DISS: A Novel Data Invalidation Scheme for Swap-Data on Flash Storage Systems

Dingcui Yu*,1, Longfei Luo*,1, Han Wang1, Yina Lv2, ✉Liang Shi1
1College of Computer Science and Technology, East China Normal University, Shanghai, China
2Department of Computer Science, City University of Hong Kong, Hong Kong, China

Abstract

Storage swapping has been a critical technique used to relieve memory pressure and improve user experience. However, it generates lots of data writes in flash storage, deteriorating the lifetime and performance. In this paper, inspired by empirical studies on swap data access characteristics, we propose a novel data invalidation scheme, namely DISS, which includes two methods. First, a cross-layer swap-data invalidation method is proposed to invalidate swapped-in data at a low cost. Second, a swap data separation method is proposed to schedule swap data and file-backed data into different places. Experimental results show that DISS achieves encouraging flash lifetime and performance optimization.

Keywords

Swap, Trim, Lifetime, Flash Storage

ACM Reference Format:
Dingcui Yu*,1, Longfei Luo*,1, Han Wang1, Yina Lv2, ✉Liang Shi1, 1College of Computer Science and Technology, East China Normal University, Shanghai, China, 2Department of Computer Science, City University of Hong Kong, Hong Kong, China, . 2025. DISS: A Novel Data Invalidation Scheme for Swap-Data on Flash Storage Systems. In *30th Asia and South Pacific Design Automation Conference (ASPDAC '25), January 20–23, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3658617.3697550

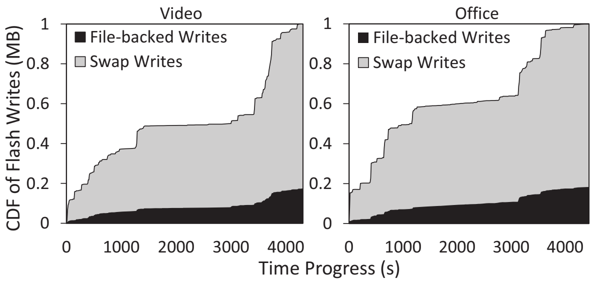

Figure 1: CDF for swap and file-backed writes

1 Introduction

Flash storage devices such as embedded multi-media card (eMMC) and universal flash storage (UFS) have been widely used as consumer devices due to their high performance, large capacity, and low price [1]. At the same time, applications nowadays need more memory resources to implement powerful functions. However, many budget phones and smart TVs are only equipped with 2GB or 3GB of main memory [2][3][4]. If some large applications run on these devices, the memory would be too small to do the execution efficiently and normally. To reduce the memory pressure for normal system operations, storage swapping is widely used in existing operating systems [5][6][7][8]. The basic idea of storage swapping is to extend the memory by using part of the storage space. When memory space is scarce, the swapping will write some data into storage. And when they are needed, these data will be read back to memory. The swapping technique can reduce the possibility of sudden killing for applications, leading to improvement of user experience.

Some studies have proven that it is much faster to recall applications with part of their pages swapped out to flash storage, compared to recalling applications with their processes killed [9]. And in the community, the swapping technique has been widely studied to further improve the user experience [5][6][7][8]. For example, Zhu et al. [6] can reduce the application launch latency by 30% with a prediction-based process-level swap mechanism.

However, as a kind of erase-before-write non-volatile storage, flash storage withstands only a limited number of writes. Enabling swapping would significantly increase the host writes to flash storage and will have a significant impact on the flash lifetime and performance. As shown in Figure 1, two workloads introduced in Section 4.1 are evaluated to show their writes from hosts, including swap writes and file-backed writes. The results show that swap writes will be the major part, which will significantly impact the lifetime and performance of flash storage. Previously, to avoid the influence of swapping on device lifetime, data compression techniques [9][10][11] and data de-duplication techniques [11][12] have been proposed. All of these works focus on reducing the number of host writes to the storage. However, with the limited main memory, there are still a lot of writes to the storage. At the same time, little work pays attention to the characteristics of the swap data in the storage. This work will propose a novel scheme to optimize swap writes-induced lifetime and performance problems.

