File Defragmentation Scheme for a Log-Structured File System

Jonggyu Park
Sungkyunkwan University
whdrb2722@skku.edu

Dong Hyun Kang
Sungkyunkwan University
kkangsu@skku.edu

Young Ik Eom
Sungkyunkwan University
yieom@skku.edu

ABSTRACT

In recent years, many researchers have focused on log-structured file systems (LFS), because it gracefully enhances the random write performance and efficiently resolves the consistency issue. However, the write policy of LFS can cause a file fragmentation problem, which degrades sequential read performance of the file system. In this paper, we analyze the relationship between file fragmentation and the sequential read performance, considering the characteristics of underlying storage devices. We also propose a novel file defragmentation scheme on LFS to effectively address the file fragmentation problem. Our scheme reorders the valid data blocks belonging to a victim segment based on the inode numbers during the cleaning process of LFS. In our experiments, our scheme eliminates file fragmentation by up to 98.5% when compared with traditional LFS.

Keywords

Log-structured file systems, file defragmentation, cleaning

1. INTRODUCTION

For several years, flash based storage devices (flash memory) have been used in various types of systems, from mobile devices to servers, due to their attractive features including low power consumption and high memory density. However, flash memory has several drawbacks such as limited lifetime and the need to perform out-place updates because of its different characteristics compared with HDD. To address the drawbacks, flash memory utilizes a software layer called flash translation layer (FTL) that spreads out write operations to extend the lifetime of flash memory. The FTL also reclaims invalidated pages caused by out-place updates during garbage collection (GC). Unfortunately, GC can be harmful to I/O performance and the lifetime of the flash memory because it generates additional I/Os to move valid pages belonging to a victim block to a new block. In particular, random writes exacerbate this overhead because they scatter invalid blocks in the flash memory. As a result, the amount of blocks to be copied during GC can increase [10].

While various flash memory based file systems [4, 14, 8] have been studied for minimizing these drawbacks, log-structured file systems (LFS) [12] effectively address them with an append-only write policy which handles most writes sequentially. However, LFS suffers from the file fragmentation problem which degrades overall performance of the file system. When several processes concurrently create files with synchronous writes on LFS, each file becomes fragmented because LFS allocates blocks from the same sequential pool to the files alternately. Also, the file fragmentation problem can be happened on LFS when a file is partially updated or the size of a file increases. Even though file fragmentation on flash memory is a less serious problem than on HDD due to the absence of seek time, it still degrades the read performance of the flash memory due to two main reasons. First, file fragmentation can split block requests because a block request represents a list of contiguous blocks [9]. The split block requests disturb effective block I/O and degrade I/O performance. In addition, flash memory has a prefetch function, where the FTL loads subsequent data in advance to enhance read performance. However, since FTLs prefetch data based on the LBAs [3], file fragmentation may cause the FTL to prefetch useless data. Consequently, it may degrade the read performance.

Previous studies about file fragmentation have focused on Ext4 and HDD [6, 13, 16, 15]. For example, Aneesh et al. [6] suggested the Ext4 multiblock allocator. It pre-allocates contiguous blocks to a file to minimize file fragmentation. But, when the pre-allocated blocks are not used, the scheme can be wasteful about space in LFS because LFS cannot reuse the blocks due

APSys '16, August 04-05, 2016, Hong Kong, China.
 ISBN 978-1-4503-4265-0/16/08...$15.00
DOI: http://dx.doi.org/10.1145/2967360.2967622

to the append-only write policy. Takashi et al. [13] also suggested Ext4 online defragmentation. It reorders already fragmented blocks to be contiguous. However, its additional write operations reduce the lifetime of flash memory.

