PCDedup: I/O-Activity Centric Deduplication for Improving the SSD Lifetime

Myoungjun Chun
Seoul National University
mjchun@davinci.snu.ac.kr

Taejin Kim
Samsung Electronics
taejin.kim@samsung.com

Sungjin Lee
DGIST
sungjin.lee@dgist.ac.kr

Jihong Kim
Seoul National University
jihong@davinci.snu.ac.kr

ABSTRACT

When data deduplication is used for extending the SSD lifetime inside an SSD, one of the key performance factors is how to manage the fingerprint cache. Since the size of the fingerprint cache is limited, the fingerprint cache should be very selective in choosing which fingerprints should be stored in the cache. In this paper, we show that write program contexts, which are automatically extracted during run time, can accurately capture their future data duplicability. Based on this observation, we propose `PCDedup` for SSD-internal data deduplication. `PCDedup` automatically filters out undesirable fingerprints from the cache, thus improving the cache hit ratio. Our experimental results show that `PCDedup` can improve the SSD lifetime by up to 16.4% over the existing deduplication scheme while the fingerprint management overhead is lowered on average by 68.6%.

1 INTRODUCTION

As the price-per-byte of NAND flash memory is rapidly decreasing by continuous innovations in the semiconductor technology (e.g., 3D NAND flash [1]), NAND flash-based solid-state drives (SSDs) have been positioned as primary storage solutions for smartphones to large-scale data centers. In spite of wide-reaching successes in various storage market segments, managing the lifetime of SSDs remains to be one of the key design challenges in SSDs [2]. A decreasing trend in the number of P/E cycles of NAND flash memory seriously limits the overall lifetime of flash-based SSDs, making it difficult for SSDs to be used in write-intensive applications.

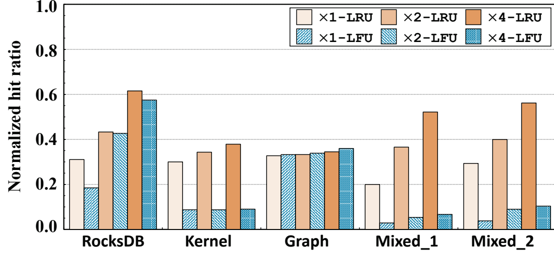
In order to extend the lifetime of flash-based SSDs, data reduction techniques are commonly employed because they reduce the total amount of data written to an SSD, which directly affects the SSD lifetime. Data deduplication technqiues are such examples by preventing duplicate data from being written again [3, 4]. In an *in-line* deduplication schemeused for SSDs, a cryptographic hash function $h(x)$ such as SHA-1 or MD5 [3, 5] is used to test whether a duplicate data exist. When a page p is written to an SSD, an SSD controller first computes the hash value $h(p)$ of the page p, which is called as the fingerprint of p. If the same fingerprint $h(p)$ is found in the SSD, it indicates that the page p is a duplicate of the page that was previously written. Since the same page p was already stored in the flash memory, the SSD controller simply updates its L2P mapping table without writing the same page again. In order to quickly decide if the fingerprint $h(p)$ matches with the existing page in the SSD, a fast memory (e.g., SRAM) cache is used for storing fingerprints of pages in the SSD. (In this paper, we call this cache *a fingerprint cache* or a *cache* when no confusion arises.)

If the size of a fingerprint cache is sufficiently large so that all the distinct fingerprints of written data can be stored, the efficiency of a deduplication scheme solely depends on how much future data writes are duplicates. However, when the size of a fingerpint cache is limited, the efficiency of a deduplication scheme is strongly affected by how the fingerprint cache is managed. For example, several groups have recently proposed different management techniques for the fingerprint cache by exploiting the workload's spatial and temporal locality [6–10].

