Freewrite: Creating (Almost) Zero-Cost Writes to SSD in Applications

Chunyi Liu[†,‡] Fan Ni[†] Xingbo Wu[†] Xiao Zhang[‡] Song Jiang[†]
[†]University of Texas at Arlington, Texas, USA [‡]Northwestern Polytechnical University, Xi'an, China
[†]{chun.liu@uta.edu, fan.ni@mavs.uta.edu, xingbo.wu@mavs.uta.edu, song.jiang@uta.edu}
[‡]{corey@mail.nwpu.edu.cn, zhangxiao@nwpu.edu.cn}

ABSTRACT

While flash-based SSDs have much higher access speed than hard disks, they have an Achilles heel, which is the service of write requests. Not only is writing slower than reading but also it can incur expensive garbage collection operations and reduce SSDs' lifetime. The deduplication technique can help to avoid writing data objects whose contents have been on the disk. A typical object is the disk block, for which a block-level deduplication scheme can help identify duplicate ones and avoid their writing. For the technique to be effective, data written to the disk must not only be the same as those currently on the disk but also be block-aligned.

In this work, we will show that many deduplication opportunities are lost due to block misalignment, leading to a substantially large number of unnecessary writes. As case studies, we develop a scheme to retain alignments of the data that are read from the disk in the file modifications by using small additional spaces for two important applications, a log-based key-value store (e.g., FAWN) and an LSM-tree based key-value store (e.g., LevelDB). Our experiments show that the proposed scheme can achieve up to 4.5X and 26% of throughput improvement for FAWN and LevelDB systems, respectively, with a less than 5% space overhead.

CCS Concepts

•Information systems → Data layout;

1. INTRODUCTION

Flash-based SSDs are high-performance storage devices whose access speed is much faster than hard disks. Increasingly more applications rely on its sustained high throughput and low latency to provide performance-sensitive services. These applications include databases [17, 14] and key-value stores [9, 10, 2]. In addition to being power efficient, a distinguished characteristic of SSDs is that they have much lower access latency and their throughput is much less sensitive to access sequentiality than hard disks.

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.

SYSTOR'17, May 22-24, 2017, Haifa, Israel

© 2017 ACM. ISBN 978-1-4503-5035-8/17/05...$15.00

DOI: http://dx.doi.org/10.1145/3078468.3078471

However, SSDs show an asymmetry between read and write accesses in multiple aspects which complicates use of the device. First, writes are slower than reads. For example, writing and reading a 4KB page may take 200 μs and 25 μs, respectively [11, 1]. Writes entail identification of a write address and updating the mapping table. Second, the flash does not allow page overwrite. A page has to be erased before new content can be written into it. Even worse, the erasure operation has to be conducted at a unit of block, which is much larger than a page (a block can have 64 or more pages [2]), and erasing a block takes around 1.5ms. If the block-level address mapping is used and a write request is issued to overwrite a page in a block, all live pages currently in the block must be migrated to an erased block, adding cost to the service of the write request. Third, writing can lead to expensive garbage collection operations, which in turn can degrade write and read performance. This issue is especially serious with lower-end SSDs whose over-provisioning space is limited and for an SSD which is nearly filled up. Garbage collection operations can also make SSD's performance less predictable [19]. Fourth, each flash page has as low as a few thousands P/E (program/erase) cycles, excessive writes compromise the SSD's lifespan [21].

Deduplication is an effective approach to avoid writes to the disk. Though it may compromise sequentiality of file block layout on the disk [22], it is not a concern for SSDs whose performance is less sensitive to the property. In many scenarios, block-level deduplication, which partitions a file into a number of fixed-size blocks and identifies duplicates among the blocks, is preferred to file-level deduplication, which attempts to find identical files, as the former is more likely to find redundancy and thus reduces more writes and saves more space [26]. Furthermore, block-level deduplication can be deployed at a position in the I/O stack close to the storage device [23, 7] or even within an SSD [6], so that the benefit of deduplication can be transparently received by the upper level software/applications.