Since a log-structured file system disallows in-place updates, LFS has to perform the cleaning process for reclaiming free space from invalid blocks which is similar to garbage collection in FTLs [11, 2]. However, the existing cleaning process cannot eliminate file fragmentation because it does not consider the order of the victim blocks. Therefore, in this paper, we present a novel cleaning scheme for file defragmentation on a log-structured file system. Our cleaning process defragments files, rearranging valid blocks based on inode numbers. Furthermore, we also present a fragmentation-aware victim selection policy to enhance our cleaning effect. Because the fragmentation degree is an important factor in our cleaning scheme, our fragmentation-aware victim selection policy considers it as well as the number of valid blocks during the victim selection process. We implement our cleaning scheme and fragmentation-aware victim selection policy in F2FS [7] which is a log-structured file system in the Linux.

The rest of this paper is organized as follows. In Section 2, we observe file fragmentation effects on the read performance and the design of a log-structured file system. Section 3 explains our cleaning scheme and victim selection policy in detail. Experimental results are presented in Section 4. The conclusions of this study are described in Section 5.

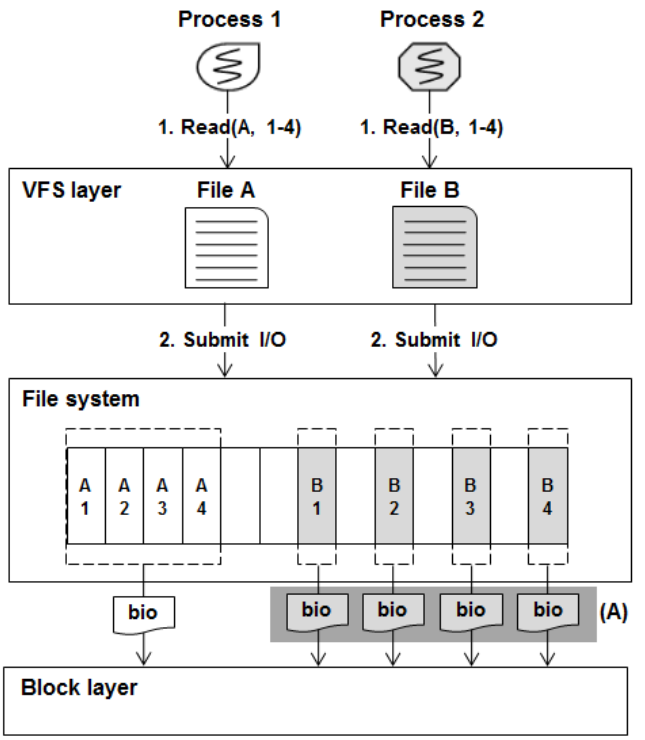


Figure 1: File fragmentation and split bios. On the file system layer, each letter stands for a file name and each number stands for a file offset.

2. RELATED WORK AND MOTIVATION

2.1 File Fragmentation

SSDs consist of multiple planes and channels which can be accessed in parallel. Utilizing this parallelism, FTLs can prefetch data from subsequent LBAs in advance to enhance sequential read performance [3]. Therefore, sequential read performance is much better than random read performance on SSD. However, file fragmentation disturbs prefetching because a file is split into multiple fragments as depicted in Fig. 1. File fragmentation also increases kernel overheads and decreases the sequentiality of block I/O. The kernel uses bio structures which have the information of contiguous blocks to submit block I/O requests to the block layer. When a process issues a read request for a fragmented file, the kernel should make more bios for the read request because they cannot be merged as described in Fig 1 (A). Since split I/O is handled one by one in the underlying storage, file fragmentation also decreases overall read performance. This file fragmentation problem can occur for several reasons. For example, when multiple processes concurrently issue synchronous write operations on LFS, which disrupt buffering and grouping writes, or a file is partially updated, it can cause file fragmentation. Also, an aged file system can have file fragmentation when files are appended with new data, due to the lack of contiguous free space[5].

We first measured how much file fragmentation can affect sequential read performance on flash memory by using a SD card and a SSD as underlying storage devices. We created 50MB files that are fragmented into 4KB – 1MB fragments with synchronous writes from IOzone [1]. We measured sequential read performance of the files with 128KB record size. In Fig. 2, the x axis is the fragment size and the y axis is sequential read performance (KB/s).

