# Optimizing Fragmentation and Segment Cleaning for CPS based Storage Devices

Qi Li[★], Aosong Deng[★], Congming Gao[★], Yu Liang[‡], Liang Shi[†], and Edwin H.-M. Sha[†]

[★]Chongqing University, China
[‡]City University of Hong Kong, China
[†]School of Computer Science and Software Engineering, East China Normal University
The corresponding author is Liang Shi: shi.liang.hk@gmail.com

## ABSTRACT

To manage the storage device of Cyber-physical systems (CPS), flash friendly file system (F2FS) has been a good choice for its good performance. F2FS is designed with consideration on the characteristics of flash storage as the key factor. Therefore, it has been highly suggested for the management of flash storage devices. However, several drawbacks of F2FS have been identified as the research work goes on, such as bad sequential read performance and segment cleaning induced performance impact. In order to solve these drawbacks, a set of empirical and comprehensive studies are performed on F2FS. Based on the studies, two novel schemes are proposed, including a multi-level threshold synchronous write scheme and a high detection frequency background segment cleaning scheme. Experimental results show that the proposed technologies present encourage improvement.

## CCS CONCEPTS

• **Computer systems organization → Architectures**;

## KEYWORDS

Log-structured file systems, F2FS, fragmentation, segment cleaning, CPS

**ACM Reference Format:**
Qi Li[★], Aosong Deng[★], Congming Gao[★], Yu Liang[‡], Liang Shi[†], and Edwin H.-M. Sha[†], [★]Chongqing University, China, [‡]City University of Hong Kong, China, [†]School of Computer Science and Software Engineering, East China Normal University, The corresponding author is Liang Shi: shi.liang.hk@gmail.com . 2019. Optimizing Fragmentation and Segment Cleaning for CPS based Storage Devices. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3297280.3297306

## 1 INTRODUCTION

Cyber-physical systems (CPS) have been developed during the last decades for its low cost and popularity. In CPS, one of the critical components is the storage device, which is designed with flash memory. To manage the storage device, file system is general deployed. There exist two types of writing mode for most file systems: synchronous write and asynchronous write. The existing research indicates that in CPSs, more than 70% of write operations are synchronous writes, and more than 75% of write operations are random [8]. Unfortunately, in the synchronous random write dominated I/O mode, the I/O performance of flash storage devices will drop dramatically [6, 14]. Log-structured file systems (LFS) [10, 14, 17] addressed the problem with an append-only manner. It divides the storage space into sequential distributed log structures. Multiple synchronous random writes will be sequentially recorded in the log structures and merged into one write request. However, LFS is a traditional file system designed for HDD, without considering the characteristics of flash-based storage like garbage collection, etc.

To optimize the I/O performance of flash storage devices, industry implemented the Friendly to Flash Storage (F2FS) file system, which is designed for the characteristics of flash storage devices [5, 11]. According to the data update frequency, F2FS divides the data storage area into three types: hot, warm and cold. In each type, data is written in an append-only logging manner like LFS. With this on-disk layout, F2FS can effectively reduce the number of synchronous random writes and improve the efficiency of segment cleaning with hot/cold data separation. Due to its advantages on flash storage devices, F2FS has become an academic hot research topic and has been used widely in several CPS based storage devices. As researches progressed, some defects of F2FS are identified. Among them, the problem of data fragmentation and background segment cleaning draw a widespread attention: **First**, because F2FS uses the append-only logging, when multiple processes simultaneously write to multiple files, F2FS assigns consecutive addresses to data from different files. As these files are updated multiple times, the data of the same file will be discontinuously distributed on the storage address, which called data fragmentation. This can lead to a decline of sequential read performance [12, 16]. **Second**, similar to LFS, F2FS needs to perform the segment cleaning to reclaim space. When out of space, the system will suspend the user I/O operation and perform the foreground segment cleaning operation. At the same time, in order to mitigate the interference of the cleaning operation on user operation, F2FS also designed a background segment cleaning mechanism, which is triggered only when the system is idle. However, in actual use, background segment cleaning is difficult to trigger. Park et al. [15] pointed out that due to the conflict with the suspend mode in the mobile device, the system almost can


*SAC '19, April 8–12, 2019, Limassol, Cyprus*

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297306

only reclaim space through foreground segment cleaning, which seriously affects the user experience.

To solve the above problems, in this paper, we proposed two techniques: (1) *MTS Write*, a Multi-Level Threshold Synchronous Write scheme, and (2) *High-Frequency BSC*, a High Detection Frequency Background Segment Cleaning scheme. *MTS Write* dynamically controls the speed of generating data fragments based on different situations. In this way, the balance between sequential read and random write can be guaranteed. *High-Frequency BSC* dynamically adjusts the frequency of background segment cleaning based on system status. At the same time, it takes into account the life of flash memory. Designing these methods is difficult and challenging. It is necessary to consider how to control the speed and using what to control it is more appropriate, and to consider the relationship between the suspend mode and the trigger of the background segment cleaning. Our contributions are as follows:

- We analyzed the performance defect of the F2FS in sequential read performance and background segment cleaning mechanism and found out the main reasons cause to them.
- We proposed *MTS Write*, which effectively improves the sequential read performance while taking into account the random write performance of the system.
- We proposed *High-Frequency BSC*, which effectively improves the background segment cleaning performance of the system without causing significant impact on the life of the flash memory.
- We have implemented these schemes on Linux. Experiments show that the proposed schemes present encouraging results.