While the block-level deduplication can potentially remove writes of redundant data to the disk, it requires the redundant data to be of block unit(s) long and block-aligned. A common on-disk data processing is to read files into the memory, insert new data and/or modify/delete existing data in the file, and finally write the updated files back to the disk. Often this process is conducted at byte offset(s) not aligned to the block boundaries. To keep data in a file contiguously laid out, or maintain a compact data layout, one may have to shift data not being updated in the file to make room for new data or eliminate the space left by the deleted data.

However, these shifted data, which are not modified and still have a copy on the disk, may become not block aligned and the deduplication system has to treat them as new blocks and also write them to the disk. One such scenario is that a user may read a (large) database file and then insert a new (small) record at the beginning of the file. This may cause the entire file to be written to the disk even with the deduplication functionality enabled.

To address the issue, we propose a design, named *freewrite*, that allows discontinuity in the file data layout, or leaves unused spaces, named *bubbles*, scattered in a file, so that the data do not have to be shifted and can remain block-aligned for the deduplication system to recognize the redundancy and avoid corresponding writes. Rationale behind the *freewrite* idea is that the time overhead for writing the bubbles to the disk and the space overhead for storing the bubble in the disk can be much smaller than those for re-writing the redundant data. In addition, by leveraging the deduplication system available at the lower level, applications can quickly incorporate *freewrite* to reduce writes without introducing metadata for managing locations of data blocks in their new layout. While overheads due to introduction of bubbles are expected to be small, a design challenge is to make sure that the benefit from reduction of writes (substantially) exceeds the overheads and/or the overheads are within limits specified by users.

In this paper we make the following three contributions:

- We identify an opportunity of reducing writes by leveraging the block-level deduplication system readily available in the kernel or in the SSD disk. By restructuring in-memory data layout before the data are written to the disk, we keep many redundant data block-aligned and avoid their writes to the disk.
- We design the *freewrite* scheme and implement it in two representative key-value stores (LevelDB and FAWN) to maximize saving of writes with minimal overheads.
- We have experimentally evaluated *freewrite* in the two applications with workloads of various characteristics. The measurements show that *freewrite* can improve throughout by up to 26% in the LevelDB system, by up to about 4.5X in the FAWN system.

2. THE DESIGN

The core idea of *freewrite* is to retain alignments of file data that have not been updated after they are loaded into the memory. To this end, we need to address three issues. First, one has to recognize these un-updated data and know where their original alignments are. Second, with the knowledge, one needs to develop a scheme determining a layout for the data to be written back so that the alignments can be retained and correctness of the data access is not compromised. Third, the overheads introduced by the scheme must be less than the performance gain and they can be capped by thresholds specified by users.

Freewrite can be implemented either in the kernel as a new feature of the file system or in individual applications. While an implementation in the file system can minimize programmers' efforts, it requires applications to disclose regions of a file that have been updated, or to supply the original files for the file system to derive knowledge about the

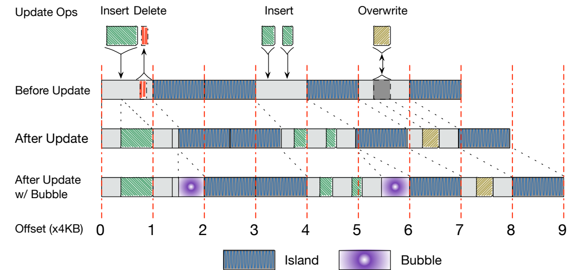


Figure 1: Example of file layouts after updates with or without using bubbles.

un-updated data. To this end, the POSIX interface would have to be changed, which is undesirable. On the other hand, only moderate effort is needed to incorporate *freewrite* into applications, and the effort can be well justified by its performance advantage.

A common procedure of processing data in the form of files on the disk is to load files into memory, update the files, and write the the updated files back to the disk, either to overwrite original files or write into newly created file(s). The update operations may be deletion of existing data, insertion of new data into the file(s), and overwriting existing data. While insertion and deletion do not modify data currently in the files, they may change positions of un-updated data in a file during the writeback, which probably makes the un-updated data un-block-aligned. By placing padding, which we name bubbles, into the file(s), *freewrite* can possibly keep them block-aligned for the deduplication at the lower level to reduce writes. Often the data processing is merely to re-organize the data with limited updates, so the reduction of writes due to the re-alignment can be significant.