In Fig. 2, sequential read performance of the file that is fragmented at a granularity of 1MB is about 7.5x higher than that of the file that is fragmented at a granularity of 4KB. This graph also shows the unsurprising pattern that sequential read performance decreases as the fragmentation problem becomes worse. This degradation of sequential read performance by fragmenation is also happened on SSD. In Fig.2, according to the size of fragments, the difference of the sequential read performance is 1.41 times at most.

2.2 A Log-structured File System

Before we propose our cleaning scheme, we explain a log-structured file system. First, we describe the layout of a log-structured file system. Unlike journaling

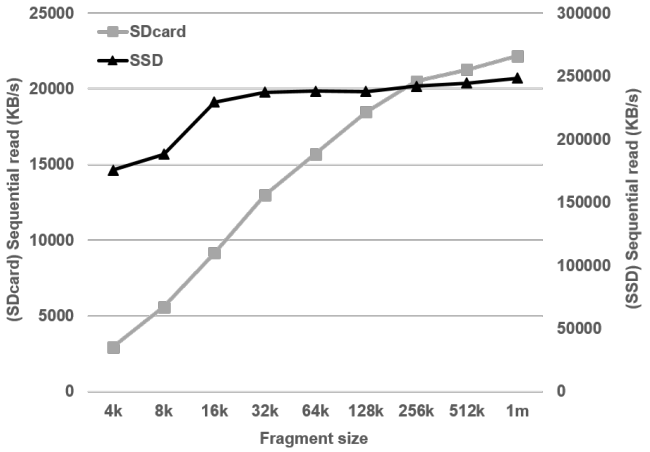


Figure 2: Fragmentation effects on sequential read performance.

file systems (e.g., Ext3 and Ext4), a log-structured file system divides storage into segments regardless of the block type, for examples inode blocks, data blocks, and direct blocks.

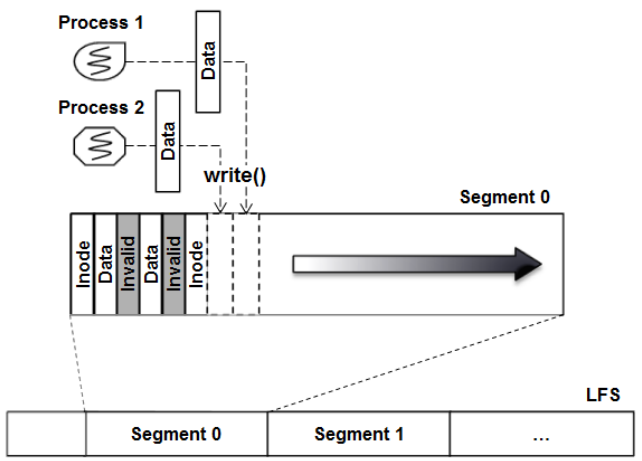


Figure 3: The layout of a log-structured file system and append-only write policy.

When a process issues write operations, LFS allocates a new block at the end of the current segment to avoid in-place updates as depicted in Fig 3. (append-only write policy). Even though the append-only policy makes LFS handle most of the write operations sequentially, it can cause file fragmentation. For example, when a process overwrites data in a file, the overwritten data becomes separated from other data belonging to the same file (file fragmentation). In addition, when many processes concurrently create files with synchronous writes, which disturb buffering and grouping writes, the files become fragmented because LFS allocates blocks from the same sequential pool to the files alternately.

LFS must perform the cleaning process to make free blocks. However, the cleaning process degrades performance because of its overhead. We describe the work of a cleaning thread in detail below.

1. Select a victim segment according to a victim selection policy (e.g., a greedy or cost-benefit policy)
2. Check if each block in the victim segment is valid or not
3. Copy the valid block to a new segment through main memory when a valid block is detected
4. Repeat steps 2-4 for all blocks in the victim segment
5. If all valid blocks are copied, the victim segment is marked as a free segment

In order to reduce cleaning overhead, LFS provides two victim selection policies, greedy and cost-benefit. Greedy policy selects a segment which has the minimum number of valid blocks to guarantee the lowest copy overhead and the most efficient free space creation. The cost-benefit policy considers the last modification time of each segment as well as the number of valid blocks to choose a victim segment which will last for a long time.

