iDiscard: Enhanced `Discard()` Scheme for Flash Storage Devices

Dong Hyun Kang
College of Info. and Comm. Engineering
Sungkyunkwan University
Suwon, South Korea
kkangsu@skku.edu

Young Ik Eom
College of Software
Sungkyunkwan University
Suwon, South Korea
yieom@skku.edu

***Abstract*—NAND flash storage devices have become one of the fundamental building blocks in cloud computing systems because they provide fast responsiveness and low-power consumption compared with HDDs. Unfortunately, flash storage devices still suffer from the *semantic gap* between the file system and the flash storage device. In this paper, we first answer a simple question: *Is `Discard()` command still a research issue?* We then propose an enhanced `Discard()` scheme, called *iDiscard*, to completely bridge the *semantic gap* between the current ext4 file system and flash storage devices. The key idea is that *iDiscard* dynamically invalidates flash pages mapped to the obviated inode blocks of the file system. For evaluation, we implemented *iDiscard* on the current ext4 file system and then compared it on the Disksim simulator and a real SSD, Samsung 850 PRO SSD. Our experimental results clearly show that *iDiscard* efficiently reduces the number of pages in the flash storage, that are valid but contain metadata for removed files, by up to 5.4x compared with existing ext4 file system.**

***Keywords*-Ext4 file system; flash storage device; discard command;**

I. Introduction

Nowadays, cloud computing is one of the popular computing models and its scale is becoming very large and complex [1]–[5]. Along with this contemporary trend, various applications (*e.g.*, database) are moving to the cloud systems that have big data in their storage. In the cloud systems, since the storage performance becomes more and more important for efficient operation of the systems, NAND flash storage devices, such as SSDs and NVMe SSDs, are commonly used as the main storage media of the systems [1], [2], [5]. Unfortunately, the NAND flash storage devices are still considered as a *black box* from the point of view of the local file system of the cloud environments even though the technologies for the flash storage device have rapidly been improved in terms of hardware and software [6]–[10].

The `Discard()` command was designed to mitigate the *semantic gap* between the file system and the underlying flash storage device. The command transfers software hints from the file system to the flash storage device when a file on the file system is deleted [11], [12]. For example, if a user deletes a file on the ext4 file system, ext4 deletes the data blocks of the file on the file system by using the `Unlink()` system call and then it reclaims the space used by the file. However, the flash pages related to the obviated file on the file system continuously reside in the flash storage device as valid state (referred to *ghost pages* in the rest of this paper) after completing the delete operation. Unfortunately, these *ghost pages* negatively affect the performance and endurance of the flash storage device because they should be copied to another flash page during the *garbage collection* (GC) of the flash storage deivce [12]–[14]. In order to prevent these unnecessary copy operations, the `Discard()` command gives a hint "*page invalidation*" to the underlying flash storage deivce. If the flash translation layer (FTL) on the flash storage device receives this hint, the FTL changes the state of corresponding flash pages from valid to invalid. As a result, FTL can avoid unnecessary copy operations during the GC. However, despite such benefits of the `Discard()` command, the mechanism using the `Discard()` command still leaves behind that of other flash-friendly mechanisms [11]–[14].

In addition, many current file systems (*e.g.*, ext3 [15], ext4 [16], and F2FS [9]) cannot always take the benefits of the `Discard()` command even though they well provide the `Discard()` command (some file systems call it `Trim()` command). This is because most file systems employ the `Discard()` command only for data blocks in their file systems. Of course, metadata blocks on the file system occupy relatively small portion of the file system space [15]–[17]; however they occasionally lead to higher *write amplification factor* (WAF) [18] because flash pages mapped to the metadata block of the file system can also become *ghost pages*. In this paper, we first focus on the relationship between metadata blocks on the file system and the underlying flash storage device. We then propose an advanced approach, called *iDiscard*, that efficiently employs the `Discard()` command to eliminate the *ghost pages* containing invalidated inode information of the file system. Our contributions can be summarized as follows:

- First, we study the layout of the widely used ext4 file system and analyze the behaviors of the file system to understand the file creation and deletion process. Since we cannot monitor the internal workings of the real flash storage device, we implement the `Discard()` command on the SSD extension of DiskSim simula-

2375-9356/18/$31.00 ©2018 IEEE
DOI 10.1109/BigComp.2018.00060

tor [19] that helps to confirm how many *ghost pages* continuously remain in the underlying NAND flash storage device.