To this end, *freewrite* takes two steps. First, it tracks all the updates that take place in each block retrieved from the on-disk file. When the data are ready to be written back, we know which blocks are modified, such as the first, fourth, and sixth blocks illustrated in Figure 1(Before Update), and which are not, such as the second, third, fifth, and seventh blocks in Figure 1(Before Update). To accommodate the updates, the current practice is to maintain a compact layout as illustrated in Figure 1(After Update), in which data are shifted to any byte-offsets in the file without leaving any holes or garbage data in the written file. While the unmodified blocks have copies on the disk, their alignments can be compromised due to the compact layout, leading to unnecessary writes. We name one or more contiguous unmodified blocks surrounded by modified blocks or at the file ends an *island*. With large islands, wastage of disk bandwidth and increase of service time can be significant. An island can stay block-aligned if a bubble (smaller than a block) is inserted, as illustrated in Figure 1(After Update with Bubbles).

Assume a block is of B bytes, a bubble is of $f * B$ bytes (where $0 < f < 1$), and the island immediately following the bubble is of k blocks large (where k = 1, 2, ...). By inserting such a bubble, *freewrite* removes writes of kB bytes at the expense of additional write of $f * B$ bytes and extra space of $f * B$ bytes on the disk. *Freewrite* has two thresholds to limit the overheads. One is to cap costs of individual bubbles

(f/k), named *bubble cap*, and the other is to cap total cost of bubbles in a file ($(\sum_{i=1}^{n} f_i) * B/S$, where n is bubble count and S is size of the file being written), named *global cap.* To be effective *freewrite* prefers a larger bubble cap to have a higher chance to retain islands' alignments, while the user wants to impose a limit on the total overheads with a pre-determined global cap. Once either cap is reached, *freewrite* does not insert a bubble before the current island. Note that there is at most one bubble between two adjacent islands. For file data whose updates and write-backs are concurrent, we use a buffer to hold the data pending for writing until an island is identified if the writes are asynchronous. Otherwise, recent island sizes are used to estimate the next one.

One complexity with *freewrite* is that metadata about the bubbles' locations and sizes must be recorded so that the bubbles can be removed when the file is read and be discounted when file offsets are computed. Persistency of the metadata incurs additional overhead. Fortunately, in many applications that maintain records in the file, bubbles are simply invalid records and no additional metadata maintenance is required.

3. IMPLEMENTATION

In this section, two key-value stores are used as examples to explain how *freewrite* can be adopted to avoid SSD writes.

3.1 Use of Freewrite in LevelDB for Higher Compaction Efficiency

LevelDB is a popular open-source key-value (KV) store based on log-structured merge (LSM) trees and plays a critical role in many data-intensive applications [5, 13, 12]. It uses a log structure to support high-performance sequential writes. However, LevelDB has to pay a very high cost to incrementally sort KV items distributed in various levels of a tree structure, or to perform compaction operation, to maintain an acceptable read performance [25]. Specifically, each level consists of a sequence of KV items sorted by their keys and segmented into one or a number of files, each of which is named an SSTable and is of about the same size (e.g., 2MB). When both are filled, Level $k+1$ can be ten times as large as Level k (k = 1, 2, ...). A compaction operation is to read one SSTable from Level k and multiple SSTables from Level $k+1$ covering a similar key range, merge-sort the two sorted lists of items, and write the resulting sorted list to the disk as multiple new SSTables. In this process, new KV items from Level k may be inserted into the list at Level $k+1$, and KV items at Level $k+1$ may be deleted by special DELETE KV items, or be replaced by items with the same keys from Level k. While the list at Level $k+1$ can be much longer than that at Level k (by up to ten times), we apply *freewrite* on the list at Level $k+1$ to minimize writes of its un-updated data back to the disk. The incorporation of *freewrite* is straightforward. As each KV item contains its size, *freewrite* does not need to track bubbles' positions in a file. Because LevelDB records KV items' positions in corresponding SSTable files and uses them to locate the items, *freewrite* does not need to track the bubble sizes. In addition, the way for LevelDB to maintain its on-disk metadata is not changed.

