentation-awareness by implementing two schemes; Defrag cleaning and Frag-aware victim selection policy. Defrag cleaning temporarily maintains metadata of valid blocks and sorts them by inode number and file offset. All blocks from the same file have an identical and unique inode number, and thus block re-ordering based on the inode number guarantees that blocks belonging to the same file will be located in a contiguous manner. Additionally, AALFS includes file offsets in the re-ordering criteria since the sequentiality of blocks in terms of file offset also determines the degree of fragmentation. To overcome the limitation that such a solution can eliminate fragmentation only within a segment, our Defrag cleaning keeps valid blocks across multiple segments in a single valid block queue and reorders all of them at once. Thus, using Defrag cleaning, AALFS can eliminate the existing fragmentation on multiple segments.

We also introduce a new victim selection policy, called Frag-aware victim selection, in order to improve the efficacy of Defrag cleaning. Our new policy calculates the fragmentation degree of each segment by comparing the inode numbers and the file offsets of two consecutive blocks. For example, if the inode number of a particular block is different from that of the previous block, AALFS considers them as fragmented since they belong to different files, and thus increments the fragmentation degree. Additionally, even if two consecutive blocks belong to the same file, fragmentation can occur when their file offsets are not sequential. Therefore, AALFS detects such cases by comparing file offsets, and then increments the fragmentation degree. AALFS utilizes the fragmentation degree of segments in the selection of victim segments, thereby increasing the probability of fragmented segments being chosen as a victim segment.

To verify the efficacy of AALFS, we evaluate our scheme with the IOzone benchmark and the SQLite database on various types of storage devices; HDD, MicroSD, flash SSD, and Optane SSD. We demonstrate in our experiments that AALFS significantly eliminates existing fragmentation using Defrag cleaning, which leads to dramatic performance improvement. Particularly, the sequential read performance of IOzone on HDD has been improved by 22.8 times after performing Defrag cleaning. The rationale behind the performance enhancement differs depending on the internals of the storage devices. Therefore, we also clarify the fundamental reasons for the performance improvements in detail, considering the internals of the storage devices.

The rest of this paper is organized as follows. Section II explains the fragmentation phenomenon and elaborates on its detrimental impact on I/O behaviors. Section III demonstrates that LFS is vulnerable to fragmentation and its conventional cleaning cannot eliminate existing fragmentation. Section IV proposes an anti-aging log-structured filesystem, called AALFS, and experimental results are presented in Section V. Section VI discusses the related work, and we conclude this paper in Section VII.

II. AN EXTENSIVE ANALYSIS ON FRAGMENTATION

A. FILESYSTEM FRAGMENTATION

Filesystem aging, or fragmentation, occurs when filesystems cannot allocate contiguous blocks to files, and accordingly, contents of files dis-contiguously lie on the filesystem layer. The rationale behind fragmentation includes repetitive modification operations and simultaneous file creations. Fragmentation has been reported to degrade I/O performance on HDDs because of its seek time [4], [5], [7], and the oversimplified myth that defragmentation is not necessary on SSDs has been pervasive [20]. In this paper, we demonstrate that fragmentation can degrade I/O performance regardless of the underlying storage type including SSD storage devices.

B. KERNEL OVERHEADS ON FRAGMENTATION

Fragmentation hinders I/O operations because of both the kernel and hardware-specific reasons. First, the current Linux kernel handles I/O operations using the bio and request structures. However, both of them can represent only contiguous
filesystem blocks. Therefore, even if a process issues a single large I/O to a fragmented file, it is split into multiple small ones, thereby increasing the number of I/O operations. Accordingly, the kernel overheads for creating such structures and managing I/O requests will be increased. Furthermore, fragmentation transforms sequential I/O operations into random ones because the corresponding blocks are sparsely and randomly located in fragmented filesystems.

Fig. 1 illustrates an example of kernel overheads induced by fragmentation. On the filesystem layer of this figure, each letter stands for a filename, and each numerical value stands for a file offset. Process 1 reads four filesystem blocks of a non-fragmented file. Since the blocks are sequential and contiguous, a single bio, request, and command can convey the I/O information. On the other hand, the single read system call from Process 2 is split into four different I/O requests due to fragmentation. Therefore, four bios, requests, and commands need to be created, thereby increasing the kernel overheads. Moreover, as shown in the case of File B in Fig. 1, when the blocks of a file do not exist in the file offset order, the I/O randomness increases, which is harmful to most storage devices that present poor random I/O performance.

**FIGURE 1. The kernel overheads induced by fragmentation.**

**C. HARDWARE-SPECIFIC OVERHEADS ON FRAGMENTATION**

Hardware-specific reasons on the overheads of fragmentation vary depending on how the storage device retrieves and stores data. A HDD moves its head to find the physical location of data whenever it accesses data on the disk. This time, elapsed to move the head, is referred to as seek time. Fragmentation increases the seek time because physical data are widely dispersed, resulting in poor spatial locality. Therefore, HDDs experience performance degradation with fragmentation.

Flash-based storage maintains a mapping table between logical and physical block addresses because flash memory does not permit in-place updates inherently. Due to the limited size of DRAM inside the storage, demand-based map caching is often adopted [8], [22], [23]. Demand-based map caching pre-fetches mapping entries of contiguous logical pages to exploit spatial locality. However, as mentioned before, fragmentation decreases spatial locality, and thus diminishes the efficacy of the map caching [8].

Additionally, flash-based storage consists of multiple planes and channels which can be accessed in parallel [24]–[26]. Utilizing this parallelism, the storage devices pre-fetch subsequent logical pages from the flash chips to DRAM in advance, for the sake of sequential read performance. Read requests to the pre-fetched data are serviced by DRAM, without accessing the corresponding flash chips. Therefore, sequential read performance is much higher than random read performance. However, fragmentation distorts pre-fetching because it increases the randomness of I/O operations. For this reason, sequential read performance can be remarkably decreased with fragmentation.

