F2FS Aware Mapping Cache Design on Solid State Drives

Congming Gao*, Yejia Di*, Aosong Deng*, Duo Liu*, Cheng Ji†, Chun Jason Xue† and Liang Shi**
*College of Computer Science, Chongqing University, Chongqing, China
† Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong

Abstract—**During the past decades, NAND flash memory based SSDs have been widely deployed in personal computers, mobile devices, and cloud systems. In order to maintain the mappings between logical addresses and physical addresses, a built-in RAM called as mapping cache is always implemented within SSDs. However, due to the cost and space limitations of RAM, only limited number of address mappings are supposed to be stored in the mapping cache. The mapping cache has been designed to exploit the access characteristics of traditional file systems, such as EXT4. Recently, flash friendly file system (F2FS) has been widely used and optimized for SSDs, which exhibits completely different access pattern and has not been taken into consideration during the design of mapping cache. In this paper, we are the first in proposing an efficient F2FS aware design for optimizing the efficiency of mapping cache. The basic idea of the work is to exploit the specific access characteristics of F2FS and propose to improve the hit ratio of the mapping cache. Experimental results show that this approach is able to significantly improve the performance of SSDs.**

I. Introduction

Recently, solid state drivers (SSDs) have been widely deployed in personal computers, mobile devices, and cloud systems. Compared with traditional hard disk drivers (HDDs), SSDs have several well-identified advantages, such as shock resistance, high random access performance, and lower power consumptions. Unlike traditional HDDs, SSDs are organized with a number of flash chips with each chip constructed with multiple blocks. Each block contains a fixed number of flash pages, which is the atomic unit of read and write operations. In order to find the physical access location of I/O requests, a flash translation layer (FTL) is designed for the address mapping between logical addresses (LA) and physical addresses (PA). For efficiently translating from LA to PA, a built-in RAM is equipped within SSD controller for caching these LA-to-PA address mappings, which is called as mapping cache. Due to the cost and space limitations of RAM, only limited number of address mappings are cached [1]. Hence, additional I/O operations are required for maintaining the data consistency between the built-in RAM and flash memory [2]. In this work, an approach is proposed to improve the efficiency of the mapping cache by exploiting the access characteristics of file systems.

Most of current mapping cache management schemes are designed with exploiting the access characteristics of traditional file systems, such as EXT3 and EXT4. For example, Chang et al. [3] proposed to exploit the file system access features to optimize the efficiency of flash translation layer (FTL). In this work, the allocation of metadata and data are separated and managed separately. Different from the traditional file systems, flash-friendly file system (F2FS) [4], which is a kind of log-structured file system (LFS) [5], has been recently used and optimized for flash memory. Based on our studies, we find that F2FS has completely different access patterns. F2FS is a state-of-the-art LFS which is specifically designed and optimized for flash based storage systems. Basically, F2FS adopts in-place write policy in metadata area while applies append-only write policy in data area. When a file is created or updated, F2FS sequentially allocates free block from the last-logging point and invalidates the previous block (for update operations) in the data area. Therefore, one block in data area can barely be multiply accessed by write requests. In addition, most of read requests on F2FS are prone to be random for the fragmentation issue, which has been studied in previous work [6]. With considering these unique access patterns of F2FS, the design of mapping cache with SSDs should be reconsidered. Otherwise, the additional I/O operations from the mapping cache will significant impact the performance of SSDs. However, none of recent proposed mapping cache management schemes are aware of these unique access patterns, which directly results in inefficiency of mapping cache.