The remainder of the paper is organized as follows: Section 2 presents the background and related works. Section 3 provides the problem statement. Section 4 provides the design and implementation of our two optimization techniques. Section 5 describes the experimental evaluation of the proposed schemes. Our conclusions are presented in Section 6.

## 2 BACKGROUND AND RELATED WORK

In this section, F2FS and two typical performance impact problems: data fragmentation and segment cleaning mechanism are presented. Then, the related works are presented.

### 2.1 F2FS, flash friendly file system

With its good performance and flash friendly characteristics, F2FS has become more and more popular among manufacturers and consumers. It has been merged into the main branch of the Linux kernel since version 3.8 and now many mobile devices utilize it as the default file system, like Huawei Mate9, Google Nexus 9, etc. F2FS builds on the concept of LFS but designed new features, such as flash-friendly on-disk layout, multi-head logging and checkpoint recovery, which take into account the characteristics of flash memory. However, like other Log-structure file systems, F2FS also can not avoid these two performance impact problems, data fragmentation and segment cleaning.

### 2.2 Data Fragmentation

F2FS initially allocates the contiguous logical addresses to the same type of data from the same file. Building on append-only logging, F2FS updates data in a way of out-of-place. Therefore, if the user continues to update the data, it will cause the logical address of the same file to be out of order. This is called data fragmentation.

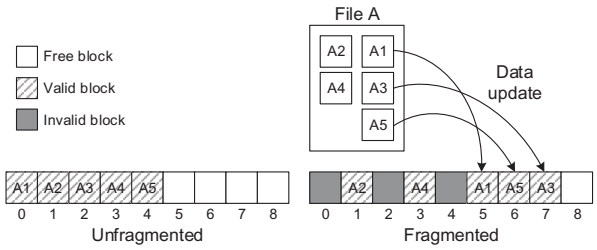


**Figure 1: An example of generating data fragments.**

Figure 1 shows an example on generating data fragments. Initially, file A has a total of 5 pages of data (A1, A2, A3, A4, A5). Their logical addresses in F2FS are continuous (1, 2, 3, 4, 5, respectively), which is un-fragmented. Then the user creates a process in which the A1, A5, and A3 data pages of file A are sequentially updated. When the update finished, the logical addresses of file A's data is out of order, which leads to the fragmentation. In a real application environment, because there may be a large number of processes performing data update operations simultaneously on files, the data fragmentation degree of F2FS is far more serious than the example. As the number of update operations increases, the data fragmentation will gradually increase, which eventually resulting in significant impact on sequential read performance.

### 2.3 Segment Cleaning and Suspend Mode

All file systems that build on append-only logging needs a way to reclaim invalid space. Segment cleaning is such a method for F2FS to reclaim scattered and invalidated blocks and secures free segments for further logging [11]. F2FS performs cleaning in two distinct manners, foreground and background. Foreground cleaning is triggered only when there are not enough free sections, which will suspend all user I/O operations before completed. While the background cleaning is triggered by a kernel thread that periodically wakes up in the background.

*2.3.1 The trigger condition of background segment cleaning.* To reduce interference to the user I/O request, F2FS sets a background thread responsible for system state detection. Only when the system is idle can the background segment cleaning be triggered. Figure 2 shows the workflow of the background detection thread. When there are not many invalid blocks in the system, more valid blocks will be migrated by background segment cleaning if it is triggered, which will generate additional write operations and reduce the life of the flash memory. Therefore, in the workflow of this background thread, it is necessary to adjust the sleep time according to the number of invalid data blocks. By increasing the sleep time, the trigger frequency is reduced, and the damage to the flash memory is reduced. From the above process, we can find that accurately detecting the idle state of the system is important to the triggering of the background segment cleaning.

*2.3.2 The suspend mode of Android device.* Unlike the traditional desktop, CPS always requires a battery to power. In order to reduce the times the device charged and extend the lifetime of the battery,

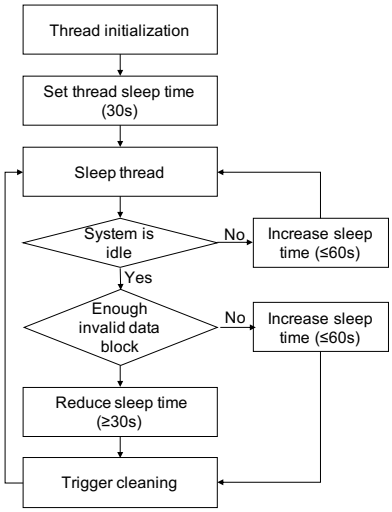


**Figure 2: Workflow of the background detection thread.**

researchers designed and implemented a suspend mechanism in the Linux kernel. In a CPS configured with suspend mode, when the system is idle for a period of time (usually within 5~15 seconds), the device will automatically switch to the suspend mode. In this mode, all processes in the system go to sleep, and most of the hardware stop working. From some tests of the suspend mode in the real environment, researchers drew the following conclusions [15]:

- During the entire test, the device is in suspend mode more than 75% of the time.
- For all system active time periods, more than 95% of them are less than 10 seconds.