Moreover, some MicroSD and eMMC devices do not support command queuing, and the devices can accept only a single command at a time [27]. Therefore, an increase in the number of I/O operations due to fragmentation serializes I/O processing, thereby significantly degrading I/O performance.

Finally, in the case of flash-based storage devices, the host issues trim commands which indicate that certain pages are invalid, in order to minimize garbage collection overheads. However, similarly to the read/write commands, a single trim command can represent only logically contiguous pages. Therefore, fragmentation increases the number of trim commands for the same amount of invalid data.

The state-of-the-art storage technology, Optane SSD, behaves a bit differently from flash SSD. Wu [28] explored the fact that Optane SSDs adopt LBA-based mapping while flash SSDs employ written-order-based mapping. Additionally, there is no pre-fetching for sequential reads in Optane SSDs. Therefore, one might assume that the fragmentation effect on Optane SSDs is minor, compared with other types of storage devices. However, Optane SSDs usually have a smaller number of parallel units than flash SSDs, and therefore, an increase in the number of I/O commands induced by fragmentation can incur resource conflicts inside Optane SSDs. Additionally, the high performance of Optane SSDs reveals the kernel overhead which is imperceptible on other storage devices.

**D. EXPERIMENTAL ANALYSIS ON PERFORMANCE DEGRADATION BY FRAGMENTATION**

To quantitatively measure the performance degradation caused by fragmentation, we create a file that is fragmented into 4KB and then sequentially read the entire file. Fig. 2 shows the normalized sequential read performance compared to that of a non-fragmented file (baseline) for each storage type. First, to measure the kernel overheads, we conducted the experiment on Ramdisk. As shown in Fig. 2, Ramdisk shows 68.7% of the baseline, even though no storage overhead exists. Fragmentation inevitably increases the number of I/O operations, and accordingly, the kernel spends more time generating and managing such I/O operations.
In the case of HDD, the sequential read performance is only 22.6% of the baseline because of the seek time inside the storage. MicroSD also severely suffers from fragmentation because the device can accept only a single command at a time and an increase in the number of commands arouses extreme queue contention. Flash SSD is comparatively less affected by fragmentation because its abundant parallel units minimize resource contention. However, the performance still decreases since the randomness of fragmented I/O requests deactivates the internal pre-fetching. Flash SSD seems more resistant to fragmentation than Ramdisk when it comes to sequential read performance. This is because the performance bottleneck lies in the storage and thus the kernel overheads are concealed.

One might assume that fast storage is immune to fragmentation. However, the Optane SSD shows 57.4% of the baseline. As previously mentioned, Optane SSDs have a smaller number of parallel units than Flash SSDs, thereby resulting in increased resource contention. Besides, the increased kernel overheads induced by fragmentation are conspicuous on Optane SSDs because the kernel I/O stack accounts for a large fraction of I/O time.

III. LOG-STRUCTURED FILESYSTEM AND SEGMENT CLEANING

A. LOG-STRUCTURED FILESYSTEM

Log-structured filesystems (LFSs) have been extensively studied for the past decade as flash technology has become mature. LFS gracefully overcomes the drawbacks of flash memory, such as garbage collection (GC), by adopting the append-only write policy. As depicted in Fig. 3, LFS divides storage space into segments and sequentially allocates new blocks from the segments. In other words, data and metadata are sequentially stored according to the written order, rather than the file offset order. Therefore, even if a process issues random writes, LFSs transform them into sequential ones, thereby increasing write performance while decreasing garbage collection overheads inside the flash-based storage.

However, the append-only write policy can incur fragmentation in several situations. First, when multiple processes simultaneously issue synchronous or O_DIRECT write operations, LFS alternately allocates new blocks to the processes, and thus each process possesses striped blocks, resulting in fragmentation. Second, LFS handles update operations by allocating new blocks to the updated data and invalidating the existing ones. Accordingly, when a process performs random updates, the corresponding file will be fragmented.

To show that LFS is vulnerable to fragmentation for the aforementioned reasons, we first performed two experiments, executing multiple file creation and random update workloads, and compared the number of non-contiguous blocks with that on the Ext4 filesystem which allows in-place updates. The detailed experimental setup is described in Section V-A. To represent LFS, we utilized the F2FS filesystem in all experiments. For the multiple file creation workload, we launched eight processes, each of which creates a 100MB file with sequential 4KB requests, where each of item is followed by a \texttt{fsync} system call. Afterward, we calculated the average number of fragments, each of which is a sequence of contiguous blocks. The Ext4 filesystem pre-allocates multiple blocks for each file in order to minimize interference among multiple concurrent write operations [16]. Therefore, even when multiple files are created at the same time, the files can have contiguous blocks. As shown in Fig. 4, the files on Ext4 have 20 fragments on average. On the other hand, the files on F2FS are highly fragmented, having 25,598 fragments on average. Under the specified workload, F2FS allocates a single 4KB block to each file at a time due to its append-only write policy.

For the random update workload, we first create a 100MB file that consists of a single large fragment and thereafter randomly update half of the file with 4KB requests,
each of which is followed by a `fsync` system call. As shown in Fig. 4, the file on Ext4 still consists of a single fragment since Ext4 performs in-place updates. In contrast, F2FS handles update operations by invalidating the existing blocks and allocating new blocks to the updated data. Accordingly, the updated blocks reside in the written order, rather than in the file offset order, and become separated from the original blocks. Consequently, the files on F2FS comprise 25,430 fragments on average, resulting in extreme fragmentation.