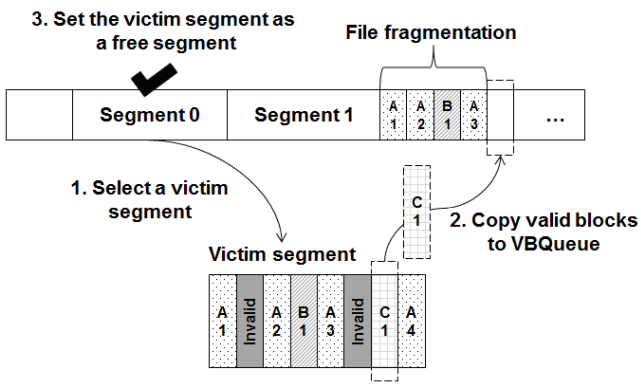


Figure 4: Existing cleaning scheme of LFS.

The cleaning process provides a potential chance to defragment files. Nevertheless, the existing cleaning process does not utilize the chance by copying valid blocks to a new segment in the original order, and thus consequently inherits file fragmentation as depicted in Fig. 4. This observation is our motivation.

3. LFS CLEANING SCHEME FOR FILE DEFRAGMENTATION

In this section, we suggest a cleaning scheme for defragmentation. We also suggest a fragmentation-aware victim selection policy which considers the degree of fragmentation to enhance our cleaning scheme.

3.1 A Cleaning Scheme for Defragmentation

The existing cleaning process cannot eliminate the file fragmentation problem because it does not consider the order of the victim blocks. To solve this problem, we suggest a cleaning scheme, which reorders valid blocks within a segment for file defragmentation on LFS. First, our cleaning scheme selects a victim by a victim selection policy and checks if blocks in the segment are valid. When our cleaning scheme detects a valid block in the victim segment, it loads the valid block into a queue (VBQueue) in main memory and delays copying the valid block to a new segment. After all valid blocks in the segment are loaded to VBQueue, it sorts the valid blocks by inode numbers and copies them to a new segment.

This sorting operation reorders valid blocks to be contiguous with other blocks belonging to the same file. This is possible because each file has one unique inode number. As a result, our cleaning scheme eliminates file fragmentation within each segment as described in Fig. 5, and, since sorting is done in main memory without additional writes to disks, it has subtle overhead and fits with flash memory which has limited lifetime. We implemented VBQueue utilizing a list structure provided by the kernel, and we utilize the list_sort() function in the kernel to sort valid blocks.

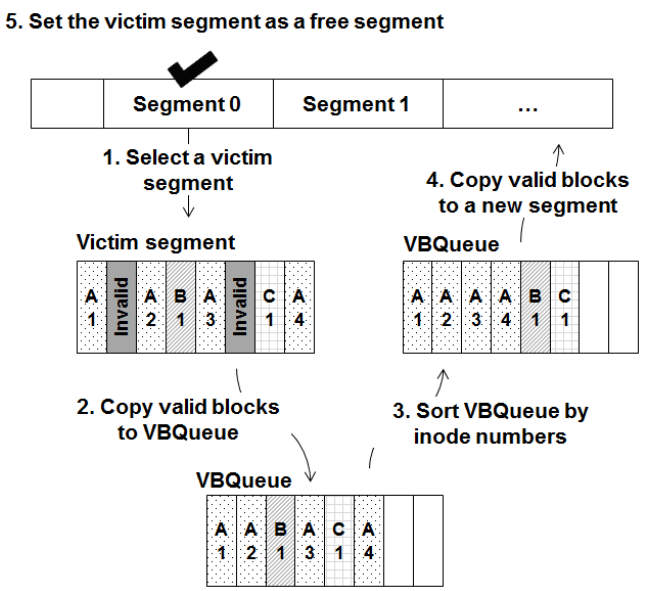


Figure 5: Our cleaning scheme for defragmentation.