We can find that the suspend mode will make the system sleep for most of the time, so that the background thread in the system won't be executed for a long time.

## 2.4 Related Work

This paper addresses two critical problems of F2FS: file fragmentation and infrequent background segment cleaning. So there are two groups of works related to this paper. The first group aims to do file defragmentation. The second group is to improve the cleaning scheme of the log-structured file system.

**File Defragmentation** There have been a series of works on the file defragmentation. DFS is a file system, which can automatically re-cluster data blocks of fragmented files into a sequential free space [1]. For some file systems, users are allowed to manually trigger defragmentation operations on selected files. For example, e4defrag [18] is a file defragmentation tool for Ext4 file system. The e4defrag reduces fragments by copying data from the fragmented file to a large sequential donor, and then re-direct original file pointers to the donor file. However, a prior work has reported that frequently to do defragmentation by copying data could shorten the lifetime of flash-based mobile storage by more than 10% [7]. To avoid massive data copies, janusd [9] is proposed to do defragmentation by re-mapping a file from fragmented logical addresses to sequential addresses. Differently, MTS Write, which proposed by this paper, dynamically controls the speed of generating data fragments based on different situations. It selects a write mode according to the degree of fragmentation in F2FS. When there are lots of fragments, the in-place update will be chosen. Otherwise, the out-of-place update will be adopted.

**Segment Cleaning Improvement** There are few papers focused on segment cleaning improvement of log-structured. J. Parketal. [16] proposed a cleaning scheme to do defragmentation while recycling space. This scheme sorts valid blocks in victim segments by inode numbers during the cleaning process. The reordered valid blocks can be contiguous with other blocks belonging to the same file, which effectively eliminates file fragmentation within new segments after cleaning. D. Park et al. [15] observed the conflict between the suspend mode of the Android device and the trigger of the background segment cleaning. They proposed a suspend-aware segment cleaning technique to conduct background segment cleaning by using the interval after screen off till the actual suspend state. Differently, the proposed High-Frequency BSC dynamically adjusts the frequency of background segment cleaning according to the amount of free space.

# 3 PROBLEM STATEMENT

In this section, we discuss and analyze the problems on the data fragmentation induced sequential read performance degradation and the conflict between background segment cleaning and the suspend mode of CPS.

## 3.1 Data fragmentation induced sequential read performance degradation

Based on the above section's analysis of the data fragmentation, there are following two reasons causing sequential read performance.

**First**, the data fragmentation affects the merging of data requests by the block layer. In the I/O stack, after the system I/O request is sent from the file system, it must be processed by the block layer and the I/O scheduling layer before it can be sent to the flash storage device. When the size of request sent by the file system is small, the block layer tries to merge multiple small size requests whose addresses are consecutive, thereby reducing the actual number of block I/O requests in the system [2, 13]. However, due to the data fragmentation in the system, when the user performs a read operation on the file, the addresses of read requests generated by file system are discontinuous. The block layer cannot merge these read requests with non-contiguous addresses, thus generates a large number of block I/O requests. This causes the read requests to accumulate and be blocked in the scheduling queue, resulting in a long scheduling latency, which decreases the sequential read performance. **Second**, the data fragmentation reduces the hit ratio of page cache acquired by the read ahead, which is a scheme designed in the Linux to improve the read performance. During the sequential read operation, the system calls a prefetch command to read a number of data pages whose addresses are next to the requests into the page cache. When continuing to read the data, the system first looks up in the page cache. If the data the user requested have been read into the page cache by the prefetch command, the user can read the data directly from the page cache instead of reading it from the flash storage device. If so, it is called a read ahead hit, otherwise, a read ahead miss. When the hit ratio is high, a portion

of the read operation in the system is completed by reading the page cache. Since the page cache is made up of RAM, the read speed is much higher than that of flash memory, which optimizes the read performance. However, due to the data fragmentation, the logically consecutive data may not be sequentially stored. Therefore, a high hit ratio cannot be ensured, which significantly increases the number of reading directly to the flash storage device and results in a decrease in the sequential read performance of the system. Through the analysis above, we can draw the following conclusions:

- As the system continues to update data, the degree of the data fragmentation will gradually increase;
- The data fragmentation interferes with the request merging of the block layer and the read-ahead mechanism of the kernel, which causes the number of read requests to increase.

## 3.2 Conflict between the cleaning trigger and the suspend mode

Based on the previous introduction, there may exist a conflict between the system idle time detection and the suspend mode mechanism in CPS, which makes it difficult to trigger the background segment cleaning.

There are two reasons: **First**, the suspend mode causes the system to be in a sleep state more than 75% of the time. Although no I/O request exists in the system at this time, the background thread used to detect the state of the system also enters to the sleep state. Therefore, the background segment cleaning cannot be triggered. **Second**, the suspend mode makes the system active time period shorter (usually less than 10 seconds). With the development of industrial manufacturing technology, in modern CPSs, the number of processor cores is gradually increasing, and the number of channels of the flash storage device is constantly increasing. These have significantly enhanced the overall parallelism of the device. Therefore, during the active time period, the system will be busier and the actual idle time will be further reduced. Because the interval of the detection for the system idle state of the F2FS is too long (more than 30 seconds), these shorter system idle times are difficult to detect, which makes it harder to trigger the cleaning.