Finally, to further strengthen our motivation, we also performed an experiment with the SQLite benchmark while creating dummy files with `O_DIRECT` in the background. In this experiment, we create a 400MB database file with SQLite sequential insert operations with sync mode. As shown in Fig. 5, Ext4 detects the sequential pattern of the SQLite benchmark and pre-allocates multiple blocks for the benchmark [16]. As a result, the fragment count of the database file on Ext4 is only 53. On the other hand, F2FS sequentially allocates new blocks according to the written order. Accordingly, the database file on F2FS becomes highly fragmented, as comprising 102,344 fragments. Therefore, the append-only policy jeopardizes LFS under the workloads that have simultaneous writes from multiple processes or random updates, with regard to fragmentation.

**B. SEGMENT CLEANING IN LFS**

As mentioned before, updating data on LFS does not overwrite existing data, and instead allocate new blocks for the updated data while invalidating the existing ones. To reuse the invalidated blocks, LFS consolidates valid blocks in a segment and migrates them to another segment. We refer to this as a cleaning process or segment cleaning. The cleaning process consists of two phases. First, LFS calculates the cleaning cost of each segment and finds the best segment for cleaning according to the victim selection policy. There are two representative victim selection policies: greedy and cost-benefit. The greedy policy chooses a segment that has the smallest number of valid blocks to minimize the migration cost. On the other hand, the cost-benefit policy considers not only the number of valid blocks but also the age of the segment, under the assumption that recently written data will be likely to be updated again within a short period.

After selecting a victim segment, LFS migrates valid blocks in the segment to a new segment. We describe the migration process in detail below and illustrate the process in Fig. 6.

1) Check the validity of each block in the victim segment
2) Migrate the valid block to a new segment
3) Repeat (1)-(2) for all blocks in the victim segment
4) If all valid blocks are migrated, the victim segment is marked as a free segment

Likewise, segment cleaning of LFS migrates valid blocks to another location, which is an opportune chance to relocate the order of file data blocks in an appropriate way. However, conventional segment cleaning maintains the original block order, thereby leaving the existing fragmentation untouched. In Fig. 4 and Fig. 5, F2FS_AC denotes the case after performing the cleaning process. Under all of the workloads, the conventional cleaning process can hardly curtail the number of fragments since it maintains the original block order during migration.

**IV. AALFS: ANTI-AGING LOG-STRUCTURED FILESYSTEM**

Heretofore, this paper has addressed the problem of filesystem fragmentation from the performance perspective and investigated the rationale behind the performance degradation with various types of storage devices. Additionally, this paper has demonstrated that LFS is vulnerable to fragmentation, showing that it becomes highly fragmented under certain workloads. Furthermore, the conventional cleaning process cannot eliminate existing fragmentation since it discounts the existence of fragmentation and migrates filesystem blocks without any consideration on the block sequence.

In this section, we introduce an anti-aging log-structured filesystem (AALFS), which performs defragmentation during the cleaning process without increasing the write traffic. Our contribution consists of two parts. First, we propose Defrag cleaning which re-orders the valid blocks based on inode number and file offset across multiple segments. Second, to maximize the efficacy of Defrag cleaning, we suggest a new victim selection policy, Frag-aware victim selection, which considers fragmentation degree of each segment as well as the number of valid blocks and their ages when selecting a victim segment. For better understanding, we describe the flow chart of AALFS in Fig. 7, with detailed explanations in Section IV-A and IV-B.
we propose Defrag cleaning existing fragmentation as it is. To solve this problem, space. The conventional cleaning process migrates valid LFS inherently executes a cleaning process to secure free data structures, vb_mt the same file based on their file offsets. additionally re-orders the valid blocks belonging to cleaning Defrag not sequentially lie in terms of file offset. Therefore, contiguous, I/O operations can still be divided if the blocks do exist before *b, and a positive value if *a should exist after *b. In the re-ordering process, AALFS first compares the inode numbers to contiguously locate valid blocks belonging to AALFS creates a vb_mt structure in the sorted order, so as to migrate the valid blocks to the new segments.

To implement Defrag cleaning, AALFS employs two new data structures, vb_mt and defrag queue. The vb_mt structure contains metadata of valid blocks such as inode numbers, file offsets, and other write-related information that is needed to perform the subsequent migration, including the addresses of valid blocks. The vb_mt structure consists of the metadata necessary to perform re-ordering and migration so that the spatial overhead can be minimized, as compared with keeping the actual data of the victim blocks. The defrag queue structure is utilized to keep vb_mt of valid blocks across multiple segments. After vb_mt of all valid blocks are stored in the defrag queue, AALFS re-orders the defrag queue utilizing inode numbers and file offsets stored in vb_mt.

For re-ordering, AALFS employs the kernel list sort API which performs merge sort. Algorithm 1 presents the comparison function of the kernel list sort API. In Algorithm 1, the comparison function returns a negative value if *a should exist before *b, and a positive value if *a should exist after *b. In the re-ordering process, AALFS first compares the inode numbers to contiguously locate valid blocks belonging to the same file. If two blocks are from the same file, AALFS compares their file offsets to rearrange the blocks in the offset order. Consequently, all the valid blocks belonging to the same file become located contiguously and sequentially in terms of file offset. Therefore, subsequent I/O requests for such blocks are no longer fragmented. After the re-ordering process, AALFS generates write operations using the write-related information of the vb_mt structure in the sorted order, so as to migrate the valid blocks to the new segments.