In this paper, an F2FS aware mapping cache design is proposed in improving the performance of SSDs. Based on the different read and write access patterns of F2FS, mapping cache is separated into read mapping cache and write mapping cache for exploiting their access characteristics respectively. For the read access mappings, we use traditional management policy, such as Least Recently Used (LRU), to achieve locality exploration. For the write access mappings, we propose to exploit the specific write access characteristic of F2FS. With this approach, the specific access characteristics of F2FS can be well exploited for performance improvement. One of the challenges for the separated mapping cache design is the cache partition scheme between these two parts. In order to solve this issue, we propose a multiple access hard threshold (MAHT) scheme for the write mapping cache, which is used to determine the minimal number of entries for the write mapping cache. With applying this approach, the efficiency of read mapping cache is improved with little impact to the write mapping cache. Hence, the performance of SSDs can be highly improved. To the best of our knowledge, this is the first work in proposing F2FS aware mapping cache design to improve the performance of SSDs. Extensive experimental results show that the proposed approach is able to reduce read and write latencies to 31.2% and 38.7%, on average. This work achieves the following contributions:

- Analyzed the access patterns of F2FS, which significantly

*Liang Shi is the corresponding author: shi.liang.hk@gmail.com.

impact the design of mapping cache within SSDs;
- Proposed a separated mapping cache management scheme for F2FS based storage systems: read mapping cache and write mapping cache. These two parts are managed with separated management schemes and their sizes are determined based on the access characteristics of F2FS;
- An efficient implementation of proposed approach is presented. Experimental results show that the proposed approach is able to improve the performance of SSDs.

The rest of the paper is organized as follows. Section II presents the background and related work. Section III presents the problem of traditional mapping cache under F2FS. In Section IV, the proposed mapping cache management is proposed to improve the performance of SSDs. Experiments are presented in Section V. Section VI summarizes this work.

II. Background and Related Work

A. Mapping Cache within SSDs

Mapping cache is a built-in RAM of SSDs, which is used to store the address mappings between LA and PA. It is able to provide faster access latency for a request. However, the mapping cache size is always limited for cost and space restrictions. In order to match the size limitation of mapping, there are three types of mapping schemes developed during last decades based on their access granularity, as follows [7].

- **Block-level mapping**: it maps logical block number to physical block number in the granularity of flash block, which always only requires a small size mapping cache. In block-level mapping, LA is mapped to fixed page offset in a physical block. However, such a rigid mapping approach directly results in poor performance, especially that it would introduce bad random access performance.
- **Page-level mapping**: it maps logical page numbers (LPNs) to physical page numbers (PPNs) in the granularity of flash page, which always requires a large size mapping cache. Due to the space limitation of mapping cache [8], recent work proposes demand-based page-level mapping (DFTL), which is designed to only cache valuable address mappings [2].
- **Hybrid mapping**: it realizes the trade-off between page-level mapping and block-level mapping, which is able to achieve better performance compared with block-level mapping [9]. However, due to the costly merge operation during garbage collection, hybrid mapping can hardly achieve desired performance compared with page-level mapping.

Since DFTL is able to achieve the better performance, it is adopted as the default scheme in this work. DFTL is designed to exploit the locality of the accesses from file systems. It only caches the valuable address mappings, and the rest address mappings are stored in translation pages. However, if the access characteristics have little locality or the access patterns are not well studied, the performance would be significantly impact [2]. In this work, we find that the access characteristics of F2FS are very special, which cannot be well exploited by DFTL. In this case, we need to redesign the mapping cache management scheme to achieve improved performance.

B. File Systems for Flash based Storage Systems

Traditionally, file systems are designed for HDDs. With the widely replacement of HDDs with SSDs, file systems have been redesigned and optimized for SSDs. Different from traditional file systems, the file systems on SSDs are always designed based on the logging file systems. This is because flash memory has the characteristics of not-in-place updates. In this case, the logging feature can be well matched with the not-in-place updates.

F2FS is one of the most well developed file systems with keeping features of SSDs in mind. F2FS partitions the logical address space into data area and metadata area. In the data area, data address space is further partitioned into segments, where data are written in the append-only scheme. With applying append-only, the data can be written in parallel to achieve improve write performance at SSDs [10] [11]. In the metadata area, the metadata including superblock, segment information table, node address table and segment summary area. All of these medata data are updated in-place. In this work, we will exploit the specific architecture and access characteristics of F2FS to optimize the design of mapping cache with SSDs.