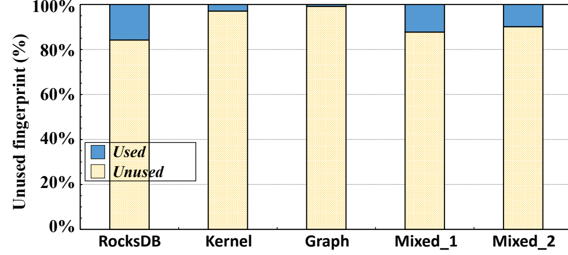
In this paper, we argue that *I/O activity-centric* fingerprint cache management is needed for maximizing the efficiency of a fingerprint cache so that more duplicates can be identified by a deduplication technique. Unlike conventional content-centric approach where no I/O context information is exploited, we show that the future data duplicability can be reliably predicted at the I/O-activity level (e.g., logging and compaction activities in RocksDB [11]). That is, by monitoring data duplication characteristics at each I/O activity, we can predict whether data written from the same I/O activity will produce duplicate copies or not in the future.

Based on this observation, we propose a novel I/O-activity centered deduplication technqiue, called `PCDedup`, that efficiently improves the SSD lifetime. By exploiting the future

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APSys '19, August 19–20, 2019, Hangzhou, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6893-3/19/08…$15.00
https://doi.org/10.1145/3343737.3343747



(a) Hit ratio variations under different cache sizes and replacement policies.



(b) Usage breakdown of fingerprints before their evictions.

Figure 1: Comparisons of existing fingerprint cache management techniques.

data duplicability at the I/O-activity level, PCDedup efficiently manages a fingerprint cache. Unlike the existing scheme, PCDedup employs an admission test for the fingerprint cache and decides which fingerprints will be stored in the fingerprint cache based on the future data duplicability of the I/O activity, thus preventing unnecessary fingerprints from polluting the fingerprint cache. PCDedup automatically detects I/O activities during runtime based on write program contexts (PCs) [12]. In order to accurately reflect the time-varying nature of in-line SSD deduplication, PCDedup periodically monitors each PC's future data duplicability and updates a list of admissible PCs to the fingerprint cache.

In order to evaluate the effectiveness of PCDedup, we have implemented PCDedup by extending an existing page-level FTL using a flash emulation environment [13]. Our exprimental results using real-world applications show that PCDedup can reduce the total amount of written data by up to 16.4% over the existing deduplication scheme while the fingerprint management overhead is lowered on average by 68.6%.

The rest of this paper is organized as follows. We explain the key motivations behind PCDedup in Section 2. Section 3 describes the design and implementation of PCDedup. The experimental results are shown in Section 4. Finally, we conclude with a summary and future work in Section 5.

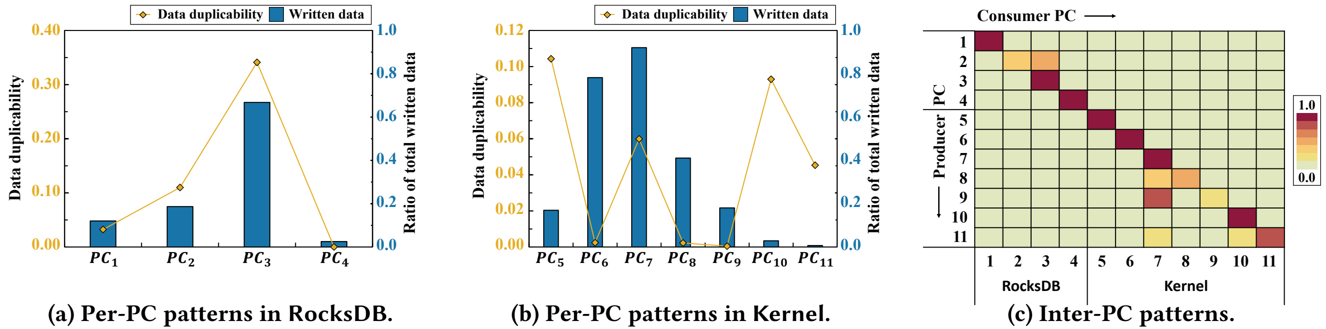
2 BASIC IDEA

2.1 Inefficient Fingerprint Cache Management

In order to understand how the fingerprint cache size affects the overall efficiency of an in-line deduplication technique, we quantitatively evaluated the performance of existing fingerprint cache management techniques while varying the fingerprint cache size. For each benchmark program Φ, we first computed the total number N_Φ of fingerprints that appear more than once in its execution. For our evaluation using the benchmark program Φ, we used three different fingerprint cache configurations: ×1, ×2 and ×4 configurations where the size of a fingerprint cache is set by N_Φ, 2× N_Φ, and 4× N_Φ, respectively. For each cache configuration, we tested two representative cache replacement policies, LRU and LFU, used in most in-line deduplication techniques [8]. See Section 4.1 for a detailed description of benchmarks.