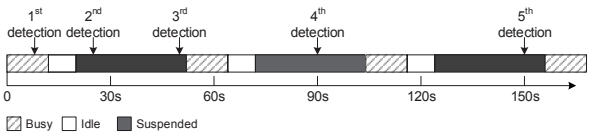


**Figure 3: An example of the system idle state detection.**

Figure 3 shows an example of the system idle state detection process. We can find that on the system timeline, the system idle periods are too short to be easily detected by the background thread with fewer attempts. At the same time, under the default setting, after the thread fails to detect the idle state, its sleep time will be increased, which reduces the frequency of detection. In this case, the possibility of the idle state of the system being detected is further reduced. Therefore, the background segment cleaning will not be triggered for a long time, and the remaining free space of the system will be less and less until the foreground segment cleaning is triggered. At this point, the overall I/O performance will drop significantly. Through the analysis above, we can draw the following conclusions:

- The short idle time in the system is the root cause of the background segment cleaning being difficult to be triggered;
- The current detection frequency of the background thread is really low, which further increases the difficulty in triggering the background segment cleaning.

# 4 DESIGN OF THE OPTIMIZATION SCHEME

In this section, we present the design and implementation of *MTS Write* and *High-frequency BSC*, two optimization schemes for the F2FS file system relating to the performance of the sequential read and background segment cleaning.

## 4.1 MTS Write for Sequential Read Improvement

From the analysis in the previous section, we have identified the reasons for the decline in sequential read performance of F2FS, and the key to optimizing is to reduce the degree of the data fragmentation.

In several CPS systems, more than 70% of write operations are synchronous [4]. It can be seen that the sequential read performance can be effectively improved by reducing the number of the fragments generated by *synchronous write operations*. Based on this, F2FS provides a native configurable write mode, Single-Level Threshold Synchronous Write. In this mode, if the object of a synchronous write is not a directory file and the amount of data is less than the out-of-place update threshold (the default threshold is 32KB), this write operation will update data in an in-place manner. Obviously, this simple scheme can mitigate the data fragmentation. However, when the size of the synchronous write request is small, all the data will be updated in in-place, which results in a significant degradation in the synchronous random write performance.

Figures 4 and 5 show the changes in the number and delay of write requests when F2FS uses out-of-place and in-place to perform synchronous random write operations. As can be seen from these two results, when using the in-place update for synchronous random writes compared to the out-of-place update, the block layer generates more block I/O requests due to the discontinuous request address, which causes higher scheduling delay. And this higher scheduling delay ultimately leads to a decrease in the synchronous random write performance of the F2FS file system. As the number of processes simultaneously performing synchronous random writes increases, the degree of performance degradation increases gradually. When the number of processes reaches 10, the number of write requests increases by 116.2%, and the write request latency increases by 79.8%. Unfortunately, most synchronous write requests in the system are smaller than the default threshold 32KB, and the random write performance of flash storage devices is much lower than read performance. Therefore, Single-Level Threshold Synchronous Write technology has some drawbacks.

In order to optimize the sequential read performance, while taking into account the random write performance, based on the Single-Level Threshold Synchronous Write technology, we design and implement a more adaptive optimization scheme: *MTS Write*. The scheme is composed of two modules, a *Fragmentation Judging* module, and a *Write Mode Selection* module. The *Fragmentation*

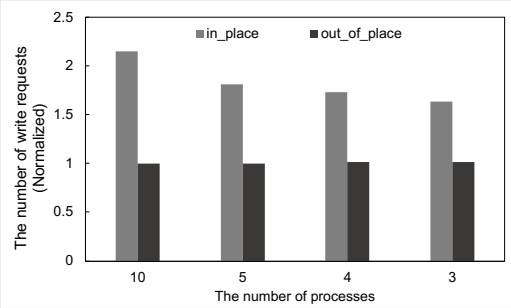


**Figure 4: The normalized number of write requests.**

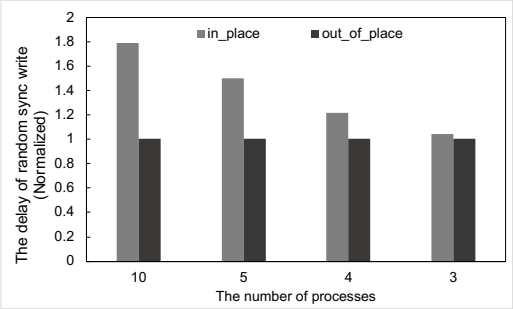


**Figure 5: The normalized latency of synchronous random write request.**

*Judging* judges the degree of the data fragmentation according to the number of invalid data blocks in the F2FS file system and transmits the result to the *Write Mode Selection* module. The *Write Mode Selection* module dynamically performs threshold selection based on the current degree of the fragmentation and determines the mode in which the data is written. Next, we introduce the two modules in detail. Figure 6 is a schematic of the scheme.