- Second, we design an advanced `Discard()` scheme, called *iDiscard*, to completely bridge the semantic gap between the file system and NAND flash storage device and implement it on the ext4 file system. The key design challenge comes upon us when the file system sends the `Discard()` command for *ghost pages* to the underlying flash storage device. To solve this, we utilize the characteristics of the ext4 layout [20]. To the best of our knowledge, this work is the first study to dynamically issue the `Discard()` command for the metadata blocks of the file system.
- Third, we evaluate *iDiscard* on DiskSim simulator to clearly confirm the effectiveness of our approach. Also, we compare the WAF value of *iDiscard* with that of the traditional ext4 on a real SSD, which is one of the latest commercial SSDs (Samsung 850 PRO). Our evaluation gives a meaningful result that *iDiscard* reduces the number of valid pages on the file system by up to 5.4x and reduces the WAF value by up to 2.3% on the real SSD.

The rest of the paper is organized as follows. We first describe the background and motivation for this research in Section II. Then, we present the design of *iDiscard* in Section III in detail, and evaluate our scheme with the traditional ext4 file system in Section IV. Section V briefly discusses the related work. Finally, we outline our conclusions and possible extensions in Section VI.

II. Background and Motivation

In order to better understand our scheme, we briefly introduce layout and internal behaviors of the ext4 file system. Then, we show how many *ghost pages* are left within the flash storage device after completing delete operations.

A. File Creation and Deletion on Ext4 File System

Ext4 file system is widely being used in various environments, including enterprise, desktop, and mobile environments. The ext4 file system divides the whole space of storage into block groups, and each block group has its own layout including block group descriptor, super block, data block bitmap, inode bitmap, inode table, and data block, to efficiently maintain file metadata and file data [20]. Two bitmap blocks of a block group are responsible for indicating which inodes or data blocks are available. The inode table consists of consecutive 4KB inode blocks and an inode block contains 16 inodes, assuming that the inode size is 256B. Each inode records metadata of a file and it is used when ext4 file system accesses the contents of file data blocks.

Figure 1 illustrates a detailed example of file creation, based on the traditional ext4 file system. If a 4KB file is created, the ext4 file system finds a free data block by

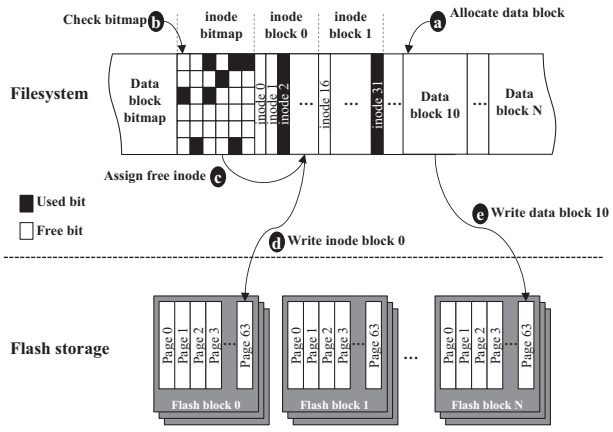

Figure 1: An example of file creation on existing ext4 filesystem

referencing the data block bitmap and then allocates it to store the file contents (Figure 1a). At the same time, the ext4 checks the inode bitmap to find a free inode and then it assigns the inode to the new file (Figure 1b–c). Next, the ext4 records the file metadata (*e.g.*, file size, timestamps, and pointers to data blocks) into the assigned inode to handle the information on the file. Finally, the updated data block and the inode block which contains the assigned inode, are simultaneously reflected to the underlying flash storage device by kernel events, such as the `fsync()` system call or time period triggering (Figure 1d–e). As a result, two flash pages can be allocated inside the underlying flash storage device for the 4KB file: one for the data block and the other for the inode block.

On the other hand, when the created file is erased with an `Unlink()` system call (*i.e.*, link count of the file reaches zero), the ext4 file system only invalidates one flash page mapped to the data block of the file system. The major reason is that an inode block of the ext4 is composed of 16 inodes and some inodes in the inode block may keep valid metadata of other files. However, current ext4 file system never gives a hint, "*page invalidation*", to the underlying flash storage device even though all inodes belonging to the same inode block have invalid metadata of the files. As a result, flash pages mapped to the inode blocks of the file system continually remain in the flash storage device with valid state (*i.e.*, they become the *ghost pages*). Unfortunately, these *ghost pages* can lead to higher WAF value and performance degradation of the flash storage device because they should be copied to other flash pages during the GC on the flash storage device even though the pages are never reused.