By analyzing the access characteristics of the swap data, we find that most of the data in the storage are updated by swapping out before swapping in. This finding motivates us that **when the data are swapped in, most of them can be deleted in storage since these data would not be used again.** Based on this

*These authors contributed equally to this work. This work is supported by the NSFC (62072177), and Shanghai Science and Technology Project (22QA1403300). The corresponding author is Liang Shi (shi.liang.hk@gmail.com).

motivation, a straightforward method is to call existing TRIM commands to trim the swapped-in data. We call this simple solution as swap trim (STRIM) in this work. TRIM is a widely used command [13][14][15][16][17], which is issued by the operating system (OS) for the deleted files, and the device controller recognizes and invalidates the related data. In this case, when performing garbage collection (GC) inside storage, the expired data would not be migrated to new blocks. With this approach, the performance and lifetime of storage can be well improved. However, there are two problems for STRIM. First, the system may crash when accessing the swapped-in data which is clean and directly released in memory. Second, the improvement is limited by the high execution cost of TRIM commands [18][19][20].

To solve these issues, we propose a novel data invalidation scheme for swap data, namely DISS. DISS comprises two components, including a cross-layer swap data invalidation scheme (CL-SDI) and a swap data separation scheme (SDS). First, CL-SDI is proposed to invalidate the swapped-in data during reading them to minimize execution costs. To support CL-SDI, a method called swap data recognition is proposed to recognize swap data in the storage controller by making use of the swap partition information with a low overhead. Unlike STRIM, CL-SDI does not require additional commands, and data invalidation can be executed immediately. Since some operations for data invalidation can be merged with the reading process, data invalidation costs will be significantly reduced. Furthermore, for the clean data invalidation, a clean data handler is integrated into CL-SDI to avoid the system crash. Second, since the access characteristics of swap data and file-backed data are different, which is made more significant by CL-SDI, SDS is proposed to isolate them in flash storage to further reduce GC costs. To the best of our knowledge, this is the first work to invalidate swap data in flash-based storage systems. Experimental results show that the proposed scheme can reduce writes and improve performance significantly compared to the state-of-the-art.

2 Background and Motivation

2.1 Flash-based Swap Storage Systems

With applications becoming larger and more versatile, more memory resources are needed to run applications' processes, and main memory has been a critical resource for the overall system. In this case, many processes that need user interaction or system functionality would be killed evidently [9]. To tame the memory-tight scenario, storage swap, a memory-extending technique, is proposed and widely studied for flash storage systems [21][5][6][7].

Figure 2 shows the process of a flash-based swap system. Specifically, when the free space of memory is lower than a watermark, a swap process named *kswapd* will be wakened up. After checking the data is the swap data using the function *PageAnon()*, this data will be written to the storage for memory reclaiming if the *PG_dirty* flag of the page frame is set. This process is called swap-out. When the swapped-out data are requested again, they will be read back to memory. This process is called swap-in. In this case, the flash storage can pretend to be part of memory space via the swapping. This paper divides swap-in into two categories. If the swap-in occurs after the swap data in the storage is updated by swap-out, it is called a *normal-swap-in*. If the swap data in the storage is swapped in twice continuously without being updated, it is called a *swap-in-and-in*.

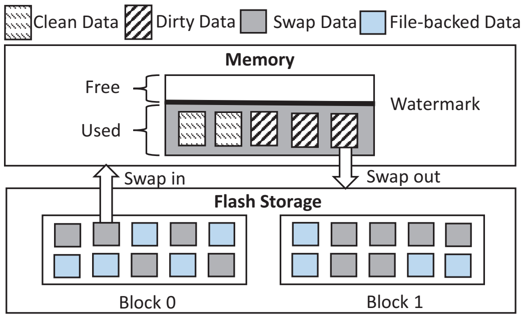

Figure 2: Flash-based swap system

2.2 Data Invalidation for Flash Storage

To invalidate data in flash storage, TRIM has been widely studied and adopted [13][16][15][17]. TRIM is a command dedicated to flash storage [13], which is used to notify the flash storage which data is invalid from the OS. The process of TRIM is presented in Figure 3. First, the OS will generate and issue a TRIM command for invalid data (❶), which comprises the start logical block address (LBA) and the size. Second, when the storage controller processes the TRIM command, the corresponding mapping table entry (MTE) will be loaded into the on-chip buffer (❷). The MTE is used to build the connection between the LBA issued from the OS and the physical block address (PBA) where the data are stored in. As a result, it can be used to determine whether the data is valid. Finally, the MTE is modified to mark the data as invalid (❸). In flash storage, the garbage collection process (GC) is used to reclaim space. Specifically, during GC, a victim block will be selected and the valid data in it will be moved to another free block. For example, Block 1 will be selected based on the greedy policy [22] and one page will be moved. Then the victim block is erased to reclaim space and allow data to be written to it. Since TRIM can decrease the number of valid data stored in the storage, GC-induced writes can be reduced.