C. Mapping Cache Management

Page-level mapping is able to achieve the desired performance while a large buit-in RAM is required. Due to the space limitation of RAM, DFTL has been proposed to cache only valuable address mappings. In order to recognize valuable address mappings, many previous works are proposed to further improve the performance of DFTL.

For example, CDFTL implement two-level mapping caches to exploit the temporal locality and spatial locality, respectively [12]. In this way, more valuable address mappings meeting principle of locality can be stored within mapping cache. S-FTL exploits the spatial locality of address mappings through adopting translation page as the caching unit in mapping cache [13]. For keeping more valuable address mappings, translation page containing least number of dirty address mappings is evicted. To further exploit the spatial locality, TPFTL aims to keep more sequential accesses into mapping cache [1] as the unit of request. Since a large request contains multiple flash page access, TPFTL proposes to load all address mappings related to the entire request for reducing cache misses.

However, the above mapping cache management schemes do not take the special access patterns of F2FS into consideration. In this work, we design an efficient F2FS aware mapping cache management scheme to exploit the specific access characteristics, which is studied as follows.

III. Problem Statement

Recently, F2FS has been widely adopted in optimizing the performance of SSDs [4]. Compared with traditional file system, F2FS adopts append-only write policy in data area, which presents completely different access patterns. When multiple processes is creating or updating files simultaneously, F2FS always writes new data to a new free block sequentially. This directly results in file defragmentation of F2FS. Hence, for read requests, their access pattern from data area is becoming more random [6]. For write requests, append-only write policy enables files appended or updated in ascending order of LAs so that sequential writes are generated. In this case, with considering the read-access-only or write-access-only feature of data [14], previous blocks in data area can hardly be accessed again.

In order to understand the access patterns of F2FS, we collect the multi-time read accesses and multi-time write accesses under F2FS and EXT4, respectively. The multi-time accesses are for the data which are accessed more than two times. Since multi-time accesses are the main contribution in improving the hit ratio of mapping cache, the numbers of multi-time read and write accesses are collect to show the different access patterns between F2FS and EXT4. The results are presented in Figure 1. The detailed experiment settings and workload information can be found in Section V.

In Figure 1, comparing the number of multi-time read and write accesses between EXT4 and F2FS, we can find that F2FS has only 39.2% and 0.8% read and write requests accessed multi-time, on average. Based on the results, we have the following observations:

- F2FS has smaller number of multi-time read accesses. This is because that append-only write policy will update the previous read accessed data to a new free block so that subsequent read requests can not re-access the previous locations. The reason why Media application shows more number of multi-time read accesses is that media files can hardly be updated.
- For all three applications, compared with the number of multi-time write accesses of EXT4, the number of multi-time write accesses of F2FS is quite small. This is because that update operations in data area of F2FS can not modify the data in-place. Hence, these update operations induced multi-time write accesses in EXT4 disappear in F2FS.

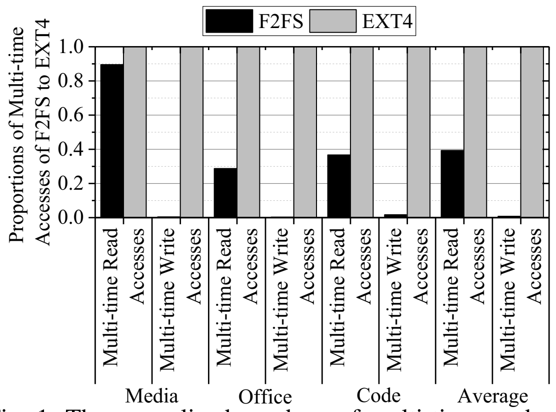


Fig. 1: The normalized numbers of multi-time read accesses and multi-time write accesses under F2FS and EXT4.

Based on the above observations, we find that the access patterns of F2FS are completely different with the access patterns of EXT4. However, many previous proposed mapping cache management schemes are proposed to exploit the access patterns of traditional file systems, such as EXT4. Considering the development of F2FS, current mapping cache management schemes should be re-designed for adopting the access patterns of F2FS.