3.2 Use of Freewrite in FAWN for Higher Garbage Collection Efficiency

FAWN is a key-value store whose KV items are persistently stored on an on-flash data log file [2]. The item locations, or offsets in the log file, are tracked by an in-memory index structure, which is a hash table. The index can be rebuilt from the data log even if it is not persisted immediately. To this end, upon a delete request FAWN appends a DELETE marker at the log tail to invalidate the corresponding item in the log. In the log-structured store, any update operations for servicing delete and overwrite requests leave reclaimable spaces, or trash, in the log. Periodically garbage collection is carried out to copy all valid items to a new log file and then remove the old log file. In the process, some physical disk writes can be removed by a block-level deduplication if un-updated data can keep their block alignments in the old log. Respecting the two thresholds (bubble cap and global cap), *freewrite* opportunistically inserts bubbles into the new log to keep selected data blocks aligned. Similar to that in LevelDB, *freewrite* in FAWN does not need to maintain its own metadata about bubble locations and sizes. Instead, each bubble is self-identifiable by following a regular KV item's format with a special key that is outside of the valid key range. A side effect of the approach is that bubbles smaller than a key size cannot be used.

As files with bubbles inserted by *freewrite* cannot be directly used by programs unaware of the new file format, we built a tool to remove bubbles from a file and turn it into a regular one. For files constantly updated in a certain time period, *freewrite*'s advantage can be substantial if bubbled are allowed in the period.

4. PERFORMANCE EVALUATION

To evaluate *freewrite*'s performance, we implemented it into LevelDB [13] and FAWN [2], and extensively evaluate them to reveal its performance behaviors.

4.1 Experiment Setup

For the evaluation of *freewrite* in LevelDB, we instrument LevelDB 1.1 on its compaction operation to enable *freewrite*. In the experiments, the key size is fixed at 8 bytes. The value size of each item is 512 bytes unless stated otherwise. Each SSTable is 2MB large, and we send `PUT`s (writes) requests to the system until the KV store reaches 1GB. Among the `PUT`s, there are about 4% requests for repeated keys, which represent overwrites, or equivalently delete and then insert operations. A LevelDB store of 1GB usually has four levels, where compaction operations occur between every two adjacent levels. For the evaluation of *freewrite* in FAWN, we implement a prototype for a single server. We place the same workloads on the FAWN system as those on the LevelDB system. The measurements were taken after the store is created and when garbage collection is initiated. Unless stated otherwise, we set *bubble cap* at 12.5%, or 512B out of a 4KB block size, and the global cap is set at 5% of the total data size. To enable deduplication at the block layer, we use Dmdedup [23], a device mapper target in the kernel (Linux v4.7.1 in). The experiments were conducted on a Dell R630 server with two Xeon E5-2680v3 CPUs, each has twelve cores and 30MB LLC. The server is equipped with 128GB DDR4 memory, and a Samsung 840 EVO 1TB SSD.

To show *freewrite*'s effectiveness for improving applications' performance with various access patterns, we use three

key distributions in the workloads. In the first workload, all keys are uniformly distributed in a given key range. In the second workload, keys are also distributed in the range. However, they are clustered. Within each cluster they are 20 items of continuous keys. This represents the workload scenario where an object, such as a database file, is split into smaller KV items for storage with data spatial locality preserved. Google's BigTable is such an example [5]. The third workload is a mixture of the above two key distributions, where there are five uniformly distributed keys between every two key clusters. Unless stated explicitly, the experiment results are collected with the mixed workload in following discussions.

4.2 Experiment Results

Below we present the experiment results on the two systems with different workloads.

Throughput with different access patterns.