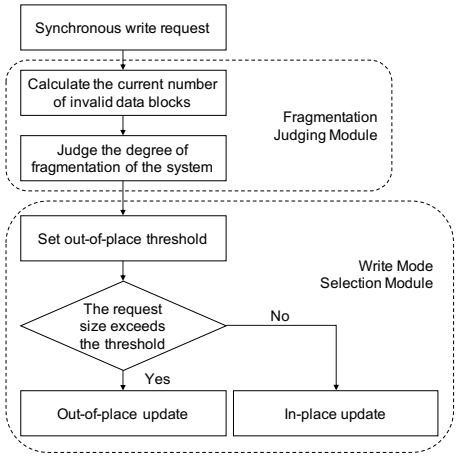


**Figure 6: The process of MTS Write.**

*4.1.1 Fragmentation Judging module.* In order to implement the dynamic selection of the data writing mode, we need to detect the degree of the fragmentation in real time. But it is time-consuming and not practical to scan the entire storage space for the statistics of the fragments. It also brings about a large number of read operations, which has a great influence on the synchronous write. In addition, the data update generates invalid data blocks, of which the number gradually increases as the number of update operation increases. Meanwhile, in section 2.1 we have pointed out that the continuous update of user data is the main cause of the data fragmentation, and the degree of data fragmentation will increase as the update operation increases. Thus we can determine the degree of fragmentation based on the number of invalid data blocks in the system. In the super block of the F2FS file system, system space related metadata is stored. From this information, we can read the total user space capacity, the number of valid data blocks and the free user space capacity. By formula (1), we can get the number of invalid data blocks in real time:

$$Iblocks = Ublocks - Vblocks - Fblocks \tag{1}$$

Where $Iblocks$ denotes the number of invalid data blocks, $Ublocks$ denotes the total number of user space blocks, $Vblocks$ denotes the number of valid data blocks, and the $Fblocks$ denotes the number of free blocks.

After calculating $Iblocks$, the module uses it to judge the degree of the fragmentation. We use $th(i)$ to denote the threshold of the degree of the fragmentation. $th(i)$ can be configured between 1 to 99. In our scheme, the values of $th(i)$ are 10, 20, and 40, to indicate different degrees of fragmentation of low, medium, and high, respectively. Formula (2) is used to calculate the number of critical invalid blocks $CIblocks$ in different $th(i)$ levels:

$$CIblocks = Ublocks \times \frac{th(i)}{100} \tag{2}$$

Then, the degree of the data fragmentation of the system is determined according to the size relationship between $CIblocks$ and $IBlocks$, which is transmitted to the write mode selection module.

*4.1.2 Write Mode Selection module.* This module selects the write mode for each synchronous write request according to the result getting from the last module. When the level of the degree of the fragmentation is low, this module sets a small out-of-place update threshold for synchronous write operations (the minimum threshold in our scheme is 16KB). At this time, these write operations are still written in out-of-place because the size of most of them is bigger than the threshold. As the fragmentation gradually increases to medium level, the module adjusts the out-of-place update threshold to 32KB or 64KB, which reduces the number of some out-of-place write operations and slows down the speed of generating data fragmentation. When the system enters the state of the fragmentation of high level, all the synchronous write requests will be written in in-place to prevent generating new data fragments, ensuring that the read performance of the system does not continue to decrease.

Compared with the Single-Level Threshold Synchronous Writing, our scheme is more adaptive and can dynamically select the data writing mode, which optimizes the performance of sequential read while minimizing the degradation of the performance of synchronous random write.

## 4.2 High-Frequency BSC

As can be seen from the analysis of section 3.2, since we can do nothing when the system is suspended, the key to improve background segment cleaning is to make good use of idle time.

Obviously, by increasing the detection frequency of the background thread, we can increase the number of trigger of background segment cleaning. But it also brings too much data migration, resulting in a reduction in the life of flash memory. Therefore, we propose a technique to control the trigger frequency of the background segment cleaning, so that it can balance the cleaning performance and the life of flash memory.

Combined with the above analysis, we designed and implemented *High-Frequency BSC* to optimize the performance of the background segment cleaning of the F2FS file system. The technique is composed of two modules, a *System Status Detection* module, and a *Fragment Cleaning Frequency Control* module. Figure 7 is the schematic of the technique.

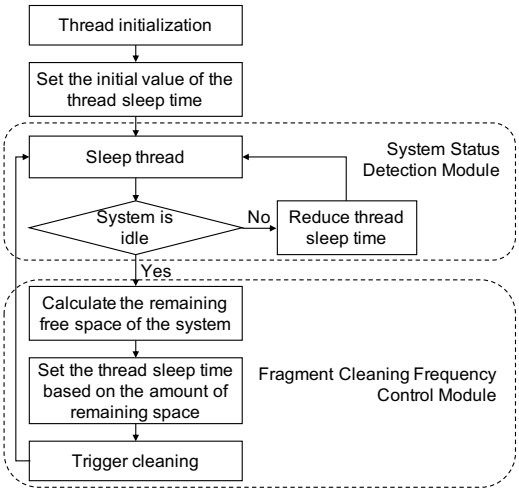


**Figure 7: The process of High-Frequency BSC.**

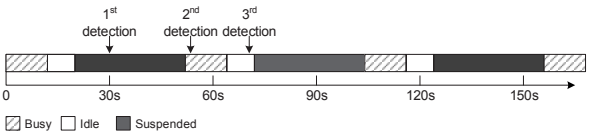


**Figure 8: An example of a successful detection.**

*4.2.1 System Status Detection module.* This module is mainly responsible for detecting the status of the system. As shown in Figure 7, when the background thread failed to detect the idle state, the module will continuously reduce the sleep time of the thread to increase the detection frequency until it succeeds. Figure 8 shows an example of a successful detection: the idle state was detected with fewer attempts compared with Figure 3.