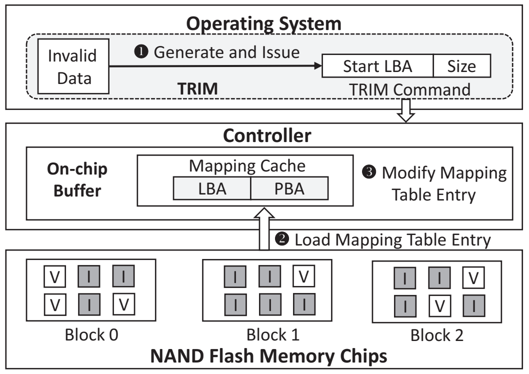

Figure 3: The process and influence of TRIM. Shadowed squares (marked with "I") in the Blocks are invalid data while the others (marked with "V") are valid ones.

2.3 Motivation

To optimize device lifetime and performance in flash-based swap systems, we first explore the access characteristics of swap data. Some real workloads are analyzed, which are traced in four scenarios, including Video, Office, Game, and Social. During each workload, four behaviors are constantly switching, including watching a video (V), editing a powerpoint (O), playing a game (G), and browsing some websites (W). The duration and sequence of four behaviors in them are different as shown in Table 2, and the details of these workloads will be described in Section 4. The number of the *normal-swap-in* and *swap-in-and-in* are collected, and shown in Figure 4. The statistical results show that the *normal-swap-in* takes a large proportion for the four workloads, where the proportion is around 94.5% in all cases, while only a little part of data are *swap-in-and-in*. This indicates that most of the swap data are modified in memory and then swapped out to release the memory space rather than being directly released. This is because the swap data mainly includes heap and stack data which are generated and used dynamically by the programmers during the running of the program. They are generally modified to suit the operation of the program. **This observation implies that most of the swapped-in data stored in the flash devices can be directly invalidated.**

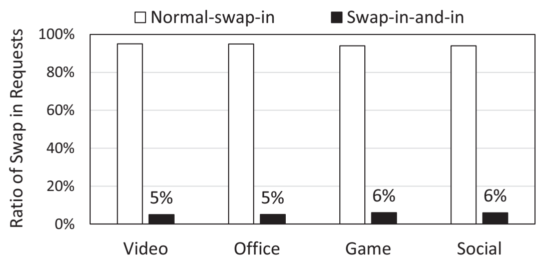

Figure 4: The ratio of normal-swap-in and swap-in-and-in

Based on the above finding, a simple solution is to call the existing TRIM command to invalidate the swapped-in data. We call this method as swap trim (STRIM) in this paper. When OS swaps in data, it will generate a TRIM command to invalidate this data. When the storage controller handles this TRIM command, it will load the corresponding MTE to the mapping cache in the on-chip buffer. Then the MTE will be modified to indicate that this data is invalid data. During the GC process, the invalid data is not required to be moved. The lifetime and performance will be improved. However, this simple scheme has two critical issues. First, the system may crash if all swapped-in data are directly released. This happens because there still exists some data that are accessed belong to the case *swap-in-and-in*. These data are clean during memory reclamation. So they will be evicted directly without being swapped out. If these data are invalidated, when they are required again, the OS will not be able to get the data and the system may crash. Second, the execution costs of the TRIM commands are high enough to block user requests [18][19][20]. The execution costs consist of three parts as shown in Figure 3: ❶ generate and transfer commands, ❷ load MTEs, and ❸ modify MTEs. Among them, ❶ and ❸ are simple operations, and the cost of ❷ is the major cost. Although STRIM can reduce overhead by merging LBA continuous TRIM commands [13], the improvement is still limited according to our experiments. This is because merging TRIM commands will delay the execution of TRIM commands. During the postponement, GC may be activated and swapped-in data are required to be moved. Based on the above finding and issues, we propose a novel data invalidation scheme for the swap storage system in this paper.

3 Design of DISS

3.1 Overview

To address these issues, a novel data invalidation scheme for swap data is proposed, namely DISS. The overview of DISS is shown in Figure 5. There are two components: a cross-layer swap data invalidation scheme (CL-SDI) and a swap data separation scheme (SDS). CL-SDI is responsible for invalidating swapped-in data at a low cost and avoiding the system crashing problem caused by *swap-in-and-in*. Additionally, SDS is proposed to store swap data and file-backed data in different blocks due to their differing access characteristics, which are further differentiated through CL-SDI.