The overview of Defrag cleaning is presented in Fig. 8. First, AALFS checks the validity of filesystem blocks in the victim segments. If a block is determined to be valid, AALFS creates a vb_mt structure for the block, collecting its inode number, file offset, etc., and inserts the vb_mt structure into the defrag queue. AALFS repeats this process until all the valid blocks in the victim segments have been checked. Afterward, the defrag queue is sorted based on inode number and file offset to eliminate the existing fragmentation. Finally, AALFS migrates the valid blocks in the sorted order to new segments and sets the victim segments as free segments. Since the defrag queue keeps only the metadata of valid blocks to minimize spatial overheads, the actual data movement occurs not from the defrag queue, but from the original locations of the valid blocks. As a result, the fragmented valid blocks are defragmented, and thus, subsequent I/O requests to those blocks will not be divided, resulting in performance improvement. Note that, since the defrag queue covers multiple segments, AALFS can eliminate a large amount of fragmentation, depending on the number of segments the filesystem cleans at a time. The time complexity of Defrag cleaning is O(s · b), where s is the number of segments to clean at a time and b is the number of blocks in a single segment. In this paper, the number of blocks (b) in a segment is equal to 512, assuming that the segment size is 2MB and the block size is 4KB.

Algorithm 1 Comparison Function for Sorting Valid Blocks

| Input: | struct vb_mt *a, struct vb_mt *b |
| Output: | -1 if *a should exist before *b |
|        | 1 if *a should exist after *b |

Function comp_func (struct vb_mt *a, struct vb_mt *b):

if a reigning_num > b reigning_num
    return 1;
end
if a reigning_num == b reigning_num
    if a file_offset > b file_offset
        return 1;
    end
end
return -1;

End Function

B. FRAG-AWARE VICTIM SELECTION POLICY IN AALFS

Existing victim selection policies cannot maximize the defragmentation effect from Defrag cleaning because they...
do not consider the existence of fragmentation. For example, suppose that two segments have an identical number of valid blocks while one is highly fragmented and the other is not. Although the fragmented one can be defragmented by Defrag cleaning, the existing victim selection policies may choose the non-fragmented segment because they discount the existence of fragmentation. For this reason, we also suggest the Frag-aware victim selection policy to exploit Defrag cleaning of AALFS.

In contrast to other policies, our victim selection policy calculates how much a segment is fragmented and utilizes it in determining the victim segment. To quantify the amount of fragmentation, we introduce a new metric, called a fragmentation degree, which denotes the number of fragmented valid blocks. Algorithm 2 describes how to calculate the fragmentation degree of a segment in detail. In the Frag-aware victim selection policy, several additional variables and a structure are introduced to calculate the fragmentation degree as shown below.

- **frag_degree**: the fragmentation degree of the segment
- **cur_block**: the current block in the segment
- **cur_inode**: the inode number of the current valid block
- **prev_inode**: the inode number of the previous valid block
- **cur_offset**: the file offset of the current valid block
- **prev_offset**: the file offset of the previous valid block
- **ino_tree**: the set of all the inode numbers of which valid blocks have been checked

First, AALFS checks the validity of the blocks in a segment. If a block is determined to be valid, AALFS compares **cur_inode** with **prev_inode** to check whether the current block and the previous block belong to the same file or not. If **cur_inode** and **prev_inode** are different, AALFS additionally checks whether the current inode number exists in **ino_tree**. When the inode number does not exist in **ino_tree**, AALFS inserts the inode number into the **ino_tree** (#1). In this case, AALFS judges that the block is not fragmented because hitherto no other blocks belonging to the same file exist in the segment. If the inode number exists in **ino_tree**, it means that there are fragmented blocks belonging to the same file, and they can be defragmented during Defrag cleaning of AALFS. In this case, AALFS increments **frag_degree** (#2).

---

**Algorithm 2 Calculating the Frag. Degree of a Segment**

Output: **frag_degree**

Function `get_frag_degree(victim segment)`:

```plaintext
for each block in a segment do
    cur_inode ← the inode number of the current block;;
    cur_offset ← the file offset of the current block;;
    if cur_block is valid then
        if cur_inode ≠ prev_inode then
            if cur_inode is not in ino_tree then
                insert cur_inode into ino_tree; > (#1);
            else
                frag_degree++;
            end
        else if cur_offset ≠ next(prev_offset) then
            frag_degree++;
        end
    prev_inode ← cur_inode;
    prev_offset ← cur_offset;
end
```

End Function
If \( \text{cur_inode} \) and \( \text{prev_inode} \) are identical, AALFS compares \( \text{cur_offset} \) with \( \text{prev_offset} \) to inspect their sequentiality. Even if the two blocks belong to the same file, fragmentation occurs when their file offsets are discontiguous. Therefore, if \( \text{cur_offset} \) is not the subsequent offset of the \( \text{prev_offset} \), AALFS increments \( \text{frag_degree} \) (#3). Afterward, to check the next block, AALFS modifies \( \text{prev_inode} \) and \( \text{prev_offset} \) to the current ones (#4) and moves on to the next block. This process repeats until all the blocks in the segment have been checked.

Calculating the fragmentation degree of each segment induces additional computations, which might prolong the duration of the cleaning process. To minimize the computational overheads and increase the efficiency of our scheme, AALFS calculates the fragmentation degrees for only a subset of the segments, as described in Algorithm 3. First, AALFS selects several victim candidates according to the existing cost-benefit algorithm which considers the valid block counts and the modification time. Second, the fragmentation degrees of the candidate segments are calculated using \( \text{get_frag_degree}() \). Finally, AALFS designates the most fragmented segment (highest \( \text{frag_degree} \)) out of the candidate segments as a victim segment. The number of candidates can be configured, depending on the importance of defragmentation on the system. Note that, similarly to Defrag cleaning, the time complexity of Frag-aware victim selection is also \( \mathcal{O}(c \cdot b) \) where \( c \) is the number of candidate segments and \( b \) (512 in this paper) is the number of blocks in a segment.