B. Ghost Pages

In order to better understand the behaviors of the `Discard()` command in both the file system and the underlying flash storage deivce, we implemented the `Discard()`

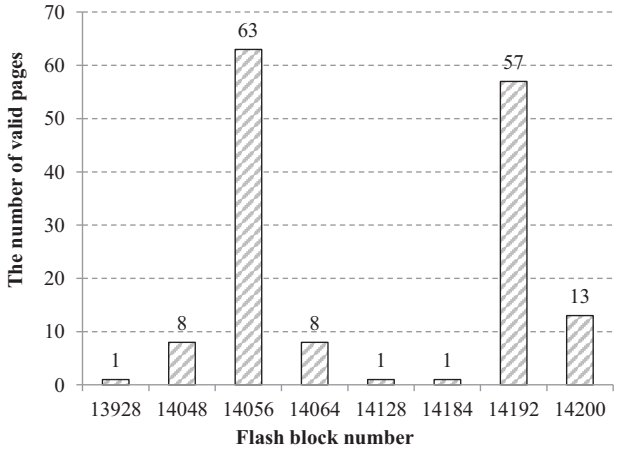

Figure 2: The utilization of each flash block on Disksim simulator

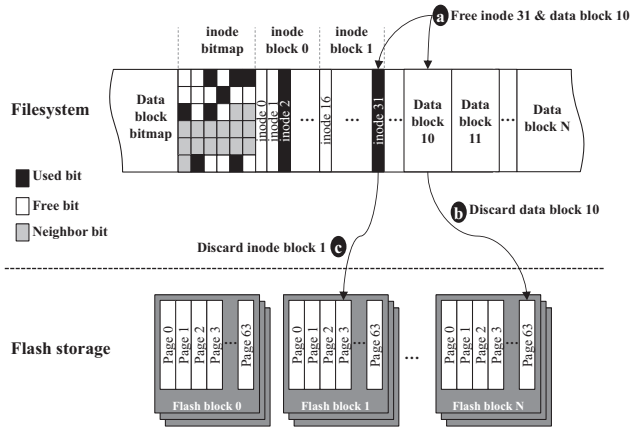

Figure 3: An example of *iDiscard* with two `Discard()` commands

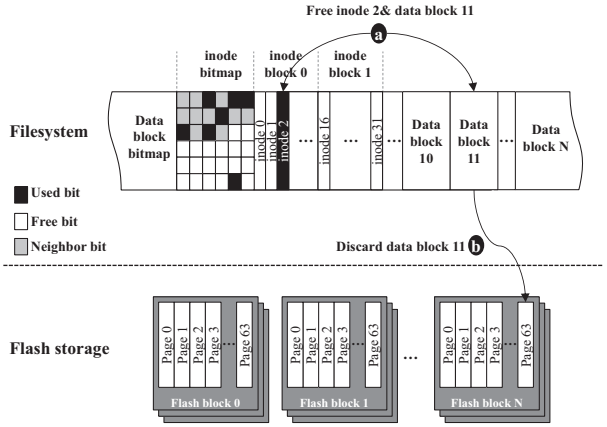

Figure 4: An example of *iDiscard* with one `Discard()` command

command on the SSD extension of DiskSim simulator [19]. To measure how many valid pages are unnecessarily left within the flash storage device, we first collected I/O traces by running the FIO benchmark and then we used the collected I/O traces as the input of the DiskSim simulator (we will describe the evaluation method in detail in Section IV).

Figure 2 shows the number of valid pages belonging to each NAND flash block after creating 2000 files and deleting them one by one (we showed only the flash blocks that contain valid pages). Unsurprisingly, the ext4 maintains a total of 152 valid pages within the flash storage device even though all files on the file system were completely erased. Since some valid pages can be used to keep the file system metadata, we also observed whether a valid page contains the inode block of the file system or not and then confirmed the fact that most valid pages keep an obviated inode information (*i.e.*, *ghost pages*). This observation strongly motivates us to propose the *iDiscard* scheme.