IV. F2FS Aware Mapping Cache Design

A. Overview

In this section, we present an overview of the proposed approach. The architecture of the F2FS aware mapping cache is presented in Figure 2. In the upper level, we use F2FS as the default file systems. In the storage controller, the mapping cache is partitioned into two parts: read mapping cache and write mapping cache. These two mapping caches are in charge of managing read address mappings and write address mappings, respectively. Within each mapping cache, address mappings are basically organized with LRU policy independently. In order to avoid hardware modification, mapping cache is partitioned logically. To distinguish the type of resided mapping entries, a tag is used for each entry, 0 for write address mapping and 1 for read address mapping.

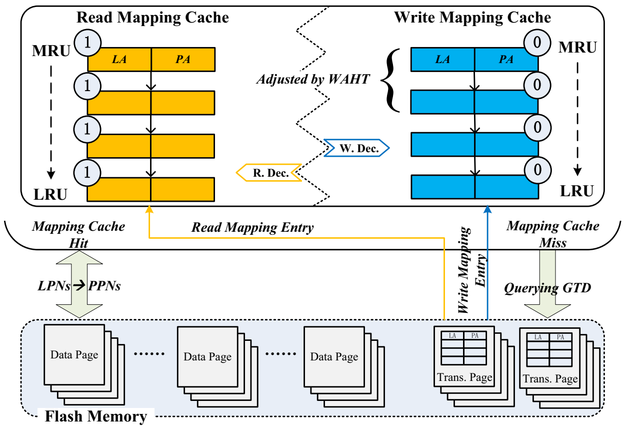


Fig. 2: The architecture of F2FS aware mapping cache.

In the following, we first discuss the request process of the mapping cache on the proposed architecture. Then, based on the architecture, we discuss mapping cache eviction approach. Finally, implementation and overhead are presented.

B. F2FS Aware Mapping Cache Design

The basic idea of the F2FS aware mapping cache design is to divide the mapping cache into two parts: one is for read accesses and the other is for write accesses. With this simple approach, the different access patterns of F2FS can be separated and well managed. In the following, we discuss how I/O requests are processed with the partitioned mapping cache.

- **I/O Request Hitting:**
  In default, we use LRU to organize all the mapping entries in each cache. For this case, there are three sub-cases: i) if a read request hits in the read or write mapping cache, its mapping entry is moved to the most recently used (MRU) position of corresponding cache; ii) if a write request hits in the write mapping cache, its mapping entry is also moved to the MRU position; iii) however, if a write request hits in the read mapping cache, its mapping entry will be moved to the MRU position of the write mapping cache. This is because the mapping entry is dirty, which should be maintained in the write mapping cache for future write back.
- **I/O Request Missing:**

In this case, free mapping entries should be allocated. If there are free entries, they are allocated right now. Otherwise, existing entries in the mapping cache should be evicted. However, there is a challenge for the eviction: which part of mapping entries should be evicted? The basic idea of the mapping entry eviction is to exploit the specific access characteristics of F2FS presented in Section III. First, we prioritize the eviction of write mapping entries which will not be accessed anymore with large probability. Second, in order to preserve these mapping entries which may be hit by metadata accesses, a multiple access hard threshold (MAHT) is set as the threshold of minimal entries of write mapping cache. Finally, if the number of entries in the write mapping cache is smaller than MAHT, entries in read mapping cache are selected for eviction.

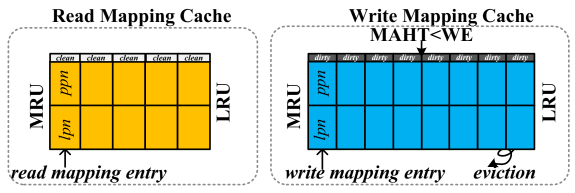
In this case, MAHT is designed to represent the number of multiple time access entries. Based on this idea, we propose to determine the MAHT as follows: Initially, MAHT is set to 0. If a write request hits one mapping entry for the first time, the MAHT is increased. If the hit entry is evicted, MAHT is decreased.