Fig. 1 summarizes the key findings from our evaluation study. The values in Fig. 1 are normalized to the hit ratio with an unlimited fingerprint cache. As shown in Fig. 1(a), the hit ratio of existing fingerprint cache management techniques is very low. For example, when the cache is managed under LRU, which works better than LFU, only 31.1% of the total duplicates can be successfully identified even though the fingerprint cache is large enough to contain all the duplicate fingerprints that appear more than once of a benchmark program. Larger cache sizes, as shown in ×2-LRU and ×4-LRU in Fig. 1(a), do not significantly improve the hit ratio either. For example, in Graph, the hit ratio of ×4-LRU is increased merely by 3.21% over that of ×1-LRU.

In order to find the root cause of the inefficiency in the fingerprint cache, we monitored how each entry in the fingerprint cache is referenced from its insertion to the cache to its eviction from the cache. For simplicity, we divided fingerprints into two sets, *Used* and *Unused*. If a fingerprint is referenced at least once before its eviction from the cache, it is added to *Used*. On the other hand, a fingerprint is evicted with no reference, it is added to *Unused*. Fig. 1(b) shows a breakdown of two sets under the ×1 configuration. As shown in Fig. 1(b), at least 84.2% of all the fingerprints belong to *Unused*. That is, most fingerprints are added to the fingerprint cache but they polluted the fingerprint cache only by evicting the useful fingerprints which belong to *Used*. In order to improve the hit ratio of the fingerprint cache, it is important to decide in advance whether a fingerprint is in *Used* or *Unused*. Unfortunately, existing management techniques cannot effectively predict *a priori* that a fingerprint will be a member of *Used* or a member of *Unused*.



(a) Per-PC patterns in RocksDB. (b) Per-PC patterns in Kernel. (c) Inter-PC patterns.

Figure 2: Data duplication patterns under different workloads.

2.2 Program Contexts as Duplication Predictors

In order to improve the hit ratio of a fingerprint cache, our main insight was that the fingerprint cache should be managed *at a higher abstraction level than the logical block address (LBA)* so that cache pollutions from fingerprints in *Unused* can be avoided. Since I/O activities are known to be effective in representing the application's I/O context [12], we focused on developing I/O activity-centric management technique. An I/O activity is captured by a program context (PC), which represents an execution phase of an application [14]. In this work, since we are interested in writes, a program context is defined as an execution path of an application which invokoes write-related system functions such as `write()` and `writev()` in the Linux kernel. We represent a PC by summing program counter values of all the functions along the execution path which leads to a write system call [12].

The key motivation behind PCDedup is that if we monitor data duplication characteristics of each PC, we may be able to predict the future data duplicability of the same PC. In order to validate that PCs are useful abstraction units in managing the fingerprint cache, we evaluated if different PCs exhibit distinct characteristics in their data duplication patterns. Fig. 2 shows the evaluation results in RocksDB and Kernel where four PCs and seven PCs were identified, respectively. In Figs. 2(a) and 2(b), PCs are compared by their data duplicability and the amount of written data. For a PC, PC_i, of a program Φ, we define the data duplicability $dd(PC_i)$ of PC_i as the probability that data written by PC_i will be duplicates of later writes from the program Φ. For example, in Fig. 2(a), $dd(PC_3)$ is 0.34 while $dd(PC_1)$ is 0.03. In RocksDB, about 34.1% of data written by PC_3 (which represents the compaction activity) are duplicates of later writes. On the other hand, only 3.2% and 0.1% of data written by PC_1 (for write-ahead logging in RocksDB) and PC_4 (for human-readable logging in RocksDB) are duplicates of later writes, respectively. Furthermore, Fig. 2(a) shows that PC_3 is responsible for about 66.8% of the total amount of writes in RocksDB. With its high data duplicability ratio, it is logical for fingerprints from PC_3 to be inserted to the cache. On the other hand, it would be not effective to store fingerprints from PC_4 into the cache with its very low data duplicability.