### Algorithm 3 Selecting a Victim Segment

**Output**: victim segment

**Function** `select_victim(victim candidates)`: 

```plaintext
for seg in all victim candidates do 
    frag_degree = get_frag_degree(seg); 
    if max < frag_degree then 
        victim = seg; 
        max = frag_degree; 
    end 
end 
return victim; 
```

#### V. EVALUATION

##### A. EXPERIMENTAL SETUP

To implement AALFS, we have modified the cleaning process and the victim selection policy of the F2FS filesystem. Specifically, for the `defrag queue` structure of Defrag cleaning, we utilized the kernel doubly linked list and its sort API. When the filesystem migrates valid blocks to another segment in `move_data_page()` of F2FS, AALFS instead gathers the metadata of the valid blocks and inserts them into `defrag queue`. After the metadata of all the valid blocks in victim segments are inserted, AALFS calls `list_sort()` to rearrange the blocks and prepares to migrate them to a new segment. To implement Frag-aware victim selection, we modified `get_victim_by_default()` of F2FS to select multiple candidate segments and calculate their fragmentation degrees. For `ino_tree` in our algorithm, we employed the RB-tree data structure provided by the Linux kernel, which guarantees high performance in entry searching, because our algorithm performs a small number of insert operations and many search operations.

To comprehensively evaluate AALFS, we ran the IOzone benchmark and the SQLite database on a system equipped with various types of storage devices, as described in Table 1. In our experiments, we disabled the in-place update function and the asynchronous cleaning of F2FS in order to represent a general log-structured filesystem.

| Experimental setup. |
|----------------------|
| **CPU** | Intel Xeon E5-2620 v4 2.1GHz 8 core |
| **Memory** | DDR4 2133MHz 32GB |
| **Storage** | HDD 7200RPM 1TB |
| MicroSD card A1 128GB |
| SATA Flash SSD 256GB |
| Optane SSD 960GB |
| **OS** | Ubuntu 14.04 LTS with Linux Kernel 4.4.0 |

#### B. IOzone EXPERIMENTS

To measure the performance improvement of sequential read operations on AALFS, we initially ran 32 processes, each of which created a single file with 4KB O_DIRECT sequential writes, so that every file is fragmented into 4KB fragments. Afterward, we deleted some of the files to invalidate some filesystem blocks and manually performed the cleaning process. Fig. 9 shows the sequential read throughput when reading one of the remaining files. The x-axis denotes the request size, and the legends show the number of segments to clean at a time. In the legends, ‘Ori’ shows the original performance before performing Defrag cleaning. Note that since we perform sequential reads, the kernel spontaneously reads additional consecutive blocks (read-ahead), and therefore, the read-ahead size is adjusted to the request size to precisely evaluate our scheme.

1) HARD DISK DRIVES

Fig. 9a shows the sequential read performance on the HDD. The original performance before cleaning shows only 5.645 MB/s regardless of the request size. Due to fragmentation, every request is split into multiple 4KB random requests, and thus, the seek time overhead inside the HDD is exacerbated. Defrag cleaning of AALFS improves the performance by up to 22.8 times, compared with that before cleaning. This result comes from the fact that AALFS significantly eliminates the existing fragmentation. As shown in Table 2, AALFS decreases the number of fragments from 25,604 to 48 on the HDD when AALFS cleans 32 segments at one time. Due to the defragmentation effect of AALFS,
sequential I/O requests are not split, and accordingly, the seek time overhead is mitigated, contributing to significant performance improvement. Especially, on the request size of 4KB in which no requests become fragmented even without cleaning, the sequential read performance is improved by around 22 times after cleaning. This result implies that the performance bottleneck on the HDD lies in the seek time, and AALFS effectively cures it.

2) MicroSD
Fig. 9b shows the sequential read performance on the MicroSD. AALFS shows 7.12 times higher throughput than the original when the request size is 128KB. The MicroSD device that we utilized in the experiment is of type A1 which accepts a single command at a time. Therefore, the increased number of I/O requests results in significant resource (including queue) contention inside the storage, thereby degrading the performance. Meanwhile, all the fragments of the files are larger in size than 128KB in the cases when AALFS cleans more than 2 segments at a time. Therefore, none of the requests become fragmented in those cases. However, the performance of the 32 segment case is around 1.21 times higher than that of the 4 segment case, when the request size is 128KB. We assume the result comes from the fact that most MicroSD devices hold partial mapping entries on DRAM instead of keeping the entire mapping table, due to deficient DRAM space. Therefore, accessing data whose mapping entries are not on DRAM requires additional movement of the corresponding mapping entries from the flash chip to DRAM, whereby the performance becomes degraded. The number of fragments when AALFS cleans 4 segments at a time is almost eight times as many as that in the case of 32 segments. Therefore, the case of 32 segments shows higher spatial locality, which leads to higher efficiency of map caching.

3) FLASH SSD
As shown in Fig. 9c, AALFS outperforms the original by up to 2.19 times when the request size is 8KB. The performance degradation on the SSD mostly originates from the fact that I/O randomness incurred by fragmentation disables pre-fetching inside the SSD. As mentioned before, the SSD consists of multiple parallel units, whereby the SSD loads consecutive logical pages to its DRAM in advance. Therefore, the more AALFS defragments, the higher the efficiency of pre-fetching becomes. As a result, as AALFS cleans more segments at a time, higher performance can be achieved due to the increased locality. Additionally, we deleted a file and issued trim commands for the deleted file to the SSD while measuring the elapsed time (Trim Cost). As mentioned before, a single trim command can express logically contiguous pages. Therefore, the number of fragments determines the sequence of cleaning.
performance of trim operations. As shown in Fig. 10, AALFS dramatically reduces the trim cost by up to 99.7% in the case of 32 segments, since the number of trim commands for the deleted file is significantly decreased after Defrag cleaning.