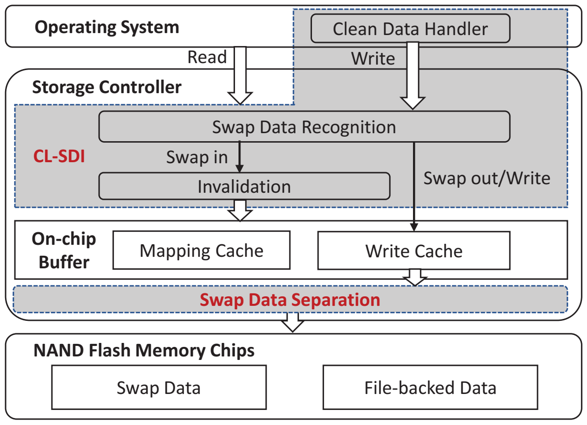

Figure 5: The overview of DISS

3.2 Cross-layer Swap Data Invalidation

To minimize the execution costs of data invalidation, CL-SDI is proposed. The basic idea is to invalidate the swapped-in data during reading them. The architecture is shown in Figure 5, which comprises three modules: a swap data recognition scheme (SDR), the invalidation process, and a clean data handler (CDH). CL-SDI is a cross-layer design that spans a range from the operating system layer to the storage layer. In this section, SDR will be introduced first which recognizes the swap data in the device controller to support CL-SDI. Second, the invalidation process of the swapped-in data in CL-SDI will be presented in detail and the improvement it brings will be discussed. Finally, CDH is proposed which avoids the OS accessing invalid data in case of *swap-in-and-in*.

3.2.1 Swap Data Recognition. To introduce how SDR works, we introduce its principle first and then present its implementation. In current operating systems, there is a swap partition that is managed by the file system. If swap is enabled, the swap partition can be created as the swap device. To logically isolate the swap partition and the file-backed partition, their logical block address (LBA) ranges are different. Therefore, when the swap partition is configured in the OS, the corresponding LBA range of the swap partition is also determined. Then, by comparing the LBA of the host commands with the LBA range of the swap partition, the device controller can recognize the swap data. Figure 6 presents the implementation of SDR. At first, the file system creates the swap partition based on the user's requirements. Then a command will be sent from the OS to the flash storage. The command is comprised of two parameters, Starting LBA and Ending LBA of the swap partition. When receiving the command, the storage controller will record the information in the on-chip buffer and persistently store it in the storage. If a read/write request is sent to the device, it is checked whether the requested LBA is within the range of starting and ending LBAs. If so, the request is a swapping request. Otherwise, it is a file-backed request. After the swap requests are recognized, the swap data invalidation process can be performed.

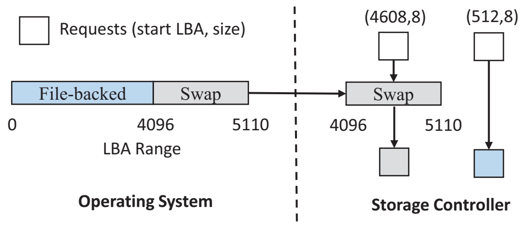

Figure 6: The implementation of Swap Data Recognition

3.2.2 *Execution Process and Benefit Discussion.* To illustrate CL-SDI clearly, Figure 7 presents the process in detail. Firstly, when a read request is issued from the OS (❶), the storage controller will classify it according to the LBA range of the swap partition as introduced in SDR. Second, the corresponding MTE is read into the mapping cache of the on-chip buffer (❷). Third, the required data can be located by the MTE and transferred to the OS (❸). The reading process ends here. Then CL-SDI invalidates swapped-in data by modifying the MTE (❹). At the same time, if the target swapped-in data is found in the write cache (❺), it will be set as clean data to further reduce flash writes. Later, when the GC starts to work, all MTEs of the pages in the victim block will be loaded into the mapping cache. And the swapped-in data will be identified as invalid data. They will be directly erased instead of being copied to another block. Thus, the number of GC-induced writes will be reduced.

The traditional data invalidation overhead is made up of three parts: transfer commands, load mapping table entries, and modify mapping table entries. First, since CL-SDI combines read and invalidate operations into a swap-in request, it does not require any additional command. The transmit command costs can be avoided. Second, CL-SDI invalidates the swapped-in data when they are read. During the reading process, the corresponding MTEs have been loaded into the on-chip buffer to locate data. Therefore, the cost of loading MTEs can be avoided. Third, the only additional cost of CL-SDI is modifying the MTEs (❹) and setting swapped-in data as clean in the write cache (❺). Since they are simple operations, this overhead is negligible. Overall, the execution cost of invalidating swapped-in data is significantly reduced by CL-SDI.