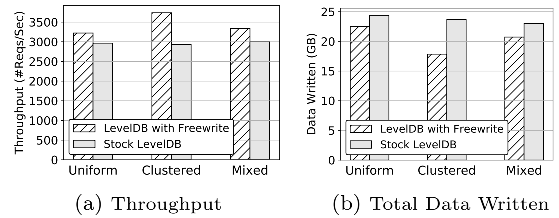
Figure 2 shows the throughput and total amount of data written to the disk, including those from users and caused by compaction operations, in LevelDB with and without *freewrite*. In the experiment the highest throughput improvement (about 22%) is seen with the workload of clustered keys, in which updates are concentrated in some smaller ranges and it is more likely that blocks are left entirely un-updated and *freewrite* can keep their alignments by inserting bubbles to remove their writes. In contrast, with a purely uniform key distribution, a data block at Level $k+1$ is more likely to receive update(s). From Figure 2b we can see that a higher percentage of data (about 24%) are kept from being written to the disk with the workload of clustered keys.

Similar trend is observed in the FAWN system in its GC operations shown in Figure 3. However, the throughput improvement and reduction of data written to the disk are much more significant (by up to 4.5X and 8.2X, respectively). One reason is that the measurements for the LevelDB systems include the initial execution phase where few compaction operations are triggered. For the FAWN systems the measurements are specifically about the efficiency of the GC operation.

To understand the impact of *freewrite* on the performance of LevelDB in the different phases of the store building, we show its throughput at different execution time periods in Figure 4. At the beginning, there is little difference between the systems with and without *freewrite* because there are few compaction operations at this time. When the store grows, the compaction operations are more frequent, and the throughput decreases significantly due to the high write amplification introduced by the compaction (about 20 in our measurements as shown in Figure 2b). At this time, *freewrite* helps to improve the throughput as some blocks' writing can be removed. With *freewrite*, the execution time of building a 1GB LevelDB store is reduced by about 21% (from about 683 seconds to about 539 seconds).

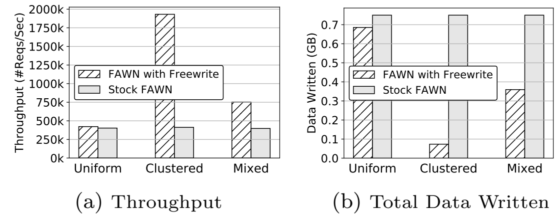
Throughput with different KV item sizes.

To understand impact of KV item size on the effectiveness of *freewrite*, we change the value size from 128 bytes to 1K bytes in the experiments for LevelDB and FAWN. Figure 5a shows the throughput and amount of data written to the disk with different value sizes for the LevelDB systems. The



(a) Throughput (b) Total Data Written

Figure 2: Throughput and amount of data written to the disk in LevelDB with and without *freewrite* for different access patterns.



(a) Throughput (b) Total Data Written

Figure 3: Throughput and the amount of data written to the disk in FAWN with and without *freewrite* for different access patterns.

highest improvement, which is about 26%, is achieved with a reduction of about 29% of written data at the value size of 1KB. Figure 6 shows the throughput and amount of written data for the FAWN systems. The performance trend is similar, except that the improvement is more significant for FAWN. The highest improvement is about 2.5X and more than 75% of written data are removed when the item size is 1KB. When the KV item size increases, the number of updates in each compaction becomes smaller and fewer blocks are updated. Correspondingly more blocks are un-updated and are available for *freewrite* to retain their alignments and to remove their writing.

Sensitivity analysis.

Although inserting bubbles for keeping un-updated blocks' alignments creates opportunities for deduplication, it may increase I/O and space overheads. While we keep the global cap at a default 5%, we vary bubble cap, or the allowed maximum bubble size in term of its percentage of the block size, to see its impact on throughput improvement and total space overhead for the FAWN system. The results are shown in Figure 7a and Figure 7b, respectively. With a larger bubble cap, there are more opportunities to insert bubbles to keep un-updated data blocks aligned and then de-duplicated (as shown in Figure 8), and accordingly there are higher improvements. When the threshold is 96%, the improvement of throughput is about 1.9X while still less than 5% disk space is used for bubbles.

5. RELATED WORK

The importance of reducing writes to the SSD to improve I/O performance and to extend their lifespan are well recognized and many efforts have been made. Addressing the same issue as we do, studies related to this paper include efforts on reducing write requests to the SSD and reusing existing data on the SSD devices.