4) OPTANE SSD

Fig. 9d shows the sequential read performance on the Optane SSD. AALFS reduces the number of fragments from 262,148 to 516 through Defrag cleaning, as shown in Table 2. Consequently, AALFS shows around 2.1 times higher throughput than the original when the request size equals to 128KB. In contrast to flash-based storage, the Optane SSD maintains the entire mapping table on DRAM and does not perform pre-fetching. Therefore, the only influence by fragmentation is the number of I/O requests, rather than their sequentiality. As a result, the performance difference among the cases except for the original case remains negligible.

C. SQLite EXPERIMENTS

To verify the efficacy of AALFS, we created a database file using SQLite insert operations with sync mode, while creating multiple dummy files with O_DIRECT in the background. Afterward, we deleted some dummy files to invalidate filesystem blocks and manually conducted the cleaning process. Finally, we issued a select query and measured the time to process the query. Fig. 11 shows the select cost (s/GB) against the database file. The x-axis denotes the request size, and the legends show the number of segments to clean at a time. As in the IOzone experiments, ‘Ori’ shows the original performance before cleaning becomes larger as the request size increases, as shown in Fig. 11d. For example, when the request size is 4KB, the fragmented requests are serviced at the same time by means of the abundant parallel units inside the SSD. The fragmentation overhead becomes inconsequential, which in turn brings less performance gain by defragmentation.

Regarding the trim cost, we deleted the database file and issued trim commands to the SSD while measuring the elapsed time (Trim Cost). As shown in Fig. 12, AALFS dramatically reduces the trim cost from 59.60 seconds to 0.28 seconds due to defragmentation by Defrag cleaning.

4) OPTANE SSD

Contrary to the flash SSD, in the Optane SSD, the performance gap between the original and AALFS after cleaning becomes larger as the request size increases, as shown in Fig. 11d. For example, when the request size is 4KB, the Optane SSD could not achieve a performance gain from defragmentation, while the performance gain increases as the request size becomes larger. The small number of parallel units leads to resource contention inside the Optane SSD when the request size gets larger. As a result, AALFS curtails the select cost by up to 37.7% when the request size is 128KB, compared with that of the original.

D. FRAG-AWARE VICTIM SELECTION POLICY

To evaluate our victim selection policy, we compared it with the cost-benefit policy on the MicroSD. In this experiment, as described in Table 4, we concurrently created various sizes of files with different request sizes. In this experiment, the smaller the request size is, the more fragmented the file becomes. When we created the files, we utilized O_DIRECT write operations of IOzone while creating dump files for fragmentation. Afterward, we deleted the dump files and manually performed Defrag cleaning. To observe the performance difference between the two victim selection policies, we conducted only a limited number of cleaning processes. In our experiment, we set the number of victim candidates to

2) MicroSD

AALFS shows 80.7% lower select cost than the original when the request size is 128KB on the MicroSD. Similar to the IOzone experiments, AALFS reduces the number of fragments from 10,098 to 201 via Defrag cleaning in the case of 32 segments. Therefore, AALFS mitigates the resource (including queue) contention inside the storage, resulting in performance improvement.

3) FLASH SSD

As shown in Fig. 11c, AALFS shows only 35.7% of the select cost compared with that of the original, when the request size equals to 4KB on the Flash SSD. Unlike other storage devices, the performance gap between the original and AALFS after cleaning lessens as the request size increases. Since the SSD supports command queueing, it accepts multiple commands and services them in parallel if possible. Accordingly, when the request size is 128KB, the fragmented requests are serviced at the same time by means of the SSD. The fragmentation overhead becomes inconsequential, which in turn brings less performance gain by defragmentation.

2) MicroSD

AALFS shows 80.7% lower select cost than the original when the request size is 128KB on the MicroSD. Similar to the IOzone experiments, AALFS reduces the number of fragments from 10,098 to 201 via Defrag cleaning in the case of 32 segments. Therefore, AALFS mitigates the resource (including queue) contention inside the storage, resulting in performance improvement.

3) FLASH SSD

As shown in Fig. 11c, AALFS shows only 35.7% of the select cost compared with that of the original, when the request size equals to 4KB on the Flash SSD. Unlike other storage devices, the performance gap between the original and AALFS after cleaning lessens as the request size increases. Since the SSD supports command queueing, it accepts multiple commands and services them in parallel if possible. Accordingly, when the request size is 128KB, the fragmented requests are serviced at the same time by means of the SSD. The fragmentation overhead becomes inconsequential, which in turn brings less performance gain by defragmentation.

Regarding the trim cost, we deleted the database file and issued trim commands to the SSD while measuring the elapsed time (Trim Cost). As shown in Fig. 12, AALFS dramatically reduces the trim cost from 59.60 seconds to 0.28 seconds due to defragmentation by Defrag cleaning.

4) OPTANE SSD

Contrary to the flash SSD, in the Optane SSD, the performance gap between the original and AALFS after cleaning becomes larger as the request size increases, as shown in Fig. 11d. For example, when the request size is 4KB, the Optane SSD could not achieve a performance gain from defragmentation, while the performance gain increases as the request size becomes larger. The small number of parallel units leads to resource contention inside the Optane SSD when the request size gets larger. As a result, AALFS curtails the select cost by up to 37.7% when the request size is 128KB, compared with that of the original.