We observed similar per-PC data duplication patterns in Kernel as shown in Fig. 2(b). PC_5, PC_7, and PC_{10}, which correspond to write activities that produce executable files during a kernel compilation process, are likely to be duplicated later by succeeding compilation steps. On the other hand, PC_6, PC_8, and PC_9, which create temporary files that are quickly deleted, are less likely to be deduplicated later.

In our evaluation, we also observed that not all PCs are equally duplicable each other. Instead, for a given PC PC_i, there exist one or two dominant PCs, PC_j or PC_k, whose writes are duplicates of the previous writes from PC_i. We call PC_i a producer PC and PC_j and PC_k consumer PCs. Fig. 2(c) illustrates that strong producer-consumer PC relationships exist in Mixed_1 using a heat map representation. For each producer PC shown along the y-axis, each row shows its consumer PCs with their proportions of duplicability withe the producer PC. For example, for the producer PC PC_2 (which represents the flushing activity in RocksDB), 46.3% of its writes are duplicates with PC_2 itself while PC_3 (which represents the compaction activity) are responsible for the remaining 53.7% of duplicates with PC_2. The strong producer-consumer PC relationship provides a useful direction for reducing the search overhead of the fingerprint cache. If the consumer PCs of a given PC were known, fingerprint search steps could be limited to its consumer PCs, thus dramatically reducing the overall fingerprint search overhead.

3 DESIGN AND IMPLEMENTATION OF PCDEDUP

As we have observed in Section 2.2, PC provides useful hints to predict future duplication patterns. Using PC information, PCDedup is able to make a better decision in three main stages of the deduplication process, *cache insertion*, *victim selection*, and *fingerprint search*.

Fig. 3 shows an overall write procedure of PCDedup. It first collects PC information from incoming I/O requests (a PC extractor) and computes fingerprints (a hash engine). A PC-based admission controller decides whether or not to cache

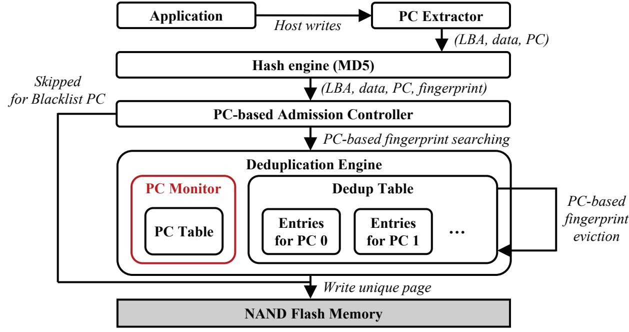


Figure 3: An overall write procedure of PCDedup.

time window 0

PC no.	Dedup_ Count	Req_ Count	Dedup_ Ratio	Relation_ Bitmap				Blacklist	Cumulative_ Dedup_Ratio
1	40	100	0.00	■	■	■	■	N	0.00
2	1	20	0.00	■	■	■	■	N	0.00
3	0	10	0.00	■	■	■	■	N	0.00
4	5	50	0.00	■	■	■	■	N	0.00

Total deduplication ratio = 0.26
Blacklist threshold parameter = 0.2
Blacklist threshold = 0.26 × 0.2 = 0.052

time window 1

PC no.	Dedup_ Count	Req_ Count	Dedup_ Ratio	Relation_ Bitmap				Blacklist	Cumulative_ Dedup_Ratio
1	0	0	0.40	■	□	□	■	N	0.40
2	0	0	0.05	□	■	□	□	Y	0.05
3	0	0	0.00	□	□	■	□	Y	0.00
4	0	0	0.10	■	□	□	■	N	0.10