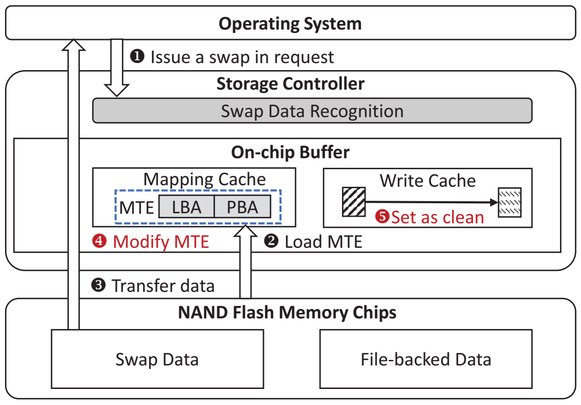

Figure 7: The execution process of CL-SDI

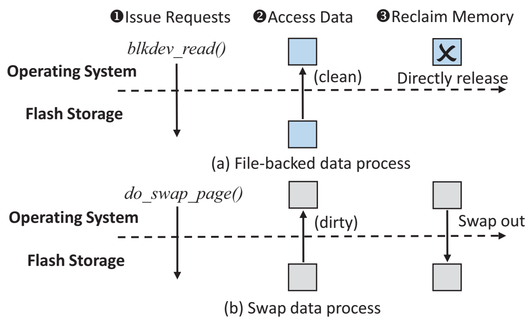

Figure 8: Comparison between file-backed data process and swap data process

3.2.3 *Clean Data Handler.* To avoid system crashes caused by *swap-in-and-in* when employing CL-SDI, CDH is proposed to set all swapped-in data as dirty in memory. This ensures that when they are evicted from memory, they will be swapped out to storage and can be accessed later. Figure 8 shows the implementation of CDH. Since the OS uses different functions to access file-backed data and swap data, it is easy to treat them differently. When the OS uses *blkdev_read()* to access file-backed data, these data are initially clean in memory. As a result, when they are evicted, they will be released directly. In contrast, when the OS accesses swap data using the function *do_swap_page()*, the *Page_Dirty* flag in the corresponding page frame will be set. Consequently, all swapped-in data will be swapped out when they are evicted from memory. It is true that this approach generates additional writes for the clean swapped-in data. However, since the amount of such data is small, as shown in Figure 4, their impact on storage lifetime and performance is minimal, as demonstrated by our experiments.

3.3 Swap Data Separation

Previously, some work stated that the hotness of swap data and file-backed data is different [7]. Additionally, their access characteristics are further differentiated by CL-SDI. With CL-SDI, both swap-in and swap-out processes can generate invalid swap data. The swap-in processes invalidate the swap data with CL-SDI, while the swap-out processes update the swap data to make the expired data invalid. During the application switching, which is the users' common behavior for consumer devices, some swap data are required to launch an application, while some swap data should be swapped out to reclaim memory. As a result, lots of swap-in and swap-out processes will invalidate a large amount of swap data. On the basis of this observation, a swap data separation scheme (SDS) is proposed to further reduce GC-induced writes.

The basic idea of SDS is to separate the swap data and file-backed data into different blocks, which is similar to multi-streaming [23]. Different from multi-streaming, SDS does not rely on labels issued by the OS to identify different categories of data. The process of SDS is shown in Figure 9. Specifically, there are two active blocks for swap data and file-backed data in each plane. First, when a write request is issued from the OS (❶), SDR will identify the category of the data (i.e., swap data or file-backed data) and write it to the write cache of the on-chip buffer (❷). The data in the write cache is managed by the algorithm, e.g., the least recently used algorithm (LRU). When the write cache is full, some data should be written to the storage to reclaim space. The data will then be written to the corresponding active block (❸). For example, the swap data will be written to Block 1 and the file-backed data will be written to Block 2. Then the MTE of the written data will be generated in the mapping cache to build the connection between LBA and PBA (❹). If the active block is filled with data, the device controller will find the next free block as an active block. To be noticed that this new active block should not be the same as the active block of another kind of data. Since a large amount of data are invalidated by switching applications, grouping the swap data into the same block will significantly reduce the number of valid data in these blocks. For instance, all the data in Block 0 are invalidated, and there are no copied data when Block 0 is selected as the victim block during GC. Thus, SDS can further reduce GC-induced writes.