D. FRAG-AWARE VICTIM SELECTION POLICY

To evaluate our victim selection policy, we compared it with the cost-benefit policy on the MicroSD. In this experiment, as described in Table 4, we concurrently created various sizes of files with different request sizes. In this experiment, the smaller the request size is, the more fragmented the file becomes. When we created the files, we utilized O_DIRECT write operations of IOzone while creating dump files for fragmentation. Afterward, we deleted the dump files and manually performed Defrag cleaning. To observe the performance difference between the two victim selection policies, we conducted only a limited number of cleaning processes. In our experiment, we set the number of victim candidates to
the cost-benefit policy for file E. The cost-benefit policy only considers age and the valid block counts of segments. On the other hand, our policy chooses a victim segment based not only on their age and the number of valid blocks but also on the fragmentation degree in order to exploit the defragmentation effect of Defrag cleaning. The performance of reading the other files remains unchanged because we performed the cleaning restrictively, and so, the segments of those files were not selected as victim segments.

VI. RELATED WORK AND DISCUSSION

Fragmentation has been continuously studied by many researchers. Ji et al. [8] observed the fragmentation
problem in mobile systems and investigated the performance degradation considering the internals of flash-based storage. Conway et al. [7] and Kadekodi et al. [6] discovered that fragmentation occurs on contemporary filesystems, such as Ext4, F2FS, etc., and negatively influences SSD as well as HDD. Additionally, they proposed aging tools to replay fragmentation for various filesystems. However, previous work focused only on HDD and SSD without consideration on Optane SSD which is one of the state-of-the-art technology storage devices. To the best of our knowledge, this is the first work to examine fragmentation on various types of storage devices including Optane SSD, while providing detailed hardware-specific reasons.

Hahn et al. [20] proposed a copyless defragmenter, called Janus, which defragments filesystems without additional write operations. Janus eliminates fragmentation by modifying the mapping table inside the SSD instead of migrating the physical data. However, Janus requires a dedicated SSD that supports the L2P remapping operation, which is not publicly available.

The previous work of AALFS [29] re-orders valid blocks based exclusively on the inode number. However, it cannot eliminate fragmentation in the case when the valid blocks randomly exist in terms of file offset. Even if the blocks from the same file continguously lie on the filesystem, sequential I/O operations to those blocks are still fragmented if the block sequence does not follow the file offset order. To solve this problem, AALFS includes file offsets in the re-ordering criteria, thereby eliminating fragmentation in any cases. Additionally, the previous work can defragment blocks within a single segment so that it cannot eliminate fragmentation if the subsequent blocks are located in another segment. On the other hand, AALFS performs the re-ordering process against multiple segments using the defrag queue, thereby increasing the defragmentation effect.

Here is a summary of the advantages and limitations of AALFS. First, AALFS can effectively eliminate fragmentation, especially in the case when multiple data blocks from the same file co-exist within several segments. One of such cases is when multiple processes perform sequential writes simultaneously. Second, the actual amount of I/Os generated by Defrag cleaning is identical to that of the conventional cleaning scheme. Therefore, AALFS does not curtail the storage lifetime compared with the conventional LFS. Third, the time complexity of Defrag cleaning is $O(s \cdot b)$, where $s$ is the number of segments and $b$ is the number of blocks in a segment. However, additional computation for Defrag cleaning has a negligible effect on the elapsed time of the cleaning process. Finally, the elapsed time of the cleaning process due to the fact that the duration mainly depends on the I/O performance rather than the performance of CPU computation. Therefore, AALFS does not noticeably increase the elapsed time of the cleaning process. Finally, LoC of Defrag cleaning and Frag-aware victim selection is 342 and 171, respectively. Thus, we believe our scheme can be easily deployed to conventional log-structured filesystems without requiring a high engineering cost.

However, AALFS still has some limitations. First, when data blocks belonging to the same file are located far away from each other, AALFS might not be able to put them together because it eliminates fragmentation within a limited number of segments. Such cases include a workload in which a tiny portion of a file is updated with long intervals. Second, since Defrag cleaning additionally maintains metadata of valid blocks, AALFS incurs additional memory overheads. For each valid block, 44 bytes are needed in total (4 bytes for inode number, 4 bytes for file offset, 36 bytes for storing the other metadata). Therefore, in the worst case where all the blocks (512 blocks) in a segment are valid, 704KB (44 bytes * 512 blocks * 32 segments) are needed when AALFS cleans 32 segments at a time. However, we believe that the memory overhead is negligible since AALFS immediately deallocates such memory space immediately after performing Defrag cleaning.

The main purpose of AALFS is to remove fragmentation within a single file, called intra-file fragmentation. However, since the inode numbers are sequentially allocated in order of file creation, we expect AALFS to mitigate inter-file fragmentation in the case when multiple files are read in the creation order.

VII. CONCLUSION
In this paper, we experimentally demonstrated that log-structured filesystems become highly fragmented under certain workloads, and the fragmentation degrades I/O performance regardless of the underlying storage type. This paper presented an anti-aging log-structured filesystem, called AALFS, which eliminates existing fragmentation during the cleaning process without increasing write traffic. Experimental results with both IOzone and the SQLite database demonstrate that AALFS dramatically eliminates existing fragmentation and significantly improves the I/O performance of LFS.