Figure 4: An illustrative example of updating PC table.

given fingerprints for deduplication, depending on their popularity. A deduplication engine maintains promising fingerprints with their PC values, collecting the correlation between PCs as well as their duplicate patterns. Extracting PCs from I/O requests and computing fingerprints are explained very well by many prior studies [4, 6, 12, 14]. Thus, in the rest of this section, we focus on explaining how PCDedup's deduplication engine operates to collect popular fingerprints and estimate the correlation between PCs. We then explain how PCDedup uses the collected information to better manage a fingerprint cache.

3.1 PC Monitor

PCDedup internally maintains two tables, a PC table and a Dedup table. The PC table contains PC-related information, while the Dedup table has the information for dedpulication, such as fingerprints, source LBAs, and so on. Each entry of the PC table keeps several fields to derive four parameters: 1) *Dedup_Ratio*, 2) *Cumulative_Dedup_Ratio*, 3) *Relation_Bitmap*, and 4) *Blacklist*.

Dedup_Ratio is a deduplication ratio of a specific PC, which can be expressed as $\frac{Dedup_Count}{Req_Count}$, where *Req_Count* is the number of write requests to a PC and *Dedup_Count* is the number of requests deduplicated. When the SSD receives a write request, the PC monitor first locates a table entry corresponding to a given PC and updates *Req_Count* in the table. Then, it looks up the Dedup table to figure out the desired PC exists. If does, *Dedup_Count* in the PC increases by one. The PC monitor resets *Dedup_Count* and *Req_Count* of each PC after a predefined time window. To capture a long-term characteristic of PCs, PCDedup updates a moving-average value, *Cumulative_Dedup_Ratio*, of each entry using *Dedup_Count* and previous *Cumulative_Dedup_Ratio*.

Relation_Bitmap maintains the correlation between two PCs that have a producer-consumer relationship. For example, if the same data that were previously written by PC_1 are rewritten by PC_2, it means the two PCs, PC_1 and PC_2, are strongly correlated with each other. *Blacklist* represents PCs that have had little deduplication ratios in the past and are less likely to remove future duplicate writes. When *Dedup_Ratio* is being calculated, PC monitor sees if each PC should belong to the blacklist. PCDedup compares *Dedup_Ratio* of each PC with the average deduplication ratio of all the PCs. If it is lower than 20% of the average, the corresponding PC is put into the black list. The 20% threshold value is empirically decided based on our experiments. Once a certain PC becomes a member of the *Blacklist*, all the requests belonging to that PC are filtered out and are not considered to be candidates for deduplication. Since fingerprints for black-listed PCs are not cached in the fingerprint cache, PCDedup can better use the cache only for popular data.

Fig. 4 shows an illustrative example when the PC monitor updates the PC table. Since the time window has not yet passed, the *Dedup_Ratio* of each PC has not yet been calculated. After a time window, the PC monitor updates the *Dedup_Ratio* of each PC. The PC monitor calculates the blacklist threshold value of 0.052 by multiplying the total cache's *Dedup_Ratio* of 0.26 by the Blacklist threshold parameter of 0.2. In the example, PC_2 and PC_3 showed poor deduplication efficiency (0.05 and 0.00, respectively). Therefore, PC_2 and PC_3, which have lower efficiency than the blacklist threshold, are updated to blacklist. *Dedup_Count* and *Req_Count* are then initialized for calculation of the next time window. For the example in Fig. 4, we assume that writes from PC_2 and PC_3 are duplicated only with the pages previously written by oneself, while the duplication caused by PC_1 and PC_4 occurs also on the pages written by each other. Therefore, the *Relation_Bitmap* values of PC_2 and PC_3 are updated only in the second index and the third index, respectively, while those of PC_1 and PC_4 are updated with the first and fourth indices.

3.2 PC-based Selective Deduplication

The admission controller is responsible for performing selective deduplication, and, as pointed out before, it is simply done by referring to a blacklist field in the PC table. More specifically, when a new write request comes, PCDedup checks

if its PC is blacklisted or not and then inserts the new fingerprint to the Dedup table only if it is not.