III. Design of *iDiscard*

As shown in Figure 2, the current ext4 file system still has the semantic gap between the file system and the flash storage device. In this section, we present an enhanced `Discard()` scheme, called *iDiscard*, that fully eliminates the *ghost pages* by sending the `Discard()` commands for the inode blocks of the ext4 file system. We design *iDiscard* to achieve the following goals:

- Provide enhanced `Discard()` command on the ext4 file system to completely bridge the semantic gap between the file system and the underlying flash storage device.
- Reduce the WAF value of the flash storage device to improve the performance and endurance of the NAND flash storage.

The mechanism of *iDiscard* is very simple, and it efficiently utilizes the characteristics of the ext4 layout. As mentioned in Section II, each inode block of the ext4 file system contains 16 inodes. Accordingly, we should not send the `Discard()` command to the underlying flash storage device when a file is deleted. This is because the inode block, which contains the inode assigned to the obviated file, may have other valid inodes. In the traditional ext4 file system, if a file is erased with the `Unlink()` system call, it first frees both the inode inside an inode block and data blocks related to the file and then updates its bitmap blocks, inode bitmap and data block bitmap, to clear the corresponding bits. Therefore, whenever a bit on inode bitmap is updated with the `Unlink()` system call, *iDiscard* investigates its neighbor 16 bits that identify 16 inodes belonging to the same inode block. If all neighbors have clean bit, *iDiscard* issues the `Discard()` command to invalidate the flash page mapped to the inode block of the file system. Otherwise,

Table I: The internal parameters of Disksim simulation environment

Parameter	Value
Page size	4 KB
Block size	256 KB (64 pages)
Page read latency	25 us
Page write latency	200 us
Block erase latency	1.5 ms
Overprovisioning space	15%

iDiscard follows the default rules of the ext4 file system.

For example, in Figure 3, if the 4KB file with inode number 31 is erased (Figure 3a), *iDiscard* sends two `Discard()` commands, one for the data block (Figure 3b) and the other for the inode block (Figure 3c), to the underlying flash storage device. This is because all neighbor bits in the bitmap block associated with the inode block 1 (as shown in grey boxes in Figure 3) indicate that all inodes have been erased or not been written so far. On the other hand, when the 4KB file with inode number 2 is deleted (Figure 4a), *iDiscard* sends just one `Discard()` command for its data block (Figure 4b) since some inodes in the same inode block maintain valid metadata (as shown in black boxes in Figure 4). In this way, *iDiscard* completely removes the *ghost pages* generated by the inode blocks of the ext4 file system without any corruptions of the file system and thus reduces the WAF value inside the flash storage device.

IV. Evaluation

In this section, we describe the experimental setup for evaluation. We also show our evaluation results against existing ext4 file system.

A. Experimental Setup

For our evaluation, we implemented a prototype of *iDiscard* in the Linux kernel version 3.18.25 by modifying the ext4 file system. All experimental results were collected on a Linux machine equipped with 3.5GHz Intel Core i7 CPU and 8GB DRAM running the Ubuntu 12.04 (64 bit). We also used the FIO benchmark (version 2.2.12) [21] to simulate a heavy I/O workload: the benchmark was configured to generate random write workloads with default parameters (*e.g.*, libaio engine, 32 iodepth, 4KB block, and `O_DIRECT`). In this paper, we only compare the *iDiscard* with the conventional ext4 file system which enabled the discard option.

B. Results on Disksim Simulator

To deeply understand the internal behaviors of the flash storage device, we modified the SSD extension of DiskSim simulator [19] to make it support the `Discard()` command. The DiskSim was configured with a page-mapped FTL and default parameters (Table I). To obtain I/O traces, we used blktrace with the following steps:

1) We format the storage and turn off the journaling of the ext4 to avoid interference from other components.
2) We then create 2000 files with 4KB size by running FIO benchmark.
3) We delete the 2000 files on the file system one by one.

Table II summarizes the characteristics of the collected I/O traces. As presented in Table II, *iDiscard* issues more `Discard()` commands to the underlying flash storage device compared to existing ext4 because *iDiscard* issues `Discard()` commands for *ghost pages*. To evaluate the effectiveness of *iDiscard*, we employed these I/O traces as the input of DiskSim simulator. Figure 5 shows how many valid pages still reside in each flash block after running DiskSim simulator. As shown in Figure 5, *iDiscard* reduces the number of valid pages by up to 5.4x by eliminating 124 *ghost pages* related to inode blocks of the file system. In addition, *iDiscard* can help reuse three flash blocks, 14128, 14184, and 14192, without the garbage collection on the flash storage with DiskSim because all flash pages belonging to the blocks have already been invalidated by the `Discard()` command. These results clearly confirm that *iDiscard* can reduce the WAF value on the flash storage device.