3.2 Fragmentation-aware Victim Selection Policy

Existing victim selection policies cannot exploit our cleaning scheme. For example, there are two dirty segments which have the same number of valid blocks. However, one is highly fragmented and another is contiguous. Although the fragmented one can be defragmented by our cleaning scheme, existing victim selection policies may choose the non-fragmented segment. For this reason, we also suggest a fragmentation-aware victim selection policy to exploit our cleaning scheme.

Our victim selection policy utilizes the fragmentation degree of each segment to select a victim segment. Algorithm 1 shows how to measure the fragmentation degree of a segment. First, we explain the variables and the structure used in algorithm 1.

- frag_value : the fragmentation degree of the segment
- cur_inode : the inode number of the current block
- old_inode : the inode number of the previous block
- ino_tree : a set of all previous inode numbers

To implement ino_tree, we utilize rb-tree in the linux kernel because our algorithm has few inserts and many search operations. We also implemented an ino_tree _insert_search() function to combine an insert operation and a search operation into one function to reduce duplicated work such as traversing the tree.

Algorithm 1 How to measure fragmentation degree of a segment.

```
Result: frag_value
while all blocks in a victim segment do
    cur_inode = current inode number;
    if cur_block is invalid then
        if cur_inode != old_inode then
            if cur_inode is not in ino_tree then
                insert cur_inode into ino_tree;      (#1)
            else
                frag_value++;                        (#2)
            end
            old_inode = cur_inode;                   (#3)
        end
    else
        old_inode = NULL;                            (#4)
    end
end
```

First, a victim selection thread checks whether a block in a segment is valid or not. If it is invalid, it sets old_inode to NULL (#4). If it is valid, it compares cur_inode with old_inode to check the fragmentation of the blocks. If cur_inode and old_inode are the same, they are not fragmented because they are in a same file and their LBAs are contiguous. On the other hand, if cur_inode and old_inode are different, the victim selection thread checks if the current inode number exists in ino_tree. When the inode number does not exist in ino_tree, the thread inserts the inode number into the tree (#1). In this case, the block is not fragmented because the inode number did not previously exist in the tree. If the same inode number exists in

ino_tree, it means there are fragmented blocks belonging to the same file and they can be defragmented during our cleaning scheme. In this case, the thread increases frag_value (#2). To check the next block, the victim selection thread sets old_inode to cur_inode (#3) and points to the next block. This process repeats until the thread checks all blocks in the segment.

In addition to frag_value, we also utilize the modification time and the number of valid blocks to exploit the advantages of the prior polices. First, we choose several candidates for a victim segment using modification time and the valid block counts. Utilizing our algorithm, we then calculate the fragmentation degrees of the candidates and choose the most highly fragmented segment as a victim segment.

4. EXPERIMENTAL EVALUATION

We implemented our cleaning scheme and fragmenta tion-aware victim selection policy in the F2FS file system [7] which is a log-structured file system in the Linux. We evaluated our scheme on SD card1 (16GB Kingston), SD card2 (16GB Sandisk), and an SSD (120GB Intel SSD). They all are partitioned for efficient experiments. To cause file fragmentation, we created several files using synchronous writes from IOzone [1] with 8 threads. Following that, we deleted several files to create victim segments and created several dump files to trigger the cleaning process. After cleaning, we measured sequential read performance and the number of fragments (separate groups of contiguous blocks from the same file) to quantify the degree of file fragmentation.

We also measured the effect of our fragmentation-aware victim selection policy. In this experiment, we created various sizes of files concurrently and compared the read performance of our policy with that of the cost-benefit policy.

4.1 Our Cleaning Scheme

Fig. 6 shows the sequential read performance of one fragmented file while increasing the read request size from 4KB to 2M after the cleaning process. Note that the actual I/O request size in all experiments is bigger than 128KB due to the read-ahead effect of the virtual file system (VFS) even if a process issues a 4KB read operation. Table 1 shows the number of fragments of each file in the experiments. Baseline is the number of fragments before the cleaning process. Because we created different sizes of files on the SSD and the SD cards, their baseline values are different.