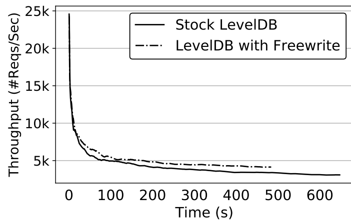
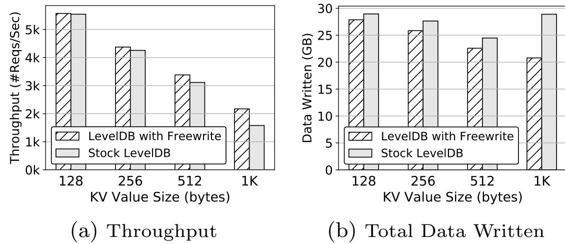
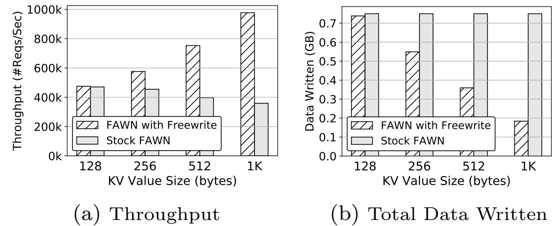


Figure 4: Throughput of the LevelDB systems with and without *freewrite* for the workload with mixed key distribution at different phases of the execution.



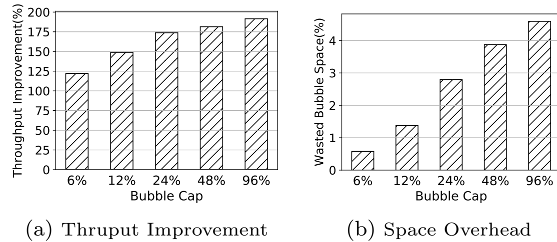
(a) Throughput (b) Total Data Written

Figure 5: Throughput and amount of data written to the disk in the LevelDB systems with and without *freewrite* with different value sizes.



(a) Throughput (b) Total Data Written

Figure 6: Throughput and amount of data written to the disk in the FAWN systems with and without *freewrite* with different value sizes.



(a) Thruput Improvement (b) Space Overhead

Figure 7: Throughput improvements and the space overhead in terms of percentage of the disk space used for storing bubbles with different bubble caps in the FAWN system. The workload with the mixed key distribution.

Efforts on Reducing Writes to the SSD.

Realizing small updates to on-SSD blocks may not justify re-writing entire blocks, I-CASH redirects the updates to the hard disk and uses the SSD to store infrequently updated and mostly read blocks [28]. Similarly, Delta-FTL redirects small updates, the deltas, to an assumed non-volatile memory and leverages locality in the workloads to merge multiple writes into one block write [24]. In the work reading one block may require two reads to different devices and re-construction of the block. Instead of re-directing small updates, Griffin identifies and redirects access of frequently updated blocks to the hard disk to reduce writes to the SSD [21]. However, as the hard disk has a much worse I/O performance than the SSD, performance of applications is compromised. In contrast, leveraging existing block-level deduplication *freewrite* trades a fraction of SSD space for potentially much reduced writes.

Efforts on Reusing Data on the SSD.

Essentially *freewrite* reuses data currently on the SSD to avoid writes. Data deduplication has been widely used in various storage systems to achieve the same goal [15, 6, 18]. For this effort made at the block level *freewrite* is a complementary technique to create opportunities for it to be more effective. If the constraint of block-based operations is removed, one alternative to *freewrite* is to introduce content-defined-chunking (CDC) based deduplication [3, 16, 27] into applications to detect, manage, and avoid duplications at variable-sized chunks. However, the CDC-based duplication detection and management of metadata can be very expensive [8]. Compared to *freewrite*, modifications to applications required by CDC-based deduplication can be extensive. VT-trees [20] uses a variant of LSM-tree to avoid copying of unchanged data blocks from old SSTables to new ones, so that writes during compactions can be reduced. However, it needs to maintain an additional layer of index structures in the application and may require extensive modifications to the application. In contrast, *freewrite* leverages readily available off-the-shelf block-level deduplication at the storage system to minimize extra efforts in the application to reduce writes, and thus makes the technique highly applicable. NetApp [4] proposed to pad zeros at the end of modified files in a file backup volume to retain other files' block alignment for effective fixed-size block deduplication. The padded zeros serve a purpose similar to that of bubbles in *freewrite*. However, insertion of bubbles is not limited at the end of a file to only retain alignments of files following it. For large files, retaining alignments for blocks within a file is also important. To this end, *freewrite* produces more deduplication opportunities by using a more intelligent bubble management technique within individual files.