C. Entry Evictions from Mapping Cache

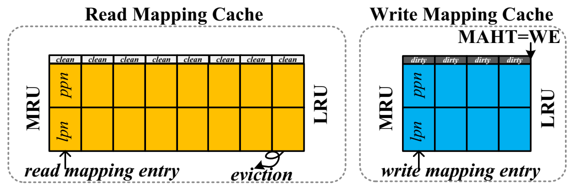
In this part, we give a detail discussion on the eviction of mapping entries, where three cases for entry eviction are presented.

Figure 3 shows the example for the three cases. In this figure, we assume there are 12 entries in the mapping cache. Initially, all the entries are free. With read and write requests issued, the corresponding entries are tagged as read or write mapping entries. MAHT is set following the process described above. We assume the number of entries in the read mapping cache is RE, and write mapping cache is WE. Based on the relationship between WE and MAHT, if a request is missing, there are three cases presented as follows.

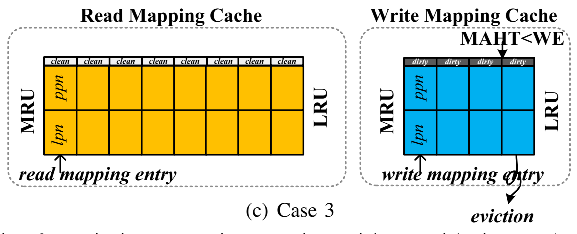
- **Case 1:** If MAHT is smaller than WE, an entry from the write mapping cache is evicted based on the LRU policy. Please note the if the missing request is read, RE is increased and WE is decreased. Otherwise, RE and WE is not changed. The reason for the design is based on the access characteristics presented in Section III. Most of the write entries in the write mapping cache are barely accessed. By prioritizing the eviction, we did not impact the efficiency of mapping cache.
- **Case 2:** If MAHT is equal to WE, all entries in the write mapping cache have been multiple accessed. Hence, an entry from the read mapping cache is suggested to be evicted following the LRU policy. In this case, for the values of RE and WE, if the missing request is read, RE and WE is not changed. Otherwise, RE is decreased and WE is increased. However, since read mapping entries have larger probability to be hit again, as shown in Figure 1, evicting a read mapping entry may introduce read hit ratio decrease. Therefore, for the value of MAHT, if the missing request is read, after evicting current read mapping entry, MAHT is decreased so that write mapping entries are able to be evicted in next missing. Otherwise, MAHT is maintained. From the above descriptions, the basic idea is to maintain the minimal number of entries in write mapping cache. However, if the efficiency of the read mapping cache is affected, the MAHT is decreased to avoid it.
- **Case 3:** The last case happens after Case 2. If the MAHT is decreased due to the read missing, read entry is evicted and MAHT is decreased. After that, if there are new request missing, the entries from the write mapping cache will be evicted again following Case 1.



(a) Case 1



(b) Case 2



(c) Case 3

Fig. 3: Evicting mapping entries with considering MAHT.

From the above descriptions, the entries in the read mapping cache are all clean and entries in the write mapping cache are all dirty. In this case, if entries are evicted from read mapping cache, there are no evict cost. Otherwise, the entries should be written back to the SSDs. Previous work [15] has done several optimization for the mapping cache eviction, such as evicting entries in the unit of translation page. In this work, we apply the similar approach. If the entries are evicted from read mapping cache, it is evicted in the unit of mapping entry. If the entries are evicted from write mapping cache, it is evicted in the unit of translation page. It means that all the entries belonging to the translation page of the current evicted mapping entries are selected and written back. Please note that the written back entries are not erased. If following request hits, it can be reused, and if they are selected for eviction, they are overwritten without additional written back operations.