The selective deduplication can be controlled by dynamically adjusting two parameters, the blacklist threshold and the time window length. When the fingerprint cache size is small, a high blacklist threshold is preferred so as to aggressively prevent unnecessary fingerprints from being cached. Conversely, when the cache is large enough for the workload size, it is necessary to decrease the threshold to perform as much deduplication as possible. It is also important to properly tune the length of the time window for high deduplication efficiency. If the length of the time window is too long, fingerprints from PCs that should be blacklisted are unnecessarily inserted into the cache until time window is over. On the other hand, if we use too short time window to prevent this, the PC table update process becomes frequent and its overhead increases. Therefore, it is necessary to match the unit of time window with the minimum unit that can catch the duplication characteristic of the workload.

3.3 PC-based Eviction Policy

In most cases, the size of the fingerprint cache cannot contain all of the fingerprints in the workload, so some fingerprints have to be evicted from the cache to insert new fingerprint. `PCDedup` uses an enhanced LRU policy that is combined with PC-based prediction, which performs better than a naive LRU policy considering only the recency of fingerprint references.

The eviction process of `PCDedup` is done in the following order. First, `PCDedup` checks whether there is a cached fingerprint from a blacklisted PC. Since blacklisted PCs are expected to scarcely cause deduplication, `PCDedup` immediately remove all such fingerprints to reclaim the cache space for new fingerprints. This is a uncommon case, but it can happen in the eviction immediately after some PCs are blacklisted. If such fingerprints do not exist, `PCDedup` retrieves the PC table to figure out which PC has the smallest *Dedup_Count* in the current time window among non-blacklisted PCs. Then, it selects the least recently used fingerprint from the PC as a victim, and evicts it from the cache.

The proposed PC-based LRU policy can solve the duplication priority inversion problem that can occur when using the naive LRU. For example, as shown in Fig. 2(a), in RocksDB, data duplication occurs mainly in the flushing context and the compaction context. Due to the nature of the LSM tree, the flushing context at the higher level is more frequent than compaction at the lower level. However, the data duplicability of the compaction context is higher than the flushing context because the data used in the compaction is less likely to change. When the naive LRU is used as an eviction policy, fingerprints created by the compaction context are frequently evicted by flushing, which means that a fingerprint with high data duplicability becomes a victim in order to insert a fingerprint with low data duplicability. The proposed PC-based LRU policy solves this problem by preventing the fingerprint of a PC showing high data duplicability from becoming a victim even if it is least recently entry.

3.4 Selective Fingerprint Searching

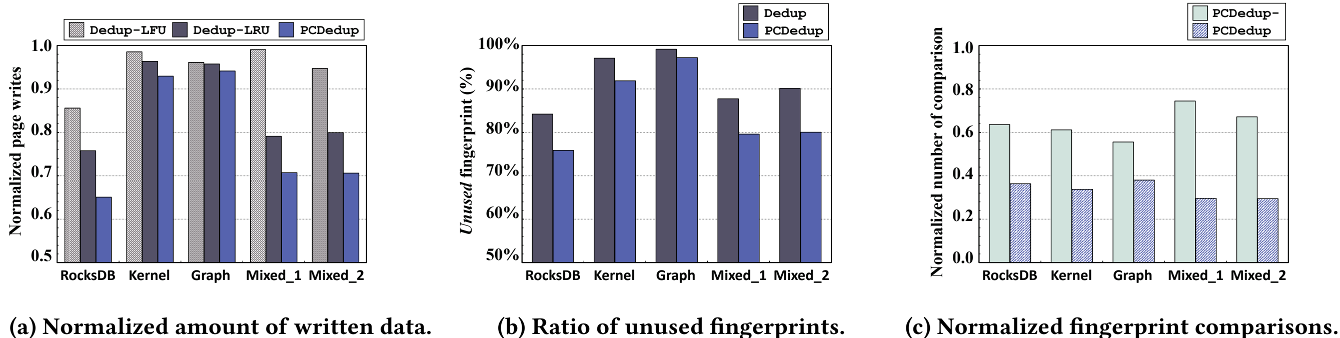
`PCDedup` reduces fingerprint search overheads by separately managing the fingerprint cache on a PC-by-PC basis. That is, `PCDedup` maintains multiple buckets in the Dedup table and puts entries with an identical PC in the same bucket. This makes it possible for us to look up only a single bucket with a smaller number of fingerprints, thereby reducing overall search latency. As illustrated in Fig. 2(c), however, there is a possibility that a fingerprint we look for are kept in another bucket with a different PC number. This problem can be addressed by seeing *Relation_Bitmap*. If `PCDedup` does not find a matched fingerprint in the designated bucket, it refers to *Relation_Bitmap* and figures out what a correlated PC is. Then, it looks up the bucket with the correlated PC. The PC-based selective fingerprint search effectively reduces the number of comparisons over the existing deduplication schemes that need to search for a matched one in a large fingerprint cache.