6. CONCLUSIONS

In the paper, we propose *freewrite* to enable almost zero-cost writes to SSDs. With *freewrite*, alignments of un-updated blocks to be written to the disk are retained, which creates deduplication opportunities for the readily available block-level deduplication system at a lower system level and significantly reduces writes to the SSD. We experimentally demonstrate *freewrite*'s efficacy in two widely used KV store systems. The results show that it achieves substantial performance improvements with low space overheads.

7. ACKNOWLEDGMENTS

We are grateful to the paper's shepherd, Philip Shilane, and anonymous reviewers who helped to improve the paper's quality. This work was supported by US National Science Foundation under CNS 1527076 and partially by NSF of China under Grant No.61472323.

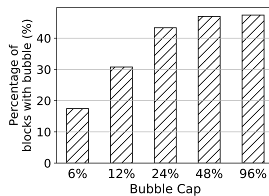


Figure 8: The percentage of blocks with bubble inserted when different bubble caps are used.

8. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 57–70, 2008.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14. ACM, 2009.

[3] A. Z. Broder. Some Applications of Rabin's Fingerprinting Method. In *Sequences II*, pages 143–152. Springer, 1993.

[4] P. Buschman. Deduplication and Incremental Acceleration in Bacula with NetApp Technologies, 2012.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[6] 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 9th USENIX Conference on File and Stroage Technologies*, 2011.

[7] Z. Chen and K. Shen. Ordermergededup: Efficient, Failure-Consistent Deduplication on Flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 291–299, 2016.

[8] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao. Quicksync: Improving Synchronization Efficiency for Mobile Cloud Storage Services. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 2015.

[9] B. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-Value Store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, 2010.

[10] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 25–36. ACM, 2011.

[11] C. Dirik and B. Jacob. The Performance of PC Solid-State Disks (SSDs) As a Function of Bandwidth, Concurrency, Device Architecture, and System Organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 279–289. ACM, 2009.

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

[13] S. Ghemawat and J. Dean. Leveldb. *https://github.com/google/leveldb*, 2011.

[14] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 529–540. ACM, 2014.

[15] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H.-u. Lee, S. Kang, Y. Won, and J. Cha. Deduplication in SSDs: Model and Quantitative Analysis. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12. IEEE, 2012.

[16] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal Content Defined Chunking for Backup Streams. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 18–18, Berkeley, CA, USA, 2010. USENIX Association.

[17] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086. ACM, 2008.

[18] D. Meister and A. Brinkmann. dedupv1: Improving Deduplication Throughput using Solid State Drives (SSD). In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–6. IEEE, 2010.

[19] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-Defined Flash for Web-scale Internet Storage Systems. *SIGARCH Comput. Archit. News*, 42(1):471–484, Feb. 2014.

[20] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building Workload-Independent Storage with VT-trees. In *11th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 17–30. USENIX Association, 2013.

[21] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD Lifetimes with Disk-based Write Caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10. USENIX Association, 2010.

[22] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: Latency-Aware, Inline Data Deduplication for Primary Storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[23] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok. Dmdedup: Device Mapper Target for Data Deduplication. In *Ottawa Linux Symp., Ottawa, Canada*, 2014.

[24] G. Wu and X. He. Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 253–266. ACM, 2012.

[25] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: an LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, pages 71–82. USENIX Association, 2015.

[26] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou. A

Comprehensive Study of the Past, Present, and Future of Data Deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[27] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou. Ddelta: A Deduplication-inspired Fast Delta Compression Approach. *Performance Evaluation*, 79:258–272, 2014.

[28] Q. Yang and J. Ren. *I-CASH*: Intelligently Coupled Array of SSD and HDD. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011.