Fig.6 (a) shows that our cleaning scheme improves read performance by up to 6.77x compared with the existing cleaning scheme on SD card1. Our cleaning scheme also eliminates 99.5% of fragmentation while the existing cleaning process eliminates 1.5% of fragmentation. This is because our cleaning scheme reorders fragmented blocks contiguously while the existing cleaning inherits file fragmentation.

Fragments	SD card1	SD card2	SSD
Baseline	2560	2560	25598
Existing cleaning	2549	2556	25260
Our cleaning	39	41	411

Table 1: The number of fragments in a file comparing our cleaning scheme with existing cleaning scheme.

We also experimented on another SD card and SSD in the same way. In all experiments, our cleaning scheme shows much better read performance than the existing cleaning process. In Fig. 6 (b), it shows up to 4.34x better read performance than the existing cleaning scheme on the SD card2. It also eliminates 98.4% of fragmentation while the existing cleaning scheme eliminates 0.2% of fragmentation. On the SSD, our cleaning scheme shows at most 1.27x better read performance than the existing cleaning scheme. Although our cleaning scheme eliminates a similar percentage of fragments in all devices, our cleaning scheme shows less performance gains on the SSD. It is because the SD card and SSD have different internal architectures such as DRAM size and the number of parallel units. We leave further exploration of the differences for future work.

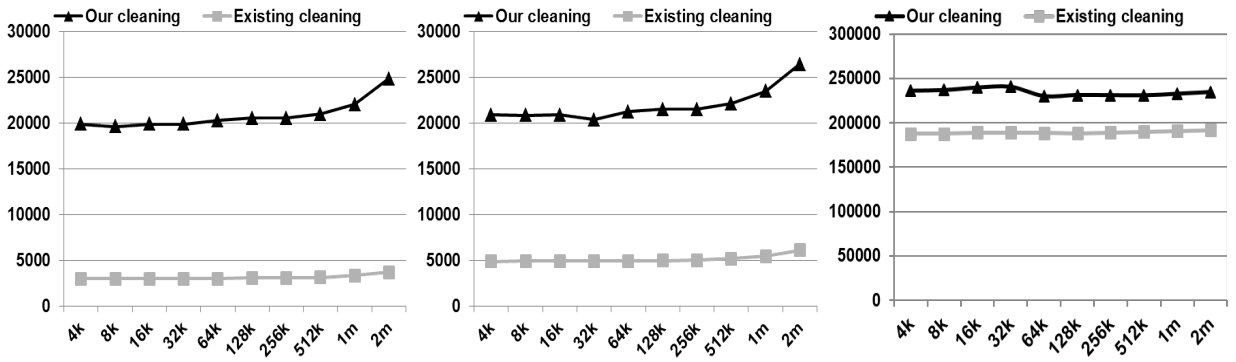
4.2 Fragmentation-aware Victim Selection Policy

Number	1	2	3	4	5	6	7	8
File size (MB)	1	1	20	20	20	20	50	100
Record (KB)	4	4	64	64	4	4	4	128

Table 2: The file information used in the experiment. All files are concurrently created with synchronous write operations.

We compared our fragmentation-aware victim selection policy with the cost-benefit policy on the SD card1 to analyze the suitability with our cleaning scheme. In this experiment, we concurrently created various sizes of files as described in Table 2. We also created dump files and deleted them to trigger cleaning processes. To verify differences in performance between the two victim selection policies, we generated a limited number of cleaning processes by configuring the size of the dump files.

Utilizing the fragmentation degree of each segment, our fragmentation-aware victim selection policy chooses a victim segment that can benefit from the substantial defragmentation effect during our cleaning scheme. On the other hand, the cost-benefit policy only considers age and the valid block numbers of segments without considering fragmentation degrees. Consequently, in Fig. 7, our fragmentation-aware policy outperforms the cost-benefit policy at file 5, 6, and 7 because our policy selects these highly fragmented files and our cleaning scheme eliminates the fragmentation. In particular, our



(a) SD card1 (b) SD card2 (c) SSD

Figure 6: Sequential read performance. X axis is the size of request. Y axis is the sequential read performance (KB/s).