When managed to detect the idle state of the system, the module tells the result to the *Fragment Cleaning Frequency Control* module.

*4.2.2 Fragment Cleaning Frequency Control module.* This module is mainly responsible for dynamically controlling the frequency of the detection to avoid its value being too high. In the F2FS file system, the foreground segment cleaning is triggered when the remaining space of the system is very low (default 5%), so we can also use the remaining space to adjust the trigger of the background segment cleaning.

In our scheme, we define the minimum cleaning trigger interval (10 seconds), the background segment cleaning trigger interval, the minimum free blocks (40% of the total user space), and the total remaining user space, denoted by $MinCi$, $Ci$, $MinFblocks$, $Rspace$, respectively. By formula (3), the module adjusts the background segment cleaning trigger interval dynamically. When there is enough free space in the system, the value of $Ci$ will increase accordingly, thereby increasing the sleep time of the background thread to limit the growth of the frequency of the background cleaning. On the contrary, when out of free space, we assign the value of $MinCi$ to $Ci$, the background segment cleaning will be triggered at a higher rate.

$$Ci = \begin{cases} MinCi \times \frac{Fblocks}{MinFblocks}, & \text{if } Rspace > MinFblocks \\ MinCi, & \text{if } Rspace \leq MinFblocks \end{cases} \quad (3)$$

In a short, by the above two modules, this scheme achieves the balance between the performance of the segment cleaning and the life of the flash memory.

## 5 EXPERIMENTAL EVALUATION

In this section, we conducted comparative experiments and analyzed the experimental data to verify the optimization effects of the two optimization schemes proposed in this paper on the performance of sequential read and the performance of the background segment cleaning.

### 5.1 Experimental Setup

We performed our testing on a 128GB SSD [3] (Table 1) formatted with the F2FS file system, and this SSD is mounted on a desktop with an Ubuntu 15.04 installed (kernel version 4.0.3).

For section 5.2, we designed three groups of comparative experiments to show the number of data fragments, the latency of sequential read and the latency of synchronous random write under different schemes. For section 5.3, we designed three sets of comparative experiments to show the trigger times of background segment cleaning, the amount of invalid data recycled and the amount of valid data migrated, under different system idle time and schemes.

**Table 1: Detailed parameters of the SSD**

| Model | Intel 545s |
|---|---|
| Capacity | 128GB |
| Interface | SATA 3.0 |
| Sequential Read (up to) | 550 MB/s |
| Sequential Write (up to) | 440 MB/s |
| Random Read (up to) | 70000 IOPS |
| Random Write (up to) | 80000 IOPS |

### 5.2 Sequential read optimization analysis

In order to evaluate the optimization effect of *MTS Write* on the performance of sequential read of the system, we design an experiment with the following steps: 1) Format the SSD with the F2FS file system, and mount it to the desktop. 2) Create a text file with big size. 3) Collect the performance of sequential read and the number of the data fragmentation. Use the collected result as the benchmark. 4) Create 5 processes, simultaneously update the file with synchronous write requests, and record the performance of the write requests. 5) Collect the performance of sequential read and the number of the data fragments. 6) Repeat steps 4 and 5.

During the experiment, we used the proposed *MTS Write* and other two synchronous write technologies of the F2FS file system, the default append-only logging technology and the Single-Level Synchronous Write technology, to perform the above evaluations, respectively.

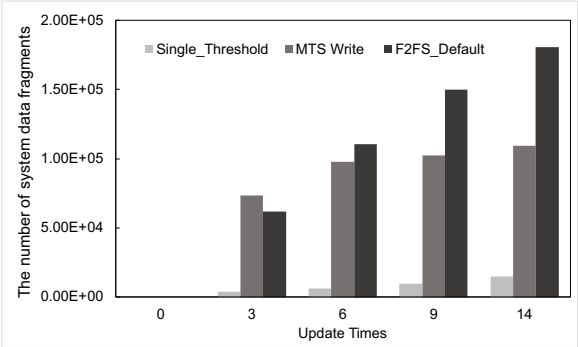


**Figure 9: The number of the data fragments.**

*5.2.1 Results analysis.* Figure 9 shows the number of the data fragments when using different synchronous write technology, where F2FS_Default denotes the F2FS's default append-only logging technology, Single_Threshold denotes the Single-Level Synchronous Write technology, and MTS Write denotes our proposed technology. The append-only logging technology makes the amount of the data fragments grow fast. Eventually, there are 180807 fragments in the system, which is much higher than the other two technologies. Due to the small size of the synchronous write operation, the Single-Level Threshold Synchronous Write technology writes a lot of data in an in-place manner, so that the number of data fragments grows slowly, eventually producing 15070 data fragments. When the degree of the data fragmentation is low, *MTS Write* makes the amount of the data fragments grow rapidly, but as the degree of the fragmentation is gradually increased, this technology increases the out-of-place update threshold to make more data be written in an in-place manner, so that the growth rate of the data fragments in the system slows down. Finally, the number of fragments in the system is 109970. Compared to the default append-only logging technology, the number of fragments caused by *MTS Write* is reduced by 39.2%.