3.4 Overhead Analysis

First, for CL-SDI, SDR only requires a command to initialize the LBA range of the swap partition and several bytes to record it. The

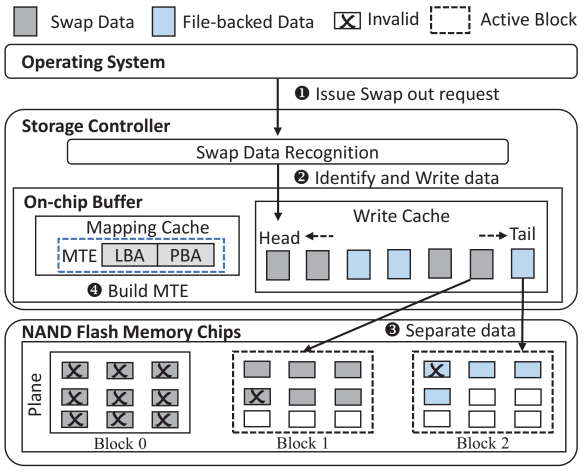

Figure 9: The workflow of the Swap Data Separation

additional operation for CL-SDI is modifying the MTEs and setting the swapped-in data as clean data in the write cache. Since they are simple operations, the overall performance will not be impacted. CDH may introduce some additional writes for the clean swap data. However, since this kind of data only takes 5.5% of the swapped-in data. The benefits of CL-SDI outweigh the overhead, making it an acceptable trade-off. The experimental results in Section 4.2 prove this conclusion. Second, SDS only requires several bytes to record one additional active block per plane, which is negligible. Overall, DISS is a low-cost data invalidation technique.

4 Experiments

4.1 Experimental Setup

Simulator: To evaluate the performance and lifetime of DISS, a simulator is developed by extending SSDSim [24]. In the simulator, several components are added, including all schemes in CL-SDI and SDS. Table 1 shows some key parameters of the evaluation. Note that to evaluate the influence on GC, 75% capacity of flash storage is filled with valid data.

Table 1: Simulation Parameters [25]

Channel	Chip	Die	Plane	Block	Page
8	2	2	4	512	128
tR	**tPROG**	**GC Ratio**	**Aged Ratio**	**DRAM Size**	**SSD Size**
4us	75us	85%	75%	256KB	32GB

Workloads: Four real workloads are used to evaluate the proposed schemes, including Video, Game, Social, and Office. They are collected from a real commercial computer that is equipped with Linux 4.4.0. To simulate the scenario of high memory pressure, the memory size is constrained to 2 GB. Specifically, all four workloads include the operation of watching videos (V), playing games (G), browsing websites (W), and editing powerpoint (O). The difference between them is that the running time and order of the applications are different. To collect traces, blktrace [26] is used. Table 2 shows the application running time and sequence for four workloads. For the first half, e.g., from G to V in Video, we play a game, browse websites, edit a powerpoint, and watch a video in sequence. And each application is launched for the first time. When a new application is launched, other applications that have been launched are running in the background. For the second half, all the applications are running. Among them, one application runs in the foreground, and others run in the background.

Table 2: Workloads Application Switch Sequences

Time(min)	5	5	5	30	5	5	5	5
Video	G	W	O	**V**	O	W	G	V
Game	O	V	W	**G**	W	V	O	G
Social	V	O	G	**W**	G	O	V	W
Office	G	W	V	**O**	V	W	G	O

Table 3 shows the characteristics of the four workloads after tracing, in which the number of reads/writes, average read/write size, and swap read/write ratio are presented. Four workloads show

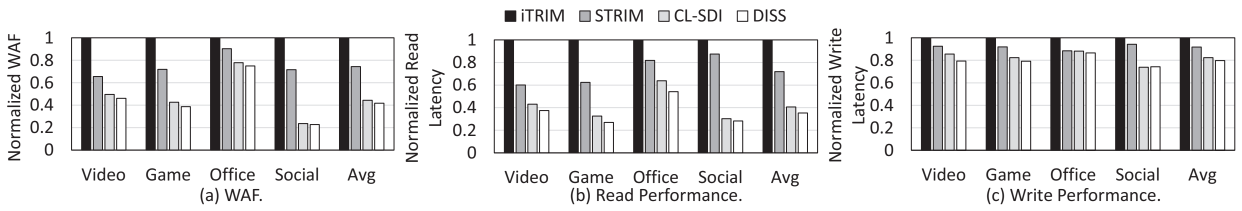

Figure 10: Experimental results which are normalized to those of iTRIM.

Table 3: Workload Characteristics

	Video	Game	Office	Social
# of Reads	267192	273327	222869	310358
# of Writes	13320	14815	13979	17120
Avg.R.size	15.10 KB	14.40 KB	15.35 KB	14.54 KB
Avg.W.size	265.92 KB	264.40 KB	222.90 KB	214.79 KB
Swap.R.Ratio	35.8%	39.6%	36.5%	34.1%
Swap.W.Ratio	82.8%	83.7%	83.0%	81.8%

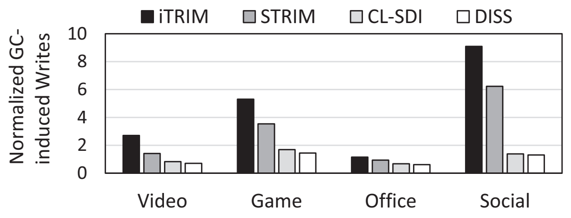

Figure 11: Number of GC-induced writes normalized to that of host writes

similar characteristics. Under high memory pressure, the swap writes of four workloads are significantly more than the user writes.