D. Implementation and overhead analysis

In this section, we give an example for request hit or miss in the mapping cache in Figure 4. Assume that there are 6 requests, and each request requires one address mapping. (1)-(2) The read request hits in read mapping cache. Then, the physical location of accessing flash page is acquired and data are read from the physical page. (3)-(5) The write request hits in read mapping cache; Read mapping entry is added to

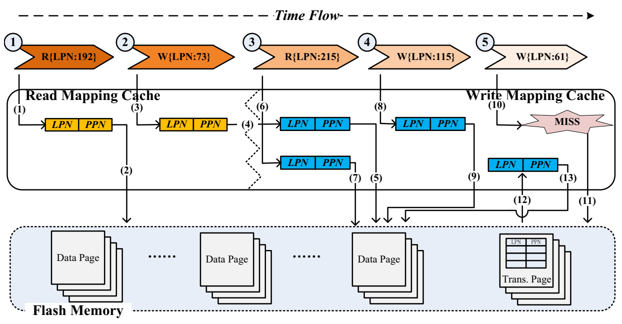


Fig. 4: An example for request hit/miss in mapping cache.

write mapping cache; Get the physical location of accessing flash page and write new data to this physical page. (6)-(7) Read request hits in write mapping cache; Then, get the physical location of accessing flash page and read data from this physical page. (8)-(9) Write request hits in write mapping cache; Then, get the physical location of accessing flash page and write new data to this physical page. (10)-(13) Write (Read) request misses in mapping cache; Query global table directory (GTD) and get the physical location of required translation page; Read translation page and add required write (read) entry into write (read) mapping cache; Determine the physical location of access flash page; Then, write (read) new data to (from) this physical page.

TABLE I: The characteristics of all traces.

Workloads	Trace Size	Footprints	WR[1]	RR[1]	Cache Size
Media	4.7GB	4.5GB	94.5%	5.5%	512KB
Office	867.2MB	847.6MB	97.1%	2.9%	128KB
Code	2.4GB	1.5GB	63.7%	36.3%	256KB
9W1R	1.3GB	1.1GB	87.9%	12.1%	128KB
7W3R	1.3GB	920MB	65.2%	34.3%	128KB
5W5R	1.4GB	680MB	45.9%	54.1%	128KB
3W7R	1.5GB	440MB	28.2%	71.8%	128KB
1W9R	1.6GB	200MB	12.3%	87.7%	128KB

[1] WR: Write Ratio; RR: Read Ratio.

Overhead Analysis: The implementation of the proposed approach needs to maintain a tag per address mapping. Each tag only needs one bit. Assume that if the size of mapping cache is 512KB and each address mapping occupy 8B. In this case, only 8KB is needed to maintain this tag. In addition, three counters are needed to maintain the size of read mapping cache, the size of write mapping cache and MAHT. The maximum space cost for each counter is 16 bits for a 512KB mapping cache. Hence, the space overhead is negligible for mapping cache. For the computation overhead of the proposed approach, there are only `MOD`, `OR` and `AND` operations, which also introduce negligible cost.

V. Experiment and Analysis

A. Experiment Setup

1) Trace-driven Simulator: We use a trace-driven simulator, Flashsim [16] to evaluate the proposed approach. In order to simulate a state-of-the-art SSD, related parameters of Flashsim are updated.

In the study, an 100GB SSD is modeled, where each block contains 64 flash pages with a page size of 4KB. Page read latency is set as $25us$, page write latency is set as $200us$, and block erase latency is set as $1.5ms$ [10]. The default FTL is DFTL [2]. In order to minimize the cost of mapping cache, DFTL is designed to exploit the locality of workloads. The size of mapping cache is highly correlated with the capacity of SSDs. In this work, we vary the size of mapping cache from 128KB to 512KB based on the footprint of each workload [15]. Figure I presents the size of the mapping cache for each workload and their corresponding footprints. Although the access spaces of applications are irregular, we select the closest capacities of mapping cache among several fixed values, which are presented in Figure I. Other components in SSD are implemented in default.

2) Workloads: In order to collect traces under F2FS, we mounted F2FS [4] on an 120GB intel SSD. Then, we ran three common applications (media, office and code) for generating user access behaviours.

In addition, in order to quantitatively analyze the impact between read access pattern and write access pattern, we also synthesized several traces in this work. These synthetical traces are collected through generating multiple processes for writing and reading a large file. Each process writes or reads one file repeatedly. Table II summaries the access behaviours of all workloads.