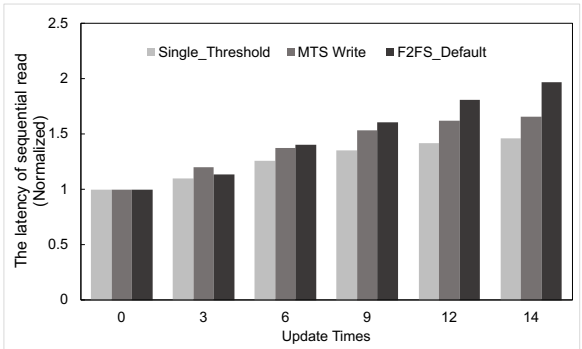


**Figure 10: Normalized sequential read latency.**

Figure 10 shows the performance of the sequential read when using different synchronous write technology. The Single-Level Threshold Synchronous Write technology effectively reduces the data fragments and finally the system sequential read performance is improved by 50.6%. In comparison, *MTS Write* alleviates the data fragmentation to a certain extent, and finally achieves a performance improvement of 30.8%.

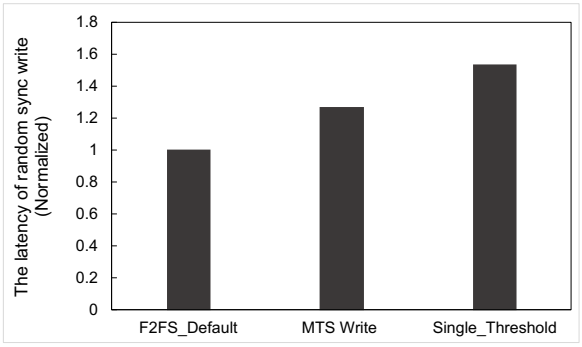


**Figure 11: Normalized synchronous random write latency.**

Figure 11 shows the performance of the synchronous random write when using different synchronous write schemes. Compared with the default append-only logging technology, the Single-Level Threshold Synchronous Write scheme caused the performance reduced by 53.5%. The proposed *MTS Write* can dynamically select the write mode, which has less impact on the performance of synchronous random write, and the performance only reduced by 26.9%.

From the above results, in summary, *MTS Write* can obtain the information of the data fragmentation in real time, and dynamically select the data write mode to control the speed at which fragments are generated, so that it can effectively improve the sequential read performance, and at the same time, take into account the performance of random synchronous write.

## 5.3 Background segment cleaning optimization analysis

In this section, we verified the performance optimization effect of the proposed *High-Frequency BSC* for the background segment cleaning. In addition, we verified that this solution reduces the impact of the background segment cleaning on the life of the flash memory. We design the experiment with the following steps: 1) Format the SSD with the F2FS file system, and mount it to the desktop. 2) Create an initial file. 3) Create 2 processes to read and write the file. 4) Set a period of system idle time to suspend the execution of the processes. 5)When idle time is over, awake the processes to continue to read and write the file. 6) Repeat steps 4 and 5 a total of 5 times. Access the debug file (*/sys/kernel/debug/f2fs/status*) of the F2FS file system, and record the information about the background cleaning.

We used three different schemes *High-Frequency BSC*, the F2FS default background segment cleaning technique, and a *High-Frequency BSC* without the control module to perform the above experiments under different system remaining space.

*5.3.1 Results analysis.* Figure 12 shows the number of times the background segment cleaning triggered when using different background cleaning schemes, where F2FS_default denotes the F2FS file

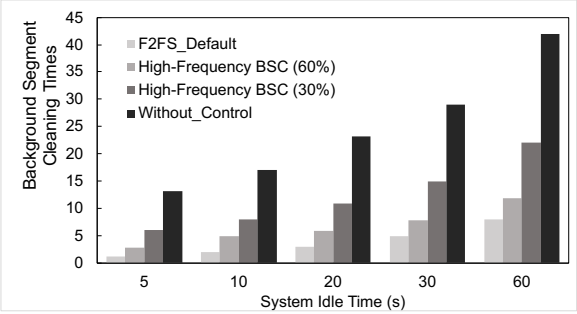


**Figure 12: The number of times the background segement cleaning triggered.**

system default background segment cleaning mechanism, Without_Control denotes *High-Frequency BSC* without the control module, High-Frequency BSC (60%) and High-Frequency BSC (30%) respectively denote the situation of the proposed *High-Frequency BSC* when the remaining space of the system is 60% and 30%. As can be seen from the figure, in a short period of system idle time, compared with the default method, the proposed technique can effectively increase the times of background cleaning triggers and dynamically control the trigger frequency according to the capacity of the remaining space of the system.