Table II: The characteristics of collected I/O traces on a machine with 4 CPU cores, 8GB DRAM, and Samsung 850 PRO SSD

	ext4 with `Discard()` on	ext4 with *iDiscard*
# of read requests	81	81
# of write requests	2165	2165
# of discard requests	1998	2122

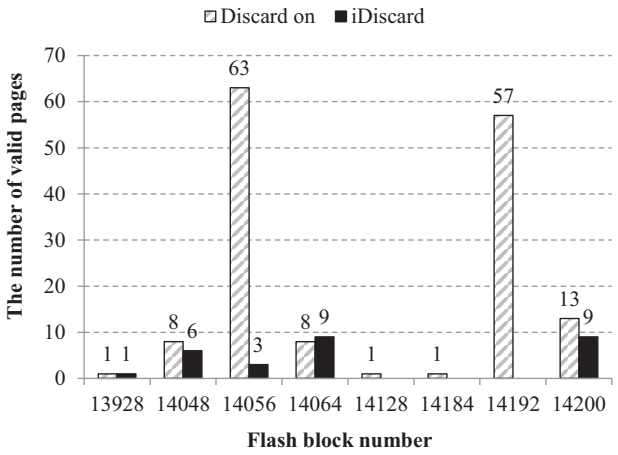

Figure 5: Comparison between `Discard()` on and *iDiscard* in terms of the utilization of valid flash blocks

C. Results on Real SSD

To verify that the expected WAF reduction can be achieved on real SSDs, we evaluated *iDiscard* on the Samsung 850 PRO SSD with 256GB of storage capacity [22]. Fortunately, since most commercial SSDs well support the S.M.A.R.T (Self-Monitoring, Analysis and Reporting) technology, we can easily obtain the WAF value on the real SSDs [23]. To calculate the WAF value, we periodically collected the wear leveling count (ID #177) and total LBAs written (ID #241) from the Samsung SSD by using the S.M.A.R.T technology while running FIO benchmark [23], [24]. For fair comparison, we executed the FIO benchmark with the following steps:

1) We first format the Samsung SSD with the discard option of the ext4 file system and turn off the journaling of the ext4.
2) We then go through an aging process, which randomly writes 256GB of data at a granularity of 1MB.
3) We format once more to reset the utilization of the file system without the discard option of ext4.
4) We run FIO to create 1MB files until the utilization of the file system is full (*i.e.*, 100% space utilization).
5) We delete all files on the file system one by one.
6) Finally, we jump to step 4 to iterate five runs of experiments.

Figure 6 shows the WAF values calculated by `Discard()` on and *iDiscard*. As we expected, *iDiscard* reduces the WAF value by up to 2.3% on the real SSD, Samsung 850 PRO. This reduction is reasonable because inode blocks on the ext4 file system occupy relatively small portion of the whole file system space. However, Figure 6 clearly confirms that *ghost pages* mapped to the inode blocks of the ext4 sometimes lead to higher WAF value because *iDiscard* maintains lower and more stable WAF values compared to ext4 with `Discard()` on. These results are very important because the WAF value directly impacts on both the performance and endurance of the NAND flash storage device even though there is no significant change.

In summary, our evaluation results with both the Disksim simulator and Samsung 850 PRO SSD showed that the semantic gap between the file system and the underlying flash storage device can negatively impact the performance and endurance of the flash storage device because it can increase the WAF value inside the flash storage device. We further believe that the effectiveness of *iDiscard* can be much higher on the low-end flash storage devices, such as TLC based SSDs.