TABLE II: The access behaviours of all workloads.

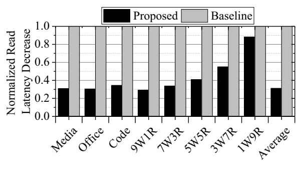
Workloads	Access Behaviours
Media	Listen to music cyclically; View photos repeatedly; Watch movie; Copy music, photos and movie.
Office	Edit words, slides and excels; Copy office files; Open office files repeatedly.
Code	Read code repeatedly; Coding; *Make* operations.
9W1R	9 processes are writing and 1 process is reading.
7W3R	7 processes are writing and 3 processes are reading.
5W5R	5 processes are writing and 5 processes are reading.
3W7R	3 processes are writing and 7 processes are reading.
1W9R	1 process is writing and 9 processes are reading.

B. Experiment Results

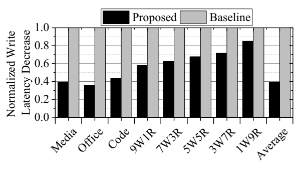
The proposed approach aims at improving the performance of SSDs. In this section, the *read latency and write latency improvement* are evaluated. Then, the *cache hit ratio* is evaluated to show the reason why performance is highly improved. Finally, we collect the changes of MAHT to show the effectiveness of the proposed approach.

1) Performance Improvement: In this section, we present the achieved performance improvement based on the proposed F2FS aware mapping cache design. The baseline represents work with DFTL as the default mapping cache management scheme. Due to its unawareness of the access characteristics of F2FS, DFTL has much worse performance. In Figure 5, both read and write latencies are presented, which are normalized to the read and write latencies of DFTL. As shown in this figure, we can find that the read and write latency are reduced to 31.2% and 38.7%, respectively, on average.

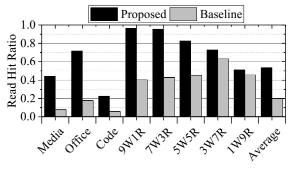
***Read Performance*:** In Figure 5(a), the read latency of the three real workloads are reduced to less than 35% compared with DFTL. That means the proposed approach is able to highly improve the performance of SSDs through separating the management of read and write mapping entries. In order to reveal the details for the improvement, we evaluate the read hit ratio on the mapping cache, shown in Figure 5(c). In this figure, we can find that the read hit ratios of these three workloads are significantly improved. That is, more valuable



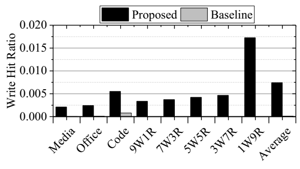
(a) Normalized Read Latency



(b) Normalized Write Latency



(c) The hit ratio of read requests



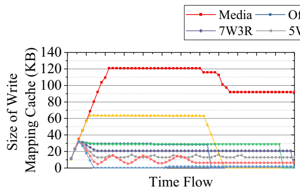
(d) The hit ratio of write requests

Fig. 5: Normalized performance improvement of SSDs and hit ratio improvement of mapping cache.

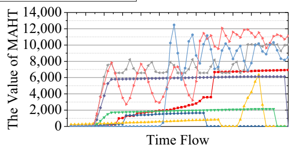
read address mappings are cached so that the read performance is able to be improved.

For the synthetical workloads, more processes are reading and less processes are writing, the weaker the performance improvement is achieved by the proposed approach. This is because when more read operations are issued to SSDs by multiple read processes, more read entries are going to be evicted due to the limited space of mapping cache. Hence, the read hit ratio is going to be reduced when more processes are reading. As shown in Figure 5(c), the improved read hit ratio is becoming smaller with the increasing of read processes.

***Write Performance*:** In Figure 5(b), the write latency of three real workloads are highly reduced as well. However, the achieved write performance improvement is smaller than read performance improvement. This is because that F2FS writes data in append-only write policy, which barely hit previous written data. Hence, the potential of write performance improvement is smaller than that of read performance improvement. As shown in Figure 5(d), the write hit ratio of three applications are quite small, which are less than 1%.