To evaluate the advantages of the proposed scheme, several metrics are used. They are the write amplification factor (WAF) and read/write latency. WAF is used to evaluate the lifetime of flash storage devices [27], which is equal to $\frac{Flash\_Writes}{Host\_Writes}$.

4.2 Experimental Results

Four schemes are evaluated as follows: *iTRIM* is the state-of-the-art TRIM method [13] that invalidates expired data when the device is idle. *STRIM* is the OS calls TRIM commands to invalidate swapped-in data with the unit of 512 continuous swap-slot (2MB), and all clean swapped-in data in memory will be written to the storage when they are evicted. *CL-SDI* is the proposed scheme that uses the cross-layer swap data invalidation scheme to invalidate swapped-in data. *DISS* is the scheme proposed in this document.

4.2.1 Lifetime Evaluation. Figure 10(a) shows the normalized WAF results for four schemes in all workloads. The number of GC-induced writes is shown in Figure 11, which is normalized to the number of host writes. In the results, several conclusions can be drawn as follows: **First**, invalidating swapped-in data can significantly reduce WAF. *STRIM*, *CL-SDI* and *DISS* reduce the WAF by 25.7%, 55.6% and 58.3% on average compared to *iTRIM*, respectively. Meanwhile, the number of GC-induced writes in the proposed scheme is significantly reduced compared to *iTRIM*. This result implies the high GC efficiency of the data invalidation method can improve flash lifetime. **Second**, the reduction in *CL-SDI* is significantly higher than that of *STRIM*. This proves that unlike STRIM, which should merge the TRIM command to reduce execution costs, CL-SDI can more timely invalidate data by combining invalidating and reading processes. **Third**, the reduction in *DISS* is slightly larger than that of *CL-SDI*. This is because SDS can further improve GC efficiency by grouping the data with similar invalidation time in the same block.

4.2.2 Performance Evaluation. Figures 10(b) and 10(c) show the normalized latency of read and write operations to that of *iTRIM* in all situations. **For read performance**, *STRIM*, *CL-SDI* and *DISS* reduce the latency by 28.1%, 59.5% and 65% on average. The reasons mainly come from three aspects. First, as described in Section 4.2.1, *STRIM* and *CL-SDI* can improve the GC efficiency and reduce the GC frequency by invalidating the swapped-in data. The interference between GC and reads will be less. Second, SDS can further mitigate the interference between GC and reads by separating swap data and filed-backed data. Third, compared to *STRIM*, *CL-SDI* can minimize the execution cost of data invalidation to reduce its interference with reads. **For write performance**, the proposed scheme *DISS* can reduce latency by 21.4% on average compared to *iTRIM*. This implies that reducing GC would also improve the write performance. The improvement is not as good as that for read performance. This is because read requests are more susceptible to interference from the GC process. On the one hand, GC and other requests are more likely to conflict with read requests than with write requests. On the other hand, since the served time of read requests is shorter than that of write requests, the influence of interference between GC and reads is also greater than that of interference between GC and writes. Therefore, all proposed schemes improve read performance more significantly.

5 Conclusion

To address the lifetime and performance reduction in the flash-based swap system, this paper analyzes the access characteristics of the swap data based on collected realistic workloads. We find that most of the swapped-in data in the storage will be updated before they are accessed again. Inspired by this finding, we proposed a novel data invalidation scheme called DISS to invalidate swapped-in data. First, DISS minimizes the invalidation costs and avoids system crashes by a cross-layer swap data invalidation scheme. Second, a swap data separation method is designed to group swap data and file-backed data into different blocks since their access characteristics are different to reduce GC-induced writes. Experimental results show that, compared to the state-of-the-art, DISS can significantly extend the lifetime and improve the performance of flash storage.