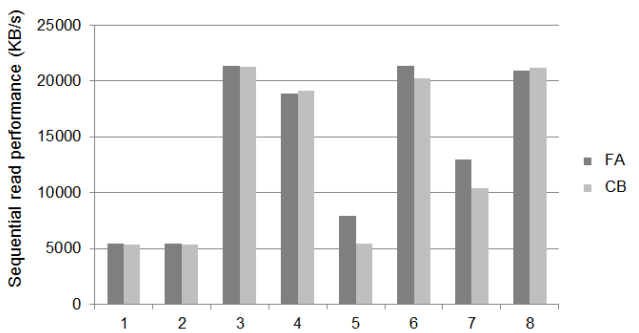


Figure 7: Our fragmentation-aware victim selection policy vs. the cost-benefit victim selection policy. FA is our fragmentation-aware victim selection policy and CB is the cost-benefit victim selection policy.

fragmentation-aware policy shows 1.4x better read performance than the cost-benefit policy for file 5. The file 5 and file 6 have the different performance although they have the same configurations. It is because more segments of file 5 are selected as victim segments than those of file 6.

5. CONCLUSIONS

In this paper, we suggest a cleaning scheme for file defragmentation within each segment on a log-structured file system. Our cleaning scheme sorts valid blocks by inode numbers during the cleaning process for defragmentation without additional writes. To enhance our cleaning scheme, we also suggest a fragmentation-aware victim selection policy. It utilizes fragmentation degrees to select a victim segment which can be defragmented effectively during our cleaning scheme. In our experiments, our cleaning scheme shows at most 6.77x better sequential read performance than the existing cleaning scheme. It also eliminates file fragmentation by up to 98.5%.

6. ACKNOWLEDGEMENTS

This work was partly supported by ICT R&D program of MSIP/IITP. [R0126-15-1065 , (SW StarLab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays] and the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2016-H8501-16-1015) supervised by the IITP(Institute for Information & communications Technology Promotion).

7. REFERENCES

[1] IOzone filesystem benchmark. http://www.iozone.org/.

[2] T. Blackwell, J. Harris, and M. I. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of USENIX 1995 Technical Conference*, pages 277–288, 1995.

[3] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 181–192. ACM, 2009.

[4] J. Engel and R. Mertens. Logfs-finally a scalable flash file system. In *Proceedings of International Linux System Technology Conference*, pages 135–142, 2005.

[5] 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 *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.

[6] A. K. KV, M. Cao, J. R. Santos, and A. Dilger. Ext4 block and inode allocator improvements. In *Proceedings of Linux Symposium*, pages 179–186, 2008.

[7] C. Lee, D. Sim, J. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *Proceedings of USENIX Conference on File and Storage Technologies*, pages 273–286, 2015.

[8] S.-H. Lim and K.-H. Park. An efficient nand flash file system for flash memory storage. *IEEE Transactions on Computers*, 55(7):906–912, 2006.

[9] R. Love. *Linux kernel development 3rd edition*. Pearson Education, 2010.

[10] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. Sfs: random write considered harmful in solid state drives. In *Proceedings of USENIX Conference on File and Storage Technologies*, pages 139–154, 2012.

[11] M. Rosenblum and J. K. Ousterhout. The lfs storage manager. In *Proceedings of USENIX Summer 1990 Technical Conference*, pages 315–324, 1990.

[12] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[13] T. Sato. Ext4 online defragmentation. In *Proceedings of the Linux Symposium*, pages 179–186, 2007.

[14] A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif. Abstract specification of the ubifs file system for flash memory. In *Proceedings of International Symposium on Formal Methods*, pages 190–206. 2009.

[15] K. A. Smith and M. I. Seltzer. File system aging-increasing the relevance of file system benchmarks. In *ACM SIGMETRICS Performance Evaluation Review*, volume 25, pages 203–213. ACM, 1997.

[16] S. VanDeBogart, C. Frost, and E. Kohler. Reducing seek overhead with application-directed prefetching. In *Proceedings of USENIX Annual Technical Conference*, pages 299–312, 2009.