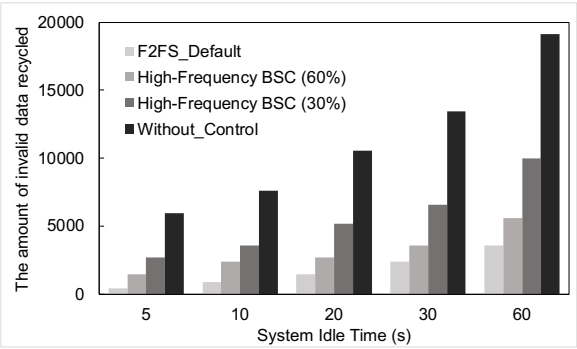


**Figure 13: The amount of invalid data recycled.**

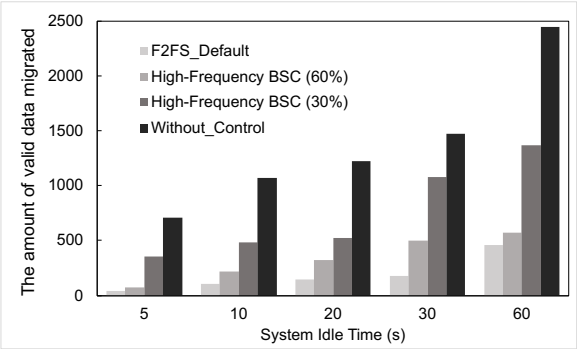


**Figure 14: The amount of valid data migrated.**

Figure 13 and Figure 14 respectively show the amount of invalid data recycled and the amount of valid data migrated when using three different schemes. Compared with the default background cleaning, our proposed scheme reclaims space more effectively, especially when the system idle time is very short (5 seconds, 10 seconds). At the same time, under different remaining system space, compared with the Without_Control scheme, the average data migration caused by the High-Frequency BSC (30%) and High-Frequency BSC (60%) decreased by 76.1% and 44.7%, respectively.

The results show that our proposed *High-Frequency BSC* can improve the performance of the background segment cleaning and take into account the life of the flash memory by reducing the number of data migration.

## 6 CONCLUSION

In this paper, we proposed *MST Write* to dynamically control the generation of the data fragments, which improves the sequential read performance (30.8% compared with the existing scheme) while taking into account the random write performance (only drops by 26.9%). We also suggested *High-Frequency BSC*, which effectively improves the performance of the segment cleaning with minimized impact on the life of the flash memory.

## 7 ACKNOWLEDGEMENT

This work is supported by the Fundamental Research Funds for the NSFC 61772092 and 61572411.

## REFERENCES

[1] Woo Hyun Ahn, Kyungbaek Kim, Yongjin Choi, and Daeyeon Park. 2002. DFS: a de-fragmented file system. In *MSCOTS*. 71–80.
[2] Daniel P Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.".
[3] Intel Corporation. 2017. datasheet of Intel SSD 545s Series. Website. https://ark.intel.com/products/125018/Intel-SSD-545s-Series-128GB-2_5in-SATA-6Gbs-3D2-TLC.
[4] Jace Courville and Feng Chen. 2016. Understanding storage I/O behaviors of mobile applications. In *MSST*. 1–11.
[5] Congming Gao, Yejia Di, Aosong Deng, Duo Liu, Cheng Ji, Jason Chun Xue, and Liang Shi. 2018. F2FS Aware Mapping Cache Design on Solid State Drives. In *2018 IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 31–36.
[6] Hyunho Gwak, Yunji Kang, and Dongkun Shin. 2015. Reducing garbage collection overhead of log-structured file systems with GC journaling. In *ISCE*. 1–2.
[7] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. 2017. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter. In *ATC*. Santa Clara, CA, 759–771.
[8] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones.. In *ATC*. 309–320.
[9] C. Ji, L. Chang, S. S. Hahn, s. Lee, R. Pan, L. Shi, J. Kim, and C. J. Xue. 2018. File Fragmentation in Mobile Devices: Measurement, Evaluation, and Treatment. *IEEE Transactions on Mobile Computing* (2018), 1–1.
[10] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. 2006. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review* 40, 3 (2006), 102–107.
[11] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage.. In *FAST*. 273–286.
[12] Yu Liang, Chenchen Fu, Yajuan Du, Aosong Deng, Mengying Zhao, Liang Shi, and Chun Jason Xue. 2017. An empirical study of F2FS on mobile devices. In *RTCSA*. IEEE, 1–9.
[13] Wolfgang Mauerer. 2010. *Professional Linux kernel architecture.* John Wiley & Sons.
[14] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: random write considered harmful in solid state drives.. In *FAST*, Vol. 12. 1–16.
[15] Dongil Park, Seungyong Cheon, and Youjip Won. 2015. Suspend-aware segment cleaning in log-structured file system. Hotstorage.
[16] Jonggyu Park, Dong Hyun Kang, and Young Ik Eom. 2016. File Defragmentation Scheme for a Log-Structured File System. In *APsys*. 19.
[17] Mendel Rosenblum and John K Ousterhout. 1992. The design and implementation of a log-structured file system. *TOCS* 10, 1 (1992), 26–52.
[18] Takashi Sato. 2007. ext4 online defragmentation. In *Proceedings of the Linux Symposium*, Vol. 2. Citeseer, 179–86.