References

[1] J. Kim, K. Lim, Y. Jung, and et al., "Alleviating garbage collection interference through spatial separation in all flash arrays," in *ATC*, 2019, pp. 799–812.
[2] J. Maring, "Best phones under $100 in 2021." 2021, https://www.androidcentral.com/best-android-phone-under-100/.
[3] Y. Khetani, "Best smartphone under 100 dollars." 2021, https://blucellphones.us/best-smartphone-under-100-dollars/.
[4] S. Andrews, "Best android tv box 2020: The top android tv devices for plex, kodi, showbox and more." 2020, https://www.expertreviews.co.uk/media-streamers/1405376/best-android-tv-box.
[5] C. Li, L. Shi, Y. Liang, and C. J. Xue, "Seal: User experience-aware two-level swap for mobile devices," *TCAD*, vol. 39, no. 11, pp. 4102–4114, 2020.
[6] X. Zhu, D. Liu, K. Zhong, J. Ren, and T. Li, "Smartswap: High-performance and user experience friendly swapping in mobile systems," in *DAC*, 2017, pp. 1–6.
[7] S. Son, S. Y. Lee, Y. Jin, J. Bae, J. Jeong, T. J. Ham, J. W. Lee, and H. Yoon, "ASAP: Fast mobile application switch via adaptive prepaging," in *ATC*, 2021, pp. 365–380.
[8] W. Guo, K. Chen, H. Feng, Y. Wu, R. Zhang, and W. Zheng, "$mars$: Mobile application relaunching speed-up through flash-aware page swapping," *TC*, vol. 65, no. 3, pp. 916–928, 2015.
[9] Y.-X. Wang, C.-H. Tsai, and L.-P. Chang, "Killing processes or killing flash? escaping from the dilemma using lightweight, compression-aware swap for mobile devices," *TECS*, vol. 20, no. 5s, pp. 1–24, 2021.
[10] C. Ji, L.-P. Chang, L. Shi, C. Gao, C. Wu, Y. Wang, and C. J. Xue, "Lightweight data compression for mobile flash storage," *TECS*, vol. 16, no. 5s, pp. 1–18, 2017.
[11] T. Song, G. Lee, and Y. Kim, "Enhanced flash swap: Using nand flash as a swap device with lifetime control," in *ICCE*, 2019, pp. 1–5.
[12] M.-C. Yen, S.-Y. Chang, and L.-P. Chang, "Lightweight, integrated data deduplication for write stress reduction of mobile flash storage," *TCAD*, vol. 37, no. 11, pp. 2590–2600, 2018.
[13] Y. Liang, C. Ji, C. Fu, R. Ausavarungnirun, Q. Li, R. Pan, S. Chen, L. Shi, T.-W. Kuo, and C. J. Xue, "itrim: I/o-aware trim for improving user experience on mobile devices," *TCAD*, vol. 40, no. 9, pp. 1782–1795, 2021.
[14] G. Kim and D. Shin, "Performance analysis of ssd write using trim in ntfs and ext4," in *ICCIT*, 2011, pp. 422–423.
[15] 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 *ICCE*, 2017, pp. 424–425.
[16] Y. Lee, J.-S. Kim, S.-W. Lee, and S. Maeng, "Zombie chasing: Efficient flash management considering dirty data in the buffer cache," *IEEE TC*, vol. 64, no. 2, pp. 569–581, 2015.
[17] S.-H. Lim and Y.-S. Jeong, "Journaling deduplication with invalidation scheme for flash storage-based smart systems," *JSA*, vol. 60, no. 8, pp. 684–692, 2014.
[18] M. Saxena and M. M. Swift, "FlashVM: Virtual memory management on flash," in *USENIX ATC*, 2010.
[19] B. Kim, D. H. Kang, C. Min, and Y. I. Eom, "Understanding implications of trim, discard, and background command for emmc storage device," in *IEEE GCCE*, 2014, pp. 709–710.
[20] S.-H. Lim and K.-J. Kim, "Empirical inspection of io subsystem for flash storage device at the aspect of discard," in *CES-CUBE*, 2016, pp. 59–63.
[21] S. Jennings, "The zswap compressed swap cache." 2013, https://lwn.net/Articles/537422/.
[22] Y. Yang, V. Misra, and D. Rubenstein, "On the optimality of greedy garbage collection for ssds," *PER*, vol. 43, no. 2, pp. 63–65, 2015.
[23] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *HotStorage*, 2014, pp. 1–5.
[24] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity." ICS, 2011, pp. 96–107.
[25] T. Kouchi, N. Kumazaki, and et al., "13.5 a 128gb 1b/cell 96-word-line-layer 3d flash memory to improve random read latency with tprog=75µs and tr=4µs," in *ISSCC*, 2020, pp. 226–228.
[26] *Blktrace for Linux I/Os*, 2021, https://git.kernel.dk/cgit/blktrace/.
[27] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *SYSTOR*, 2009, pp. 1–9.