REFERENCES
[1] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Generating realistic impressions for file-system benchmarking,” in Proc. USENIX FAST, 2009, pp. 125–138.
[2] K. A. Smith and M. I. Seltzer, “File system aging–increasing the relevance of file system benchmarks,” in Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst. (SIGMETRICS), 1997, pp. 203–213.
[3] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan, “File system logging versus clustering: A performance comparison,” in Proc. USENIX ATC, 1995, pp. 1–21.
[4] G. de Nijs, A. Biesheuvel, A. Denissen, and N. Lambert, “The effects of filesystem fragmentation,” in Proc. OLS, 2006, pp. 193–208.
[5] W. Hyun Ahn, K. Kim, Y. Choi, and D. Park, “DFS: A de-fragmented file system,” in Proc. 10th IEEE Int. Symp. Modeling, Anal. Simulation Comput. Telecommun. Syst., Oct. 2002, pp. 71–80.
[6] S. Kadekodi, V. Nagarajan, G. R. Ganger, and G. A. Gibson, “Geriatrics: Aging what you see and what you don’t see. A file system aging approach for modern storage systems,” in Proc. USENIX ATC, 2018, pp. 691–703.
[7] A. Conway, A. Bakshi, Y. Jiao, W. Jannen, Y. Zhan, J. Yuan, M. A. Bender, R. Johnson, B. C. Kuszmaul, D. E. Porter, and M. Farach-Colton, “File systems fated for senescence? Nonsense, says science!” in Proc. USENIX FAST, 2017, pp. 45–58.
[8] C. Ji, L.-P. Chang, L. Shi, C. Wu, Q. Li, and C. J. Xue, “An empirical study of file-system fragmentation in mobile storage systems,” in Proc. USENIX HotStorage, 2016, pp. 1–5.
[9] R. Kesavan, M. Curtis-Maury, V. Devadas, and K. Mishra, “Storage gardening: Using a virtualization layer for efficient defragmentation in the Wall file system,” in Proc. USENIX FAST, 2019, pp. 65–78.

[10] J. Xu and S. Swanson, “Nova: A log-structured file system for hybrid volatile/non-volatile main memories,” in Proc. USENIX FAST, 2016, pp. 323–338.

[11] M. Rosenblum and J. K. Ousterhout, “The design and implementation of a log-structured file system,” ACM Trans. Comput. Syst., vol. 10, no. 1, pp. 26–52, Feb. 1992.

[12] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, “F2fs: A new file system for flash storage,” in Proc. USENIX FAST, 2015, pp. 65–78.

[13] D. Park, S. Cheon, and Y. Won, “Suspend-aware segment cleaning in log-structured file system,” in Proc. USENIX HotStorage, 2015, pp. 1–5.

[14] Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, “Optimizations of LFS with slack space recycling and lazy indirect block update,” in Proc. 3rd Annu. Haifa Exp. Syst. Conf. (SYSTOR), 2010, pp. 1–9.

[15] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, “The new ext4 filesystem: Current status and future plans,” in Proc. OLS, 2007, pp. 21–33.

[16] A. K. KV, M. Cao, J. R. Santos, and A. Dilger, “Ext4 block and inode allocator improvements,” in Proc. OLS, vol. 1, 2008, pp. 263–274.

[17] T. Sato, “Ext4 online defragmentation,” in Proc. OLS, 2007, pp. 179–186.

[18] O. Rodeh, J. Bacik, and C. Mason, “BTRFS: The Linux B-tree filesystem,” ACM Trans. Storage, vol. 9, no. 3, pp. 1–32, Aug. 2013.

[19] (2019). [Online]. Available: https://manpages.debian.org/testing/2fs-tools/defrag.2fs.8.en.html

[20] S. S. Hahn, S. Lee, C. Ji, L.-P. Chang, I. Yee, L. Shi, C. J. Xue, and J. Kim, “Improving file system performance of mobile storage systems using a decoupled defragmenter,” in Proc. USENIX ATC, 2017, pp. 759–771.

[21] L. Yang, F. Wang, Z. Tan, D. Feng, J. Quan, and S. Tu, “ARS: Reducing F2FS fragmentation for smartphones using decision trees,” in Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE), Mar. 2020, pp. 1061–1066.

[22] H. Kim, D. Shin, Y. H. Jeong, and K. H. Kim, “SHRD: Improving spatial locality in flash storage accesses by sequentializing in host and randomizing in device,” in Proc. USENIX FAST, 2017, pp. 271–284.

[23] Y. Pan, Y. Li, H. Zhang, H. Chen, and M. Lin, “GFTL: Group-level mapping in flash translation layer to provide efficient address translation for NAND flash-based SSDs,” IEEE Trans. Consum. Electron., vol. 66, no. 3, pp. 242–250, Aug. 2020.

[24] F. Chen, D. A. Koufaty, and X. Zhang, “Understanding intrinsic characteristics and system implications of flash memory based solid state drives,” in Proc. 11th Int. Joint Conf. Meas. Modeling Comput. Syst. (SIGMETRICS), 2009, pp. 181–192.

[25] C. Gao, L. Shi, K. Liu, C. J. Xue, J. Yang, and Y. Zhang, “Boosting the performance of SSDs via fully exploiting the plane level parallelism,” IEEE Trans. Parallel Distrib. Syst., vol. 31, no. 5, pp. 2185–2200, Sep. 2020.

[26] R. Liu, X. Chen, Y. Tan, R. Zhang, L. Liang, and D. Liu, “SSDKeeper: Self-adapting channel allocation to improve the performance of SSD devices,” in Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS), May 2020, pp. 966–975.

[27] Embedded Multimedia Card (eMMC). Electrical Standard 4.51, JEDEC Solid State Technol. Assoc., Arlington, VA, USA, 2012.

[28] K. Wu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Towards an unwritten contract of Intel Optane SSD,” in Proc. 11th USENIX Workshop Hot Topics Storage File Syst. (HotStorage), 2019, pp. 1–8.

[29] J. Park, D. H. Kang, and Y. I. Eom, “File defragmentation scheme for a log-structured file system,” in Proc. 7th ACM SIGOPS Asia–Pacific Workshop Syst. (APWys), 2016, pp. 1–7.