4 EXPERIMENTAL RESULTS

4.1 Experimental Settings

In order to evaluate the effectiveness of the proposed techniques, we have implemented `PCDedup` by extending page-level FTL using a flash emulation environment [13]. Our evaluation platform can support up to the 512-GB capacity, but for fast evaluation, the storage capacity was set to 16 GB. We have compared `PCDedup` with two different existing deduplication schemes: `Dedup-LFU` and `Dedup-LRU`. `Dedup-LFU` performs existing page-based deduplication and uses LFU as fingerprint cache eviction policy. `Dedup-LRU` also performs page-based deduplication the same, but uses LRU as fingerprint cache eviction policy.

For our evaluations, we used five workloads, RocksDB (a key value database application [11]), Graph (Graphchi, a graph analysis application [15]), Kernel (a Linux kernel development workload), Mixed_1 (RocksDB + Kernel), and Mixed_2 (RocksDB + Graph). For RocksDB, *fillrandom* benchmark in *db_bench tool* was used to generate write-intensive workload. For Graph, *PageRank* algorithm was repeated 5 iterations with stack overflow network graph [16] as an input data. For Kernel, a Linux kernel was built 3 times by GCC. Two mixed workloads, Mixed_1 and Mixed_2, were used to mimic more realistic environments where multiple applications are sharing a single SSD. For the fingerprint



(a) Normalized amount of written data. **(b) Ratio of unused fingerprints.** **(c) Normalized fingerprint comparisons.**

Figure 5: Comparisons of different deduplication schemes under five I/O workloads.

cache, we used ×1 configuration (in Section 2.1) where the cache size is set by the total number of duplicate fingerprints in the each workload.

4.2 Performance Evaluation

In order to compare the lifetime gains of PCDedup over the existing deduplication schemes, we measured the number of page writes for each deduplication scheme as shown in Fig. 5(a). Fig. 5(a) is normalized to the number of page writes when no deduplication scheme was applied. PCDedup effectively reduced page writes by up to 40% (22.8% on average) and 16.4% (9.4% on average), over Dedup−LFU and Dedup−LRU, respectively. In particular, the proposed PCDedup was the most effective in RocksDB. It is because, in RocksDB, one PC (compaction) has very high data duplicability while that of others (for logging) has very trivial. In such a case, PCDedup can keep only promising fingerprints from the PC with high duplicability, while the other deduplication schemes cannot prevent unnecessary fingerprints from evicting promising ones.

To evaluate the efficacy of the cache management, we also measured the ratio of unused entries before eviction when Dedup−LRU and PCDedup. As shown in Fig. 5(b), PCDedup reduced unused fingerprints in the cache by 6.8% over Dedup−LRU. Especially, in Mixed_2, the difference between PCDedup and existing deduplication scheme was 10.11%, which was the largest. The main reason for large difference as follows. In Mixed_2, fingerprints that were never used before being evicted were mostly generated by Graph. PCDedup can detect the relatively low deduplication ratio of Graph's PCs by comparing the high deduplication ratio of RocksDB's PCs. After detection, PCDedup blacklisted all PCs of Graph, therefore the fingerprints generated by Graph cannot enter to the cache at all. However, the existing deduplication schemes always inserts the Graph's fingerprints that have high probability to be evicted without being used.