(a) Size of write mapping cache for each workload under the time flow.



(b) MAHTs of all wrokloads.

Fig. 6: Size of write mapping cache and MAHTs.

For the synthetical workloads, when more processes are reading, the achieved write performance improvement is going to be smaller. The reason is that these read processes are able to occupy most space of mapping cache so that write entries have to be evicted from write mapping cache. In order to understand the size variation between read and write mapping cache at run time, we collected the size of the write mapping cache online for each workload in Figure 6(a). As shown in the results, the size of write mapping cache is varied significantly. Initially, with the incoming write requests, its size is increased. Then, it is reduced for the increased read requests. From the results, we can find that 1W9R has a little size of write mapping cache. In this case, the benefit for read or write performance improvement will be limited.

2) MAHT Analysis: The proposed approach is able to achieve highly performance improvement based on the adjustment of MAHT, which guides the space adjustment of write mapping cache for adapting the access patterns of F2FS. In order to show the process of MAHT adjustment, Figure 6(b) is presented. In this figure, we can find that MAHT is dynamically adjusted and have a matching pattern with space adjustment of write mapping cache. That is, the proposed hard threshold is efficient to adapt the access patterns of F2FS. Hence, the achieved performance improvement of proposed approach is significant.

VI. Conclusion

In this paper, we propose an efficient F2FS aware mapping cache design, which partitions mapping cache into two separate mapping caches to exploit the access patterns of F2FS. Experimental results show that this approach decreases read latency and write latency to 31.2% and 38.7%, on average.

VII. Acknowledgement

The authors thank the anonymous reviewers for their comments and suggestions. This work is supported by the NSFC 61772092 and 61572411.

References

[1] Y. Zhou, F. Wu, P. Huang, X. He, C. Xie, and J. Zhou, "An efficient page-level ftl to optimize address translation in flash memory," in *EuroSys*. ACM, 2015.
[2] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," in *ASPLOS XIV*. ACM, 2009.
[3] Y. H. Chang, P. L. Wu, T. W. Kuo, and S. H. Hung, "An adaptive file-system-oriented ftl mechanism for flash-memory storage systems," in *TECS*. ACM, 2012.
[4] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2fs: A new file system for flash storage." in *FAST*. USENIX, 2015.
[5] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *TOCS*. ACM, 1992.
[6] J. Park, D. H. Kang, and Y. I. Eom, "File defragmentation scheme for a log-structured file system," in *SIGOPS*. ACM, 2016.
[7] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "System software for flash memory: a survey," in *EUC*. Springer, 2006.
[8] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: opportunities and challenges," in *SIGMOD*. ACM, 2013.
[9] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *TECS*, 2007.
[10] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H.-M. Sha, "Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives," in *MSST*. IEEE, 2014.
[11] C. Gao, L. Shi, C. Ji, Y. Di, K. Wu, J. Xue, and E. Sha, "Exploiting parallelism for access conflict minimization in flash-based solid state drives," in *TCAD*. IEEE, 2017.
[12] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "A two-level caching mechanism for demand-based page-level address mapping in nand flash memory storage systems," in *RTAS*. IEEE, 2011.
[13] S. Jiang, L. Zhang, X. Yuan, H. Hu, and Y. Chen, "S-ftl: An efficient address translation for flash memory by exploiting spatial locality," in *MSST*. IEEE, 2011.
[14] Q. Li, L. Shi, C. J. Xue, K. Wu, C. Ji, Q. Zhuge, and E. H.-M. Sha, "Access characteristic guided read and write cost regulation for performance improvement on flash memory." in *FAST*. USENIX, 2016.
[15] C. Ji, C. Wu, L.-P. Chang, L. Shi, and C. J. Xue, "I/o scheduling with mapping cache awareness for flash based storage systems," in *EmSoft*. ACM, 2016.
[16] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *SIMUL*. IEEE, 2009.