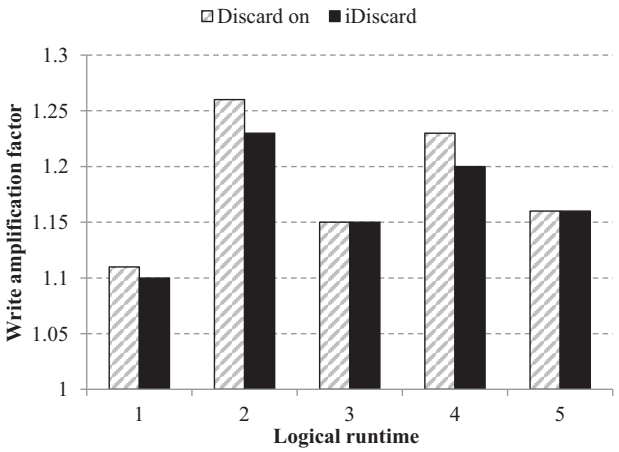

Figure 6: Comparison of the WAF values on a real SSD (Samsung 850 PRO)

V. Related Work

There have been many previous works to mitigate the disadvantages of the NAND flash storage device, such as poor random write latency, limited lifetime, no-overwrite, wear-leveling, and address translation layer (*i.e.*, FTL).

Many researchers focused on the existing IO stacks (*e.g.*, page cache and file system layer) to make them flash friendly [9], [25]–[29]. The traditional strategies for the flash storage devices are to reshape random write patterns into sequential ones. For example, some prior work redesigned the page cache layer that evicts dirty pages in a sequential way by sorting dirty pages based on logical block address (LBA) [25]–[27]. Meanwhile some researchers focused on the file system layer; SFS [28], F2FS [9], and ParaFS [29] are up-to-date flash friendly file systems that are implemented based on the log-structured file system (LFS) to reshape random patterns into sequential ones. In addition, designers of those file systems coordinated activities of the file system to classify the file blocks according to the temperatures of them (*e.g.*, hot, warm, and cold) because the blocks with the same temperature help to relieve the overhead of garbage collection. However, they still leave the *ghost pages* inside the flash storage because they do not fully utilize the `Discard()` command.

So far, several studies have presented the effectiveness of the `Discard()` command on the real NAND flash storage; some researches call it `Trim()` command and the operation of `Trim()` is completely equal to that of `Discard()` in the commercial SSDs [9], [10], [12]–[14]. In 2011, some researchers investigated the relationship between the utilization of the file system and the `Trim()` command and suggested a selective `Trim()` policy to maximize the benefits of the `Trim()` command [13]. Some papers also published system-level analyses. Lee et al. reported that small `Discard()` commands generated by the ext4 file system are harmful for the flash storage devices [9]. Kang et al. clearly verified the effect of `Trim()` command by monitoring the internal workings of the real SSD and presented that the `Trim()` command is very effective for some workloads that organize data in a log-structured manner [10].

Kim et al. analyzed differences between the `Trim()` and `Discard()` command on the real eMMC storage in terms of performance, and also presented the fact that the `Trim()` and `Discard()` commands help improve the random write performance and lifetime of the real eMMC [12].

VI. Conclusion and Future Work

In the cloud system, It is very important to understand internal structures and behaviors of the flash storage device in reducing the response time of various services of the system. In this paper, to understand the internal working of the flash stoage device, we first introduced the relationship between the file system and the underlying flash storage device. Then, we presented an enhanced `Discard()` scheme, called *iDiscard*, that completely bridges the semantic gap between the ext4 file system and the underlying flash-based storage device. By invalidating the *ghost pages*, the proposed scheme efficiently reduces the number of valid pages copied during GC. We implemented a prototype of the *iDiscard* over the current Ext4 file system by fully utilizing the layout of the file system. Our experimental results clearly confirm that *iDiscard* has promising potentials to improve the performance and endurance of the flash storage devices.

As future work, we plan to extend our work in several ways. First, we plan to explore additional ways to optimize the effectiveness of `Discard()` command. For instance, we are going to try to group a number of `Discard()` commands for I/O efficiency. Second, we will examine how *iDiscard* can be applied to other file systems, such as XFS, Btrfs, and F2FS. Finally, we will explore the impact of *iDiscard* with different flash storage devices (*e.g.*, eMMCs, SD Cards, TLC based SSDs, and NVMe SSDs) and a variety of workloads because we believe that the key idea of *iDiscard* can be widely adopted in other environments.

Acknowledgment

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (No. NRF-2015M3C4A7065696). This research was also supported by the MSIT(Ministry of Science and ICT), Korea, under the SW Starlab support program(IITP-2015-0-00284) supervised by the IITP(Institute for Information & Communications Technology Promotion).

References

[1] J. Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon, "Gecko: Contention-Oblivious Disk Arrays for Cloud Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'13)*, 2013, pp. 285–298.

[2] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and C. E. Schrock, "Janus: Optimal Flash Provisioning for Cloud Storage Workloads," in *Proceedings of the USENIX Annual Technical Confeerence (ATC'13)*, 2013, pp. 91–102.

[3] Y. Zhang, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "ViewBox: Integrating Local File Systems with Cloud Storage Services," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'14)*, 2014, pp. 119–132.

[4] M. G. Khatib and Z. Bandic, "PCAP: Performance-aware Power Capping for the Disk Drive in the Cloud," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*, 2016, pp. 227–240.

[5] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao, "CloudCache: On-demand Flash Cache Management for Cloud Computing," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*, 2016, pp. 355–369.

[6] NVM Express Overview. [Online]. Available: http://www.nvmexpress.org/about/nvm-express-overview

[7] Intel® 3D NAND Technology Transforms the Economics of Storage. [Online]. Available: https://www.intel.com/content/www/us/en/solid-state-drives/3d-nand-technology-animation.html

[8] Samsung V-NAND Technology. [Online]. Available: http://www.samsung.com/semiconductor/minisite/ssd/v-nand/technology.html

[9] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'15)*, 2015, pp. 273–286.

[10] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The Multi-streamed Solid-State Drive," in *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*, 2014, pp. 1–5.

[11] Trim Command. [Online]. Available: https://en.wikipedia.org/wiki/Trim_(computing)

[12] B. Kim, D. H. Kang, C. Min, and Y. I. Eom, "Understanding Implications of Trim, Discard, and Background Command for eMMC Storage Device," in *Proceedings of IEEE Global Conference on Consumer Electronics (GCCE'14)*, 2014, pp. 709–710.

[13] C. Hyun, J. Choi, D. Lee, and S. H. Noh, "To TRIM or not to TRIM: Judicious Triming for Solid State Drives," in *Poster presentation in the ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011, pp. 1–2.

[14] K. Kwon, D. H. Kang, J. Park, and Y. I. Eom, "An Advanced TRIM Command for Extending Lifetime of TLC NAND Flash-based Storage," in *Proceedings of IEEE International Conference on Consumer Electronics (ICCE'17)*, 2017, pp. 424–425.

[15] Ext3 Filesystem. [Online]. Available: https://en.wikipedia.org/wiki/Ext3

[16] Ext4 Filesystem. [Online]. Available: https://en.wikipedia.org/wiki/Ext4

[17] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[18] Write Amplification. [Online]. Available: https://en.wikipedia.org/wiki/Write_amplification

[19] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The Disksim Simulation Environment Version 4.0 Reference Manual (CMU-PDL-08-101)," *Parallel Data Laboratory*, 2008.

[20] Ext4 Layout. [Online]. Available: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout

[21] FIO - Flexible I/O Tester Synthetic Benchmark. [Online]. Available: http://git.kernel.dk/?p=fio.git

[22] Samsung 850 PRO SSD. [Online]. Available: http://www.samsung.com/semiconductor/minisite/ssd/product/consumer/850pro.html

[23] S.M.A.R.T. [Online]. Available: https://en.wikipedia.org/wiki/S.M.A.R.T.

[24] W.-H. Kang, S.-W. Lee, and B. Moon, "Flash as Cache Extension for Online Transactional Workloads," *The International Journal on Very Large Data Bases*, vol. 25, no. 5, pp. 673–694, 2016.

[25] B. Debnath, S. Subramanya, D. Du, and D. J. Lilja, "Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks," in *Proceedings of the IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'09)*, 2009, pp. 1–9.

[26] H. Kim, M. Ryu, and U. Ramachandran, "What is a Good Buffer Cache Replacement Scheme for Mobile Flash Storage?" in *Proceedings of the ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*, 2012, pp. 235–246.

[27] D. H. Kang, C. Min, and Y. I. Eom, "An Efficient Buffer Replacement Algorithm for NAND Flash Storage Devices," in *Proceedings of the IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'14)*, 2014, pp. 239–248.

[28] 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 the USENIX conference on File and Storage Technologies (FAST'12)*, 2012, pp. 1–12.

[29] J. Zhang, J. Shu, and Y. Lu, "ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices," in *Proceedings of the USENIX Annual Technical Confeerence (ATC'16)*, 2016, pp. 87–100.