In order to evaluate the improvement of software overhead during deduplication process of PCDedup, we measured the number of fingerprint comparison during fingerprint searching. We also evaluated PCDedup-, which excluded the selective fingerprint searching of PCDedup. Fig. 5(c) shows the normalized number of fingerprint comparison for PCDedup, and PCDedup-. For this evaluation, the number of fingerprint comparison was calculated as follows: the number of searching entire cache × the number of fingerprints comparison per searching. As shown in Fig. 5(c), PCDedup- reduced the number of fingerprint comparisons on average by 31.4% due to reduced the first factor. Furthermore, PCDedup reduced the number of comparisons by 68.6% on average by reducing the both factors.

5 CONCLUSION

In this paper, we found the inefficiency of existing fingerprint cache management scheme based on fingerprint level information. To alleviate this inefficiency, we proposed a novel fingerprint management scheme, PCDedup, which can effectively manage fingerprint cache by predicting the duplicate pattern of an application through information of the I/O activity level. Based on this prediction, PCDedup is able to make a better decision in three main stages of the dedupliciation process, *cache insertion*, *victim selection*, and *fingerprint searching*. Our experimental results showed PCDedup reduced the total amount of written data over existing deduplication scheme by up to 16.4% (9.4% on average) while the fingerprint manage overhead is lowered on average by 68.6%. Our work in this paper can be extended in several directions. For example, in this paper, we assume that the fingerprint cache is kept only in the internal memory of the SSD. However, it will be an interesting extension to keep fingerprint cache also in NAND flash page of SSD and prefetch fingerprints before they used by predicting duplicate pattern of data.

6 ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (NRF-2015M3C4A7065645, NRF-2018R1A2B6006878 and NRF-2017R1E1A1A01077410). (*Corresponding Author: Jihong Kim*)

REFERENCES

[1] M. Goldman, K. Pangal, G. Naso, and A. Goda. 25nm 64gb 130mm2 3bpc nand flash memory. In *Proceedings of the International Memory Workshop (IMW)*, 2011.

[2] Z. Fan and D. Park. Extending ssd lifespan with comprehensive non-volatile memory-based write buffers. *Journal of Computer Science and Technology*, 34(1):113–132, 2019.

[3] F. Chen, T. Luo, and X. Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[4] T. Kim, S. Lee, and J. Kim. Finededup: A fine-grained deduplication technique for extending lifetime of flash-based ssds. *JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE*, 17(5):648–659, 2017.

[5] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing nand flash-based ssds. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[6] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. idedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[7] B. Mao, H. Jiang, Z. Wu, and L. Tian. Pod: Performance oriented i/o deduplication for primary storage systems in the cloud. In *Proceedings of the IEEE International Parallel DIstributed Processing Symposium (IPDPS)*, 2014.

[8] H. Wu, C. Wang, Y. Fu, S. Sakr, K. Lu, and L. Zhu. A differentiated caching mechanism to enable primary storage deduplication in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 29(6):1202–1216, 2018.

[9] R. Koller and R. Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage*, 6 (13):1–26, 2010.

[10] M. Li, H. Zhang, Y. Wu, and Z. Zhao. Prefetch-aware fingerprint cache management for data deduplication systems. *Frontiers of Computer Science*, 13(3):500–515, 2019.

[11] Facebook. RocksDB. http://rocksdb.org/, 2013.

[12] T. Kim, D. Hong, S. Hahn, M. Chun, S. Lee, J. Hwang, J. Lee, and J. Kim. Fully automatic stream management for multi-streamed ssds using program contexts. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[13] S. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. Bluedbm: An appliance for big data analytics. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[14] F. Zhou, J. Behren, and E. Brewer. Amp: Program context specific buffer caching. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2005.

[15] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[16] Stanford Large Network Dataset Collection. https://snap.stanford.edu/